Reducing Exception Management Overhead with Software Restart Markers

by

Mark Jerome Hampton

Bachelor of Science in Computer and Systems Engineering
Bachelor of Science in Computer Science
Rensselaer Polytechnic Institute, May 1999

Master of Science in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, June 2001

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

Author . . . .
Department of Electrical Engineering and Computer Science
February 1, 2008

Certified by . . . .
Krste Asanović
Associate Professor
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chair, Department Committee on Graduate Students

Reducing Exception Management Overhead with Software Restart Markers

by

Mark Jerome Hampton

Submitted to the Department of Electrical Engineering and Computer Science on February 1, 2008, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

Modern processors rely on exception handling mechanisms to detect errors and to implement various features such as virtual memory. However, these mechanisms are typically hardware-intensive because of the need to buffer partially-completed instructions to implement precise exceptions and enforce in-order instruction commit, often leading to issues with performance and energy efficiency. The situation is exacerbated in highly parallel machines with large quantities of programmer-visible state, such as VLIW or vector processors. As architects increasingly rely on parallel architectures to achieve higher performance, the problem of exception handling is becoming critical.

In this thesis, I present software restart markers as the foundation of an exception handling mechanism for explicitly parallel architectures. With this model, the compiler is responsible for delimiting regions of idempotent code. If an exception occurs, the operating system will resume execution from the beginning of the region. One advantage of this approach is that instruction results can be committed to architectural state in any order within a region, eliminating the need to buffer those values. Enabling out-of-order commit can substantially reduce the exception management overhead found in precise exception implementations, and enable the use of new architectural features that might be prohibitively costly with conventional precise exception implementations. Additionally, software restart markers can be used to reduce context switch overhead in a multiprogrammed environment.

This thesis demonstrates the applicability of software restart markers to vector, VLIW, and multithreaded architectures. It also contains an implementation of this exception handling approach that uses the Trimaran compiler infrastructure to target the Scale vector-thread architecture. I show that using software restart markers incurs very little performance overhead for vector-style execution on Scale. Finally, I describe the Scale compiler flow developed as part of this work and discuss how it targets certain features facilitated by the use of software restart markers.

Thesis Supervisor: Krste Asanović
Title: Associate Professor

Acknowledgments

As I come to the end of this phase of my life, it seems insufficient to try to distill several years of interactions and relationships into a handful of paragraphs. Thankfully this isn't like an awards show where the producers will start playing music to cut off winners who take too long with their speeches. I'm free to ramble as long as I want. But in the spirit of "less is more," I will try to succinctly encapsulate my gratitude to those who have helped me get to this point. Of course, given my penchant for being long-winded, my "less" will almost certainly be somebody else's "more."

First and foremost, I have to give thanks to God the Father, the Giver of every good and perfect gift, and to Jesus Christ, who strengthens me daily. They richly deserve glory for anything I have achieved during my tenure in graduate school, and for anything that I may achieve in the future.

In this earthly realm, I of course have to start with my advisor, Professor Krste Asanović. When I first came to MIT, I had no idea who he was, and was initially intent on trying to secure another professor as my advisor. However, after I saw Krste give a presentation on the work he was doing, I knew which group I wanted to join. And I've never regretted my decision. Many of my former group members and colleagues in other groups have used all kinds of superlatives to describe Krste in their own thesis acknowledgments, and all of those descriptions hold true. Krste's breadth of knowledge, his ability to digest and analyze new material at a rapid rate, and his seemingly inexhaustible supply of energy that allows him to juggle a countless number of projects, are all, in a word, "scary." And yet he was willing to tolerate the inadequacies of a grad student like myself, who came to MIT with relatively little understanding of computer architecture. Even though I may know much more now (in large part thanks to him), I still can't hold a candle to his knowledge about such a vast array of subjects. I want to thank him for all of his support through the years (not just financial, although that is greatly appreciated as well), for fostering a great research group environment, and for indulging my fascination with exception handling.

I thank Professor Saman Amarasinghe and Professor Arvind for serving on my thesis committee and for providing feedback on my work.

I have interacted with many fellow students over the years who have enriched my experience in grad school. I want to begin by acknowledging my two original officemates, with whom I have developed valuable relationships. Since I listed them in one order in the Acknowledgments section of my Master's thesis, I list them in reverse order here.

Jessica Tseng was the first person in Krste's group whom I contacted when I came to MIT. She showed me how to use the simulator that I needed for my work, and then she subsequently became one of my officemates—and a very tolerant one at that, as she put up with my singing in the office. Even after we moved to different offices, I enjoyed being able to walk downstairs and strike up a conversation when I didn't feel like doing any work (which occurred frequently). I want to thank her for her friendship and support.

Mike Zhang, my other original officemate, is truly passionate about his convictions, including his love of the Yankees. Our frequent arguments about creation vs. evolution were definitely interesting, and while it's probably not the effect he intended, they also strengthened my faith. But at the end of the day, even though he may have viewed my religious convictions as silly, and I may have viewed his atheist beliefs as silly, we still respected one another, which was great.

Ronny Krashinsky entered MIT at the same time I did, earned his Master's degree at the same time I did, but then decided to break the mold and become a doctor before me. Given the vast amount of work that he did for his thesis, the fact that he was able to complete it when he did is truly awe-inspiring. I'm sure that no matter what lies in store for him, he will have a very successful future, and who knows, maybe someday I'll be working for "Ronny, Inc." after all.

Heidi Pan, even though you decided you didn't want to be my officemate anymore (just kidding), I appreciate the time that we did share an office—even if you didn't want to get roped into the crazy discussions that the rest of us had.

Steve Gerding, I'm very glad that Jessica, Mike, and I had to kick out our imaginary officemate and actually let a real person occupy the fourth spot in our office. Your devotion to all things related to Los Angeles—especially UCLA—is pretty scary, but also admirable in its steadfastness.

Sam Larsen was responsible for the bulk of the compiler front end work that made it possible for me to use SUIF and Trimaran together. His thesis work also provided a template for my parallelization phase. Beyond those contributions, I thank him for the sense of humor he brought to the various discussions which took place in our office, even though he frequently looked so serious.

I thank Mark Stephenson—a.k.a. "The Fastest Man in Computer Architecture"—not only for allowing me to witness his speed on the track (unofficially), but also for facilitating the only opportunity I had to practice my thesis defense. While I had hoped for a smoother presentation than actually occurred, the feedback I received was invaluable.

Ian Bratt created the initial port of Trimaran for the RAW architecture that I was subsequently able to adapt for Scale.

I thank all of the past and present members of Krste's group, including Seongmoo Heo, Chris Batten, Ken Barr, Emmett Witchel, Albert Ma, Rose Liu, Jae Lee, and Asif Khan. I also thank other lab members whom I have had the pleasure to meet, including Rodric Rabbah, Mike Gordon, and Michal Karczmarek.

Outside of the lab, my primary social and support network on campus consisted of members of the Black Graduate Student Association (BGSA) during the first half of my graduate career, and members of the Academy of Courageous Minority Engineers (ACME) during the second half of my graduate career. I thank BGSA for giving me a much-needed outlet for relieving stress, and I especially thank the members who helped convince me to

stay at MIT beyond my Master's degree and to pursue a Ph.D. I thank ACME for making me accountable for making forward progress in my work, for the support and encouragement, and for the weekly free food. I also thank the Graduate Students Office, particularly Dean Isaac Colbert, Dean Blanche Staton, and Dean Christopher Jones, for the support that they offered to both BGSA and ACME, not only collectively but also individually.

Thanks to Mary McDavitt for doing so much behind the scenes to help the group run smoothly. She truly goes above and beyond the call of duty. I also thank the previous administrative assistants for the group, Shireen Yadollahpour and Maria Ruiz.

Thanks to the members of my church congregation, who have enriched my life in many different ways.

A special thanks to my various family members who kept asking me when I was going to graduate and get a real job, thus motivating me to get things done so I didn't have to hear the questions anymore...well, at least the questions about graduation. Seriously, though, I thank all of my family for their love.

I owe so much to my parents, who have given me constant love, support, and encouragement through this whole process. Mom and Dad, thank you for always being there for me and for being examples who I can look up to, learn from, and try to emulate.

Finally, Emily, what can I say? We've been through so much together, and I thank you for all of the wonderful things that you've brought to my life. Thank you for always wanting the best for me, and for helping me to become a more complete person.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the continual quest to push the boundaries of high-performance, energy-efficient computing, designers employ a variety of techniques across multiple levels of the system hierarchy. The rapid rate of technology improvements enables increasing numbers of transistors to be placed on a chip. New microarchitectural approaches to hide memory access latency and reduce power consumption are introduced on a yearly basis. Novel architectures are developed that can exploit multiple forms of parallelism. Compiler writers find ways to achieve higher levels of performance in various applications. However, with all of the advancements in these various fields, one aspect of the computer system has remained largely unchanged over the years: exception handling.

Any processor needs some type of exception handling mechanism, whether it is used to deal with unanticipated events such as program errors or to implement features such as virtual memory. Where processors frequently differ is in the extent of their support for various kinds of exceptions. In general-purpose systems, support for a wide range of exceptions is mandatory. By contrast, embedded processors or supercomputers typically handle a more limited set of exceptions, as more extensive support is either not required or too costly.

Although different processors may provide varying levels of exception support, the actual methods they use to handle exceptions are generally quite similar. After the development of a handful of exception handling approaches many years ago, there has been little innovation in this field. Even seemingly different exception handling mechanisms, which are unique in certain respects, are usually lumped into one of two categories: those that are *precise*; and those that are *imprecise*. The precise exception model makes it appear as though a processor executes instructions one at a time in program order, regardless of the underlying hardware implementation. This exception model has provided many benefits over the years, and is widely considered to be the gold standard for dealing with exceptions. However, these benefits come at a cost, and recent shifts in the landscape of computing are making the problems of precise exception support more apparent. There are two recent trends in the computer industry which are combining to increase the difficulty of handling exceptions precisely. The next two sections discuss those trends.

1.1 The Decline of Out-of-Order Superscalars

For many years, processor design for general-purpose computing was characterized by increasing hardware complexity. Out-of-order superscalar execution was the paradigm of choice, and a reliance on clock frequency scaling led to deeper and deeper pipelines. Keeping the pipelines full with useful work required larger and more complicated hardware: bypass networks, branch predictors, and register files were just some of the structures that continued to grow in complexity.

This approach eventually ran into a dead end in the form of the "power wall," which caused Intel to change its processor roadmap [Wol04] and begin developing *multicore* designs that place multiple processors on a single die. A problem with using complex superscalar execution techniques is that they are generally not energy-efficient [GH96], leading to serious issues with scalability. As a result, many researchers now predict that future processors will increasingly employ simpler in-order pipelines, particularly as the computer industry goes beyond multicore architectures to *manycore* architectures, which will have thousands of processors on a die [ABC$^+$06]. This issue has motivated the design of processors such as Cell [Hof05], which forego the power-hungry structures associated with out-of-order execution and instead rely on static scheduling.

The shift away from out-of-order execution has implications for exception handling, as the same hardware structures used to implement techniques such as register renaming and branch misprediction in dynamically-scheduled superscalar designs are also used to enable precise exceptions. Thus, one could view precise exception support as "cheap" in a sense for a system that already implements performance-enhancing features such as register renaming. By contrast, a statically-scheduled in-order design would not implement precise exceptions using these hardware structures, and in general would not contain the structures at all. However, implementing precise exceptions in an in-order processor would require bypassing logic to avoid pipeline stalls. In a very simple scalar pipeline that implements a sequential architecture, the overhead of the bypass network is typically manageable, and bypassing is often in place already to avoid stalls due to data dependences. When dealing with an explicitly parallel architecture that can issue multiple operations simultaneously, supporting precise exceptions becomes more expensive, and this issue leads to the next computing trend that is troublesome for current exception handling approaches.

1.2 The Rise of Explicitly Parallel Architectures

Superscalar implementations of sequential architectures exploit program parallelism at runtime by using hardware scheduling techniques. Thus, although the actual instruction set architecture (ISA) may only specify one operation in any given time slot, that does not prohibit the underlying design from issuing multiple operations simultaneously, providing a form of "implicit" parallelism. By contrast, explicitly parallel architectures such as vector or VLIW processors extract parallelism at compile-time, and avoid the need for complex dynamic scheduling approaches. These types of designs have been used in the embedded and scientific processing domains for many years. However, they have only recently become more widely used in general-purpose systems—albeit in a somewhat limited form—with the advent of short-vector multimedia extensions such as Intel's MMX/SSE [PW96, RPK00].

Explicitly parallel architectures will likely become more widely used in the future. A key motivator for the increased interest in parallel architectures is the fact that issuing operations in parallel allows a design to maintain a constant level of performance while lowering the supply voltage; this technique reduces power as well as the total energy dissipated [CSB92]. However, this fact by itself was not sufficient to motivate the widespread use of explicitly parallel architectures in the past, as the applications that drove processor design were often not amenable to static compilation techniques. Previously, the SPECint benchmark suite [spe] was the primary workload used to gauge the effectiveness of new architectural and microarchitectural approaches. SPECint benchmarks typically have unpredictable branches, making them difficult for parallelizing compilers to handle. Thus, in order to effectively achieve parallel execution of these applications, the burden of extracting parallelism had to be placed almost entirely on the hardware, leading to complex branch predictors and high degrees of speculation. Looking forward, many of the applications that will be targeted by future processor designs have significantly more parallelism than these SPECint-style programs. Several researchers from Berkeley have developed 13 "dwarfs," or classes of applications, which can be used to guide architectural innovation [ABC^{+}06]. 12 of the 13 dwarfs have some form of parallelism and correspond to a variety of benchmarks across multiple application domains. Explicitly parallel architectures can often be used to target the parallelism inherent in these applications in a more energy-efficient manner than conventional out-of-order superscalars.

Another reason for the rise of explicitly parallel architectures in general-purpose computing is the fact that the traditional boundaries between computing domains are being blurred. In the past, there were reasonably well-defined characteristics that distinguished embedded, desktop, and server systems. While each processing domain is still distinct in some respects, the number of similarities between these categories is increasing. Low-end servers are essentially desktop PCs [HP07, Appendix D]. Asanović et al. [ABC^{+}06] point out several characteristics that embedded and high-performance server systems have in common, including sensitivity to power consumption. Perhaps the most notable blurring of boundaries has occurred as greater numbers of embedded systems have taken on the general-purpose characteristics of desktop processors. Traditionally, embedded processors were designed to perform a single task as cost-effectively as possible. While this characteristic is still true of many embedded systems, the rise of "convergence" devices in the cell phone, PDA, and video game console markets has necessitated embedded processors that can adapt to a variety of applications. These multifunction designs are often remarkably similar to desktop machines in terms of the features they support, making it difficult to draw a sharp line between the embedded and general-purpose domains [FFY05]. This fact has significant implications for future processors, as embedded systems are more sensitive to power consumption than desktop machines, further motivating the use of energy-efficient parallel architectures. Additionally, embedded memory access latencies are typically more predictable than in desktop PCs, facilitating the statically-scheduled execution that is typically found in parallel architectures.

The above shifts in the computing industry are leading to a surge of interest in explicitly parallel architectures. Recent architectures have been developed in both academia and industry to statically exploit multiple forms of parallelism, including the TRIPS architecture [SNL+03], the Scale vector-thread architecture [KBH+04a, KBH+04b], and the Cell Broadband Engine [Gsc07]. However, parallel architectures cause problems for precise exception handling mechanisms, one of which is the conflict caused by the fact that the precise exception model is designed to hide the details of a processor implementation, while architectures such as VLIW expose processor details to the user. The difficulty of handling exceptions precisely in parallel architectures has led to a lack of precise exception support in many designs, but this has typically come at the cost of certain features such as virtual memory which are essentially required in any general-purpose system.

1.3 Contributions

The decline of out-of-order superscalar processors—which use hardware techniques to implicitly exploit parallelism in a sequential ISA—and the corresponding increased interest in explicitly parallel architectures will likely make it more difficult to provide precise exception support in many future designs. These trends motivate research into alternatives to the precise exception model. The primary objective of this thesis is to develop an exception model for explicitly parallel architectures that avoids the overhead of handling exceptions precisely. This alternative exception model is based on the concept of *software restart markers*, which are indicators that the compiler places in the program to show where a process can safely be restarted after an exception is handled. Although software restart markers can be used within multiple processor domains, I mainly focus on using this model to enable support for general-purpose features in embedded systems. I also restrict the scope of this work to exception handling within a single core. This thesis makes the following contributions:

- **Classification of exception handling mechanisms:** I describe a more comprehensive classification of exception handling mechanisms than simply precise or imprecise. Defining these characteristics enables a more accurate understanding of what different exceptions actually require from an exception handling mechanism. Some authors [FFY05, HP07] have incorrectly stated that precise exceptions are needed to support certain features such as virtual memory. The classification of exception handling mechanisms presented in this thesis shows why this is not true.

- **Software restart markers:** I show how the software restart markers model can be used to handle exceptions, with a particular focus on explicitly parallel architectures. Software restart markers reduce the exception management overhead of precise exception implementations by permitting instructions to *commit out of order* while also introducing the notion of *temporary state* which does not have to be preserved in the event of a context switch. An exception handling mechanism based on software restart markers has the necessary characteristics to enable features such as virtual memory.

- **Software restart marker designs for different forms of parallelism:** I show how software restart markers can be used in vector, VLIW, and multithreaded archiectures. Each of these designs corresponds to a different usage model for software restart markers, and they collectively demonstrate that this exception handling approach is not restricted to a particular type of architecture.

- **Implementation of software restart markers in the Scale architecture:** I evaluate software restart markers in a specific processor design by implementing the model for the Scale vector-thread architecture, using the Trimaran compiler infrastructure. I show that the total performance overhead of using software restart markers to enable virtual memory support when executing vector-style code on Scale is less than 4% compared with a design that does not support virtual memory.

- **Compiling for the Scale architecture:** Since a key motivation for this thesis is the claim that parallel architectures will be more widely used in the future, I present evidence to show the benefits that can be achieved from this approach. I implement various phases of a compiler infrastructure for Scale and show that the compiler can parallelize types of code that are difficult for established vector or VLIW compilers to handle. I also target specific features of the Scale architecture that would be costly to support under the precise exception model.

1.4 Thesis Overview

Chapter 2 provides an overview of exceptions. It describes the different exceptions that might be supported in a system and discusses the popular precise exception model. While there are a variety of precise exception implementations, they suffer from several different sources of overhead. The chapter categorizes the different types of overhead to motivate the development of a new exception model. To assist in this development, exception handling mechanisms are classified according to several different axes. The requirements imposed by different exceptions are then analyzed, and the desirable characteristics of a new exception model are presented.

Chapter 3 describes the software restart markers model of exception handling. This model relies on the compiler to create regions of code in which it is possible to commit instructions out of order. Additionally, another advantage of software restart markers is that they can reduce context switch overhead through the use of temporary state. The chapter discusses issues with using software restart markers, and also presents three different usage models, which correspond to the designs presented in Chapters 4 through 6. These designs are intended to demonstrate the applicability of software restart markers to a wide range of architectures.

Chapter 4 shows how software restart markers can be used in vector architectures to support features such as virtual memory without incurring the overhead of previous approaches. Additionally, various optimizations are presented to reduce the performance overhead of software restart markers. The techniques described in this chapter are also applicable to other architectures.

Chapter 5 presents a design for using software restart markers in VLIW architectures. It shows how the use of this exception model can enable exception handling in sections of code for which designers often disable exceptions altogether. It also demonstrates a way in which the number of registers in a VLIW architecture can be reduced without hurting performance.

Chapter 6 shows how software restart markers can be applied to multithreaded architectures. It presents an example of how reducing the per-thread architectural register usage can enable a greater number of threads.

Chapter 7 presents a compiler implementation of software restart markers within the context of the Scale vector-thread architecture. It shows the compiler modifications necessary to support software restart markers for vector-style code, and also describes a potential approach to implementing software restart markers for threaded code.

Chapter 8 presents data on the performance overhead of using software restart markers to support virtual memory in the Scale architecture. It shows that this approach only incurs a small performance degradation.

Chapter 9 summarizes the contributions of the thesis and discusses future work.

Appendix A discusses the overall Scale compiler flow that was developed as part of this work, and also shows how the compiler can target certain features that are facilitated by the use of software restart markers.

Chapter 2

Exceptions Overview

In this chapter, I first define exception terminology, and then present a definition and implementations of the precise exception model. I discuss the sources of overhead from implementing precise exceptions, and introduce a more comprehensive classification of exception handling mechanisms than simply precise or imprecise. Finally, I show that certain key exceptions do not require precise exception support, and I present the criteria that should be satisfied by a new exception model.

2.1 Exception Terminology

Since there is no universally agreed upon terminology in the field of exceptions, this section defines certain terms that will be used in this thesis.

An *exception* (also commonly referred to as an *interrupt*, *fault*, or *trap*) is an event that interrupts the normal flow of program execution. When an exception is *taken*, the current process is stopped and control of the processor is transferred to another thread, the *exception handler*. Usually the handler is part of privileged operating system (OS) code, but this is not a requirement.

An *asynchronous* exception occurs independently of the program being executed, and can be taken at any convenient time after it is detected (although there is sometimes an upper bound placed on the latency to take an asynchronous exception). By contrast, a *synchronous* exception is caused by some action of the program being executed, and thus the ordering of when it is taken must be specified relative to normal instruction execution. Although some classifications describe an exception as a synchronous event and an interrupt as an asynchronous event, the term "exception" is used to describe both types of events throughout this thesis.

A *terminating* exception requires program execution be terminated, as there is no way for a software handler to restart the process once the exception is taken. An exception is *restartable* if it can be handled and then the process can be resumed from a point that ensures correct execution. An exception is *swappable* if its handling normally involves saving all of the process state to memory so that a second process can be executed, and then later restoring the original process state and resuming that process after the exception has been handled. This swapping of processes is referred to as a *context swap*, or more typically a *context switch*. (The latter term is used throughout this thesis.) Figure 2-1 shows how the mechanisms required to support each class of exception are related to each other. (Note that support for a particular class of exception does not imply that a mechanism can handle every exception within that class—e.g. support for restartable exceptions only means that a mechanism has the general capability to handle an exception and then restart the process.) A system that supports swappable or restartable exceptions will also support terminating exceptions, as the handler simply needs to kill the process upon detecting a terminating exception, rather than the more complex task of handling the exception and restarting the process when dealing with a restartable or swappable exception. Support for swappable exceptions also implies support for restartable exceptions, as handling a swappable exception requires the ability to restart the process. Note that the reverse is not true—i.e. support for restartable exceptions does not imply support for swappable exceptions. For example, there may be some exposed pipeline state that cannot be saved to memory, but if the exception can be handled without modifying this state, the process can be restarted.

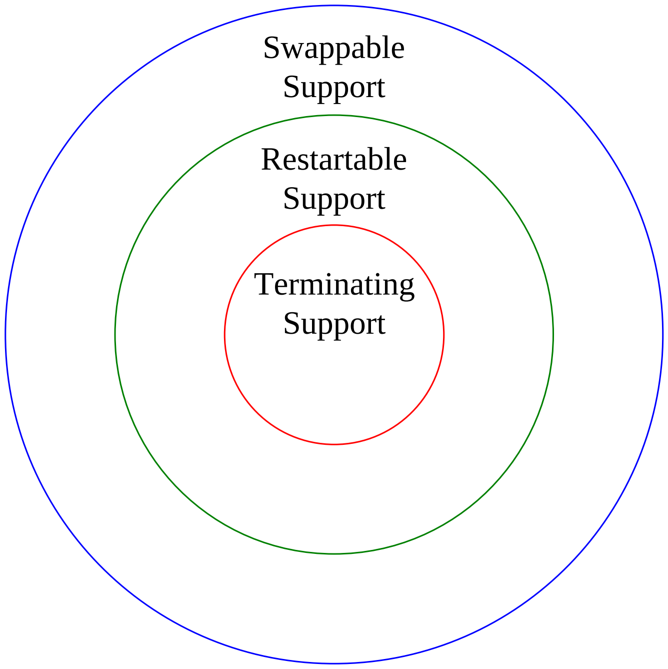


Figure 2-1: Venn diagram showing the relationship between support for terminating, restartable, and swappable exceptions.

For an illustration of the basic exception handling process, first consider Figure 2-2(a). This figure shows a scalar instruction stream being executed *sequentially*—only one instruction is being executed at any given time—and *serially*—instructions are executed in program order—with no exceptions occurring during program execution. (The definitions for sequential and serial execution are taken from Walker and Cragon [WC95].) Executing scalar instructions both sequentially and serially at run-time is consistent with the usage of a *sequential architecture*, in which instructions contain a single operation and the compiler or programmer only specifies one instruction at a time (serialization is implicit in the fact that the instructions are listed in a particular order). Figure 2-2(b) shows an alternate sequence of events in which instruction `i4` causes an exception in cycle 4, causing the current process to stop execution so that the exception handler can be run. A special machine register, the exception program counter (EPC), is loaded with the address of the excepting instruction before the exception is handled, as shown in Figure 2-2(c). If the process is to be restarted after handling the exception, then the system can load the program counter (PC) with the address contained in the EPC, and continue executing as normal, which is shown in Figure 2-2(d). The exception handler is responsible for preserving the appropriate process state so that the process can be safely resumed after the exception is handled. As mentioned earlier, for a swappable exception, the handler will save and restore all of the process state. However, even when dealing with a restartable exception, if the handler needs to overwrite any registers to perform its task, it must first save the original register values to memory and then later restore those values before restarting the process. Note that for certain types of exceptions, the excepting instruction should not be re-executed. For example, as discussed in the next section, an unimplemented instruction might need to be emulated in software. In this case, the exception handler can modify the EPC register to point to the subsequent instruction—this would cause the process to restart from instruction `i5` in Figure 2-2(d).

2.2 Exception Types

This section classifies various asynchronous and synchronous exceptions that may be supported in a system. The information presented is derived from [HP07, Appendix A] and [MV96].

2.2.1 Asynchronous Exceptions

The following list includes several asynchronous exceptions.

- **Reset** is treated as an asynchronous exception in most architectures. It wipes out architectural state, and thus is not a restartable exception.

- A **hardware malfunction** can generate a machine-check interrupt. This type of fault may cause the termination of all running processes, although in some cases only affected processes may need to be terminated. The hardware state at the time of the interrupt may need to be preserved so that it can be later examined to determine

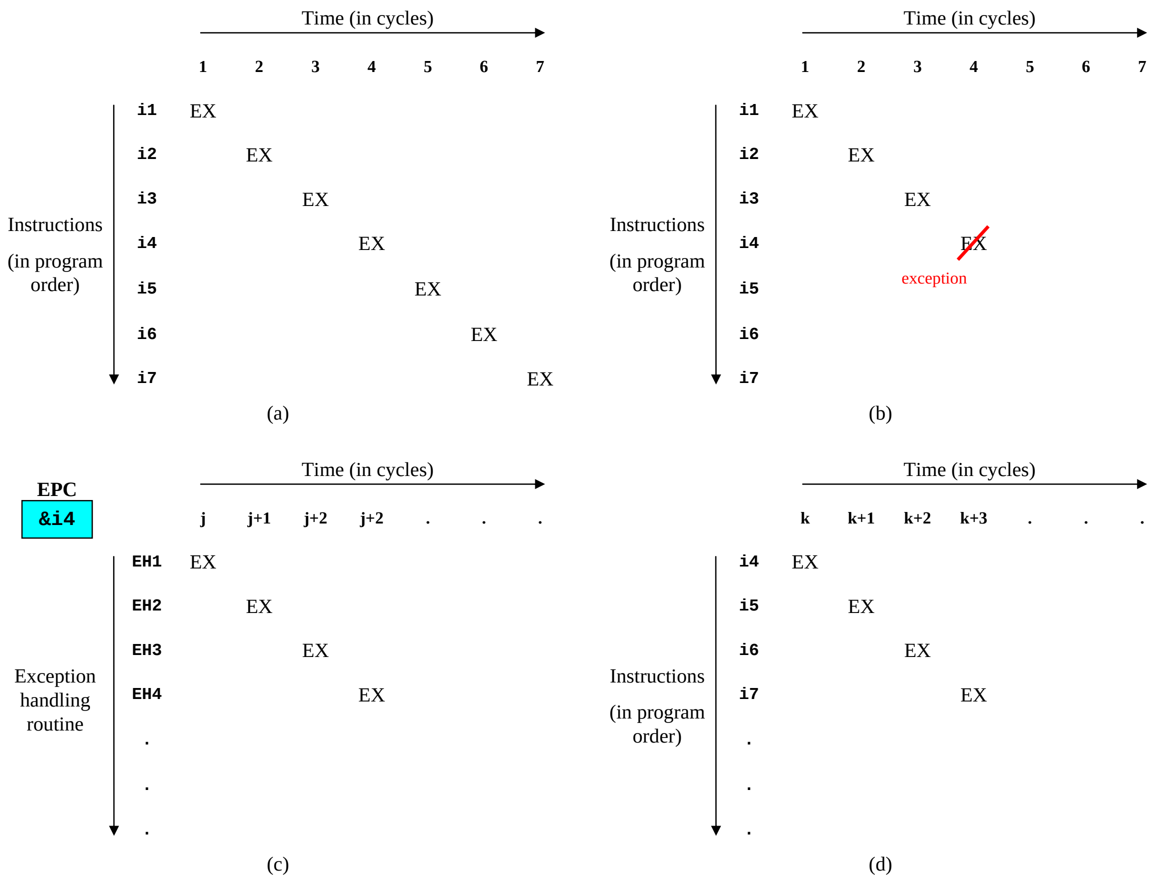


Figure 2-2: (a) Sequential and serial execution of an instruction stream with no exceptions. (b) Instruction `i4` causes an exception, causing the current process to stop execution. (c) After instruction `i4` causes an exception, its address is saved in the EPC register and the exception handler is executed. (d) Once the exception caused by instruction `i4` has been handled, the process resumes execution.

the cause of the error. The malfunction may be independent of program execution, in which case the exception is asynchronous. However, it is also possible for a synchronous malfunction to occur, so this exception is listed in both categories.

- A **timer interrupt** is an asynchronous event that is triggered when a process has executed for some predetermined amount of time. This functionality can be used to implement features such as multiprogramming, in which multiple processes time share the execution resources. When a particular process has used up its allotted time, the exception handler will perform a context switch to another process.

- An **I/O interrupt** is another type of asynchronous event. It occurs when an I/O device has finished executing some task, such as reading from disk. The exception handler may then select from a variety of different actions, such as copying data from the I/O buffers.

2.2.2 Synchronous Exceptions

The following list includes several synchronous exceptions.

- A **hardware malfunction** can be triggered by the program being executed, in which case it is a synchronous exception. The malfunction is dealt with in the manner described in the last section.

- An **illegal instruction exception** occurs when the processor does not recognize a particular opcode. This could be caused by the fact that the instruction is not defined in the architecture, in which case the process will terminate. Alternatively, the processor might not implement that particular instruction. In the latter situation, if the handler can emulate the unimplemented instruction, the process can then resume.

- A **misaligned memory access** may cause an exception depending on the type of system. The exception handler can emulate the misaligned access by using a combination of aligned accesses and bit manipulation techniques.

- A **system call** is invoked by a user-level process when it needs to request a service from the operating system. This can be used for actions such as accessing hardware devices.

- A **debugging exception** can occur when breakpoints or watchpoints are used. This type of trap halts the program and requires intervention by the programmer in order for the process to resume.

- An **arithmetic trap** may occur for a variety of reasons, including arithmetic overflow or underflow, a divide-by-zero error, or the need to handle floating-point special values such as denormals. Depending on the system and type of exception, the handler may be able to deal with the trap and resume execution of the process.

- A virtual memory [Den70] event such as a **page fault**, **memory protection violation**, or **software-managed TLB miss** will trigger an exception. Since it incurs such a significant performance overhead (typically ranging between 1,000,000 to 10,000,000 clock cycles depending on the system [HP07]), a page fault will also usually trigger a context switch to another process. By contrast, the TLB miss penalty is much smaller—e.g. on the order of tens of cycles [KS02a].

2.2.3 The Importance of Virtual Memory

As can be seen from the different types of exceptions, at least some basic level of exception support is needed in a processor, even if only to provide the capability to reset the system. Typically, events such as illegal instruction exceptions will be handled by the majority of processors (although there have been instances of processors with "halt and catch fire" illegal instructions that were not handled correctly), but support for other exceptions will vary depending on the target market. For example, many embedded processors are used to perform a single task, such as automotive control, and thus do not need to implement multiprogramming. However, with the advent of mobile computing and convergence devices, the line between embedded and general-purpose systems is being blurred, making it increasingly important for future processors to implement robust exception support. This thesis work is designed to accommodate a variety of exceptions. However, I particularly focus on demand-paged virtual memory support for three primary reasons. First, demand-paged virtual memory is essentially a requirement for general-purpose systems, and is likely to become a higher priority for future embedded systems. It provides a wide range of benefits, including protection between processes, shared memory support, the enabling of large address spaces, the enhancement of code portability, and support for multiple processes being active without having to be fully memory-resident [JM98b, JM98a]. Virtual memory exceptions can also be used to implement other features, such as garbage collection [AEL88], concurrent checkpointing [AL91], and pointer swizzling [WK92]. A second reason for focusing on virtual memory in this thesis is that virtual memory events are synchronous exceptions which require swappable exception support. Since synchronous exceptions are typically more difficult to implement than asynchronous exceptions—due to the fact that a synchronous exception must be correlated with a particular instruction—and since support for swappable exceptions implies support for restartable and terminating exceptions, a system that supports demand-paged virtual memory should be able to support most exceptions. A final reason for emphasizing virtual memory is that virtual memory events—particularly TLB misses in a system with software TLB refill—tend to occur far more frequently than other types of exceptions [RBH$^+$95] and thus typically have a greater performance impact.

Although I focus specifically on virtual memory support, it is worth noting that a demand paging mechanism can be used even in a system that does not implement virtual memory. For example, mobile embedded systems with on-chip SRAM for main memory and flash memory as secondary storage can use demand paging to reduce the amount of SRAM required [PLK+04, ISK07]. Even without virtual memory support, the use of demand paging in these systems necessitates a robust exception handling mechanism.

2.3 Precise Exception Model Definition and Implementations

This section defines the *precise exception model* and presents several mechanisms that have been used to implement precise exceptions.

For any exception handling mechanism, it is important to consider the simplicity of restarting the process after handling a restartable exception. In the example presented in Figure 2-2(b), determining the restart point is simple because execution is both sequential and serial: instructions execute one at a time in program order, so all instructions before the excepting instruction have completed, while none of the instructions after the excepting instruction have been started. Thus, the restart point after handling the exception is simply the last unexecuted instruction. As shown in Figure 2-2(c), the only hardware needed to determine the restart point is the EPC register.

When techniques such as pipelining and out-of-order execution enter the picture, things become more complicated, as there can be multiple instructions executing simultaneously, and instructions are not necessarily executed in order. If an instruction causes an exception, picking a single restart point may not be possible as the architectural state may not reflect sequential or serial execution. For example, consider the classic RISC pipeline described by Hennessy and Patterson [HP07, Appendix A]. Each instruction proceeds through five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write-Back (WB). Figure 2-3 shows a sample instruction sequence executing in that pipeline. Instruction `i4` causes an exception in the Execute stage during cycle 6. Suppose the process is immediately stopped so that all instructions are killed in cycle 7, as shown in the figure. Thus, instruction `i3` never proceeds through the Write-Back stage, which makes it difficult to restart the process (assuming the exception is restartable). For example, suppose `i3` is a store instruction. In this case, it has already updated memory and does not need to be re-executed. However, if `i3` is an add instruction, it has not yet written back its result, and so it does need to be executed again. Restarting the process from instruction `i3` can cause another problem—as mentioned previously, for certain types of exceptions, the excepting instruction should not be re-executed. If this is the case for instruction `i4`, then the system would need to execute `i3`, skip `i4`, and jump ahead to `i5` (assuming `i5` should be re-executed as well), which could lead to significant hardware design complications.

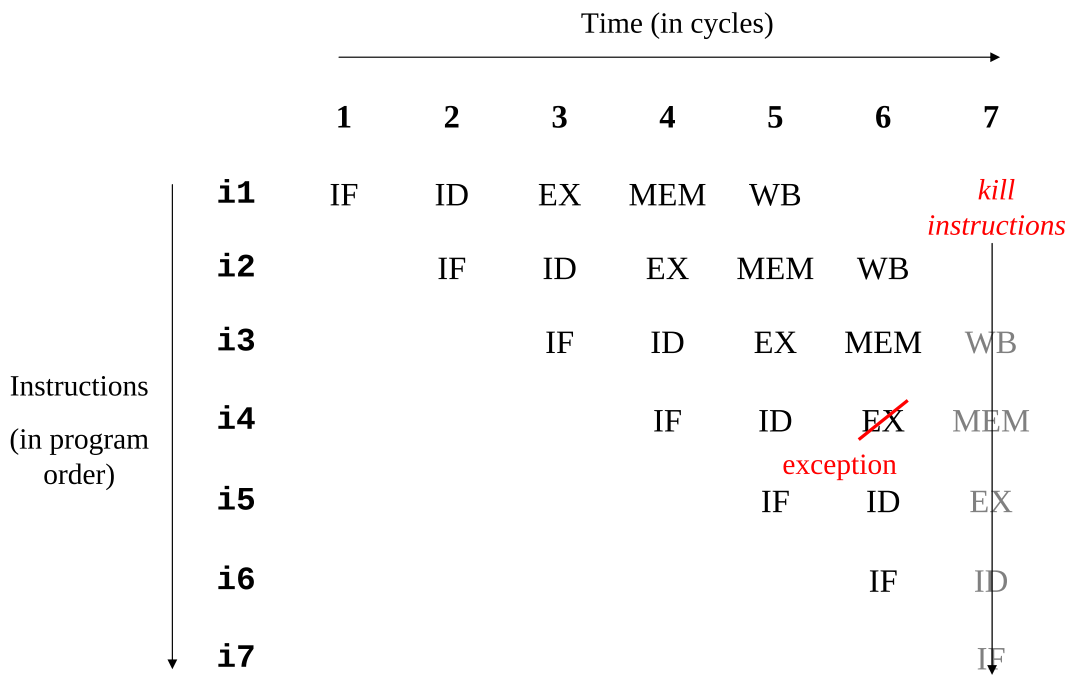


Figure 2-3: Instruction `i4` causes an exception in the Execute stage of a five-stage pipeline, and all of the instructions are killed in the following cycle, complicating the restart process.

The precise exception model avoids the above problems by giving the illusion of sequential and serial execution in processors that do not execute instructions sequentially or serially. In the following definition of the precise exception model, the instruction pointed to by the exception program counter when an exception is handled is referred to as the "EPC instruction". An exception is said to be precise if the architectural state reflects the following conditions [SP85]:

- All instructions preceding the EPC instruction in program order have completed execution and have correctly updated the architectural state.
- No instructions following the EPC instruction in program order have executed or modified the architectural state.
- If the exception is synchronous, the excepting instruction is the same as the EPC instruction. This instruction has either completed execution or is unexecuted.

The precise exception model requires instruction results to be committed in program order, or that the equivalent state will be recreated before the software exception handler begins running. From the perspective of the exception handler, the processor executes instructions sequentially and serially as shown in Figure 2-2(a). Thus, when restarting a process, picking a restart point is simple, and is only dependent upon whether the excepting instruction needs to be re-executed: if so, the process will resume execution from the excepting instruction; if not, the process will resume execution from the instruction after the excepting instruction. For the five-stage in-order pipeline shown in Figure 2-3, precise exceptions can be implemented by only handling an exception once the excepting instruction reaches the Write-Back stage (when all previous instructions have completed execution), and ensuring that architectural state (including memory) is not updated by any subsequent instructions. Typically processors will flush all instructions after the faulting instruction before handling the exception. Not only does this prevent any incorrect updates of architectural state, but it also avoids any problems that could occur from switching protection domains while still having active user-level instructions (in the case where the exception handler is kernel-level code).

Enforcing precise exceptions in the five-stage pipeline is simplified by the fact that instructions complete in program order. However, in processors that support out-of-order completion, exception handling becomes more difficult. Consider the pipeline for a superscalar processor shown in Figure 2-4, which contains multiple functional units. The functional units may have different latencies, which means that even if instructions are issued in order, they may complete out of order. So an instruction may cause an exception after subsequent instructions have already updated architectural state with their results. This can cause an additional problem if subsequent instructions overwrite the input values for the excepting instruction and the exception handler needs those values to correctly process the exception. One approach to deal with this issue is to add delay elements to the datapath so that each functional unit has the same latency. Although this increases the latency for a single instruction, the use of bypassing can potentially avoid a negative effect on overall instruction throughput. In effect, bypassing allows instruction results to be used before it is known whether instructions will complete without exception. However, inserting a bypass network does add overhead, particularly in deep pipelines that have a large number of values which need to be forwarded. Additionally, if instructions are issued out of order, or if multiple instructions are issued simultaneously, the problem of out-of-order completion arises again.

To support precise exceptions in out-of-order processors, designers typically include a dedicated commit stage, as illustrated in Figure 2-5. The idea behind this approach is that instructions are allowed to execute out-of-order, but their results will be committed to architectural state in program order, thus preserving the illusion of sequential and serial execution. Also shown in Figure 2-5 is a stage to perform register renaming so that performance will not be limited by false dependences [Tom67]. The Execute stage could contain multiple functional units as in Figure 2-4, but this is not shown for the sake of simplicity.

There have been a variety of methods used to implement precise exceptions within processors that allow instructions to complete out-of-order. The following sections illustrate the most popular techniques and briefly touch on alternative approaches. The diagram for each scheme is not meant to be an accurate representation of an actual microprocessor design showing all of the datapath details—for example, not all of the diagrams show issue queues—

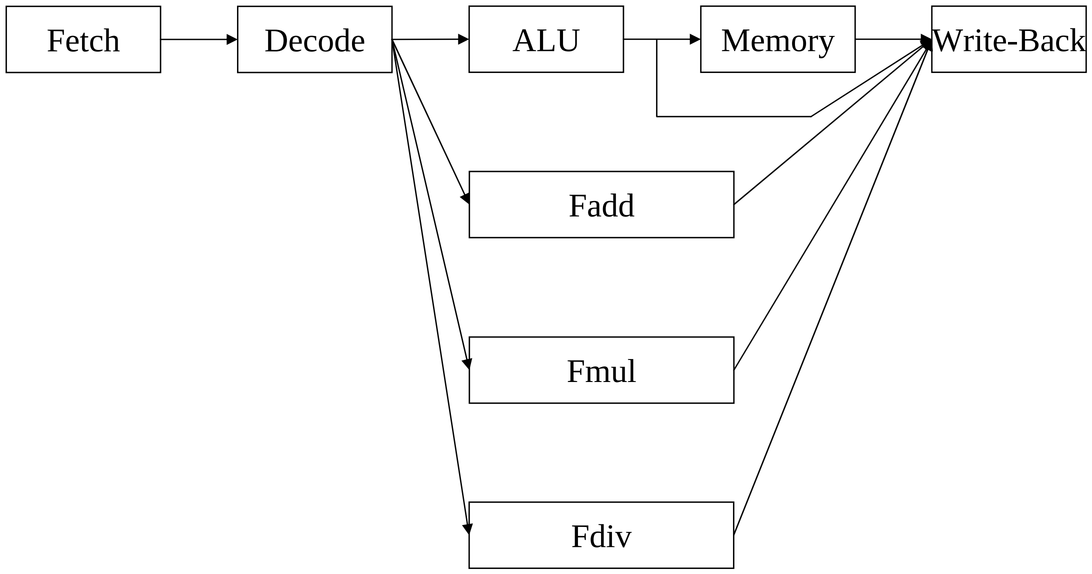


Figure 2-4: A superscalar pipeline in which multiple instructions might be in the Execute stage simultaneously.

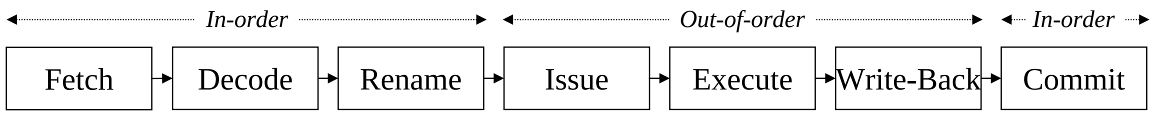


Figure 2-5: An out-of-order processor pipeline with a dedicated commit stage.

but rather a high-level generic abstraction that illustrates the key differences between each approach. In order to maintain some measure of consistency when comparing the different approaches, the following designs are somewhat modified from the way they appeared in the original works so that they correspond to the pipeline in Figure 2-5. For example, an explicit rename table is shown in each diagram. Additionally, certain changes made to the original schemes, such as Johnson's modification of the future file design [Joh91] have been incorporated into the figures. However, the basic ideas behind the original approaches remain the same, despite the pipeline changes—e.g. although Smith and Pleszkun [SP85] describe the reorder buffer, history buffer, and future file within the context of a superscalar processor with in-order issue, the same structures can be used in the out-of-order pipeline of Figure 2-5.

2.3.1 Reorder Buffer

The reorder buffer (ROB) [SP85] is the most prevalent mechanism used in today's superscalar processors to provide precise exception support. Figure 2-6 shows a high-level overview of how this design works. The fundamental idea is that the reorder buffer serves as a circular queue with head and tail pointers (not shown in the figure). Instructions enter the reorder buffer in program order at the tail before being executed. Each valid rename table entry maps an architectural register to the reorder buffer slot of the latest in-flight instruction that will write that register. The reorder buffer holds the source operands for each instruction, or a tag pointing to the entry of the instruction that will provide the missing operand. When instructions complete, they write their results back to the reorder buffer, and any instructions waiting on those values will have their source operands updated. Store values are written to the store queue. The reorder buffer then ensures that values are written to architectural state in program order by only allowing completed instructions at the head of the queue to commit. If an instruction is being committed and the rename table entry for its destination register points to the instruction's reorder buffer slot, then that rename table entry will be marked invalid to indicate that the register value should now be obtained from the architectural register file. Note that there is a path from the reorder buffer to the multiplexer which feeds source values to the reorder buffer; this is used to provide completed results that have not yet been committed. An exception may be detected at any point during execution, but it is only handled when an instruction reaches the head of the reorder buffer, as this ensures that all previous instructions have committed and no subsequent instructions have committed. Before handling an exception, the reorder buffer, rename table, and store queue will typically be flushed so that there will be no more active user-level instructions.

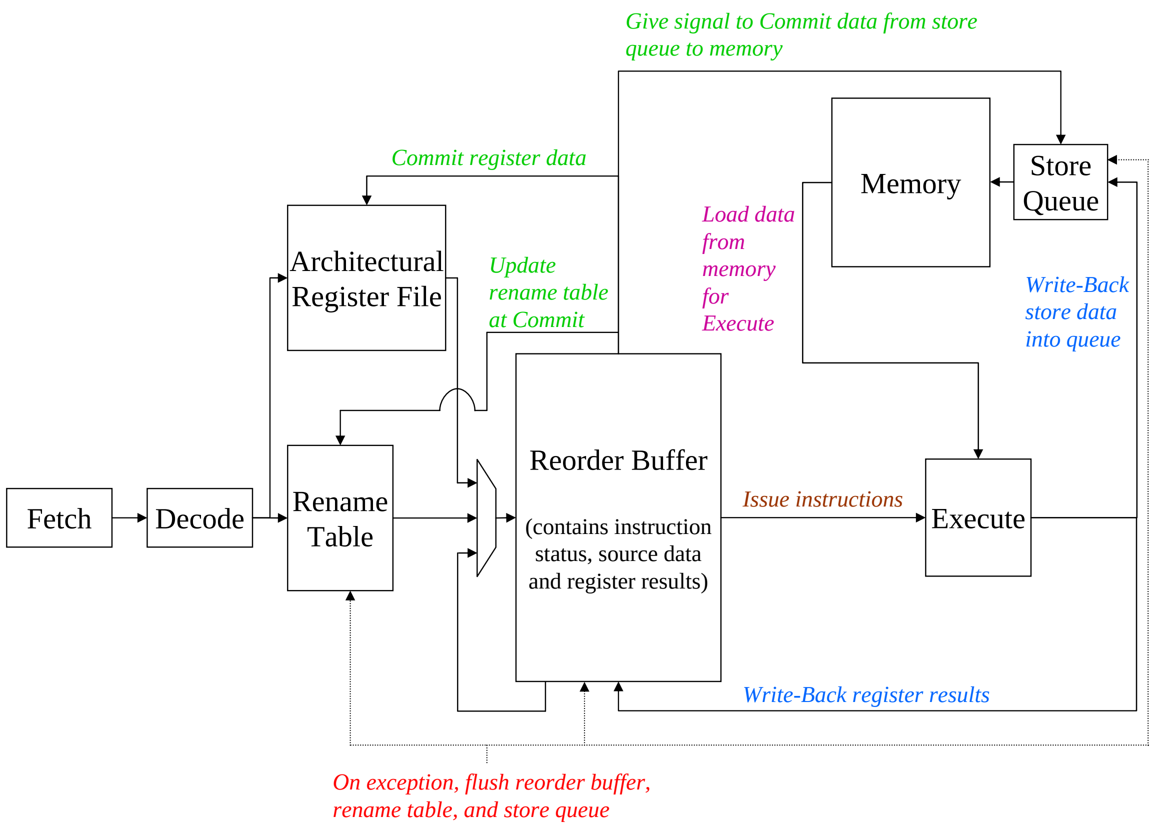

Give signal to Commit data from store queue to memory
Commit register data
Memory
Store Queue
Architectural Register File
Update rename table at Commit
Load data from memory for Execute
Write-Back store data into queue
Reorder Buffer
(contains instruction status, source data and register results)
Issue instructions
Fetch
Decode
Rename Table
Execute
Write-Back register results
On exception, flush reorder buffer, rename table, and store queue

Figure 2-6: A reorder buffer scheme.

2.3.2 History Buffer

The history buffer [SP85] is similar to the reorder buffer. However, while the latter structure ensures that the architectural state is always precise, a history buffer allows results to be written to architectural state out of order, but maintains enough information to recreate a precise state in the event of an exception. Figure 2-7 illustrates a history buffer design. When an instruction that targets a destination register is renamed, that register's current value will be written into the history buffer. The rename table entry for the destination register will be updated to point to the instruction's issue queue slot. The issue queue holds instructions that are waiting to execute as well as tags for source operands that have not yet been produced. When an instruction completes, it writes back its result to the architectural register file or to the store queue; additionally, its result is forwarded to any waiting instructions and the rename table is updated if necessary to indicate that the register value can now be obtained from the register file. When an instruction commits, the entry in the history buffer holding the old value that it overwrote is invalidated; committed stores have their values written from the store queue to memory. Again, exceptions are handled at commit time. If an exception occurs, the processor can step through the history buffer in reverse program order to restore all of the old register values.

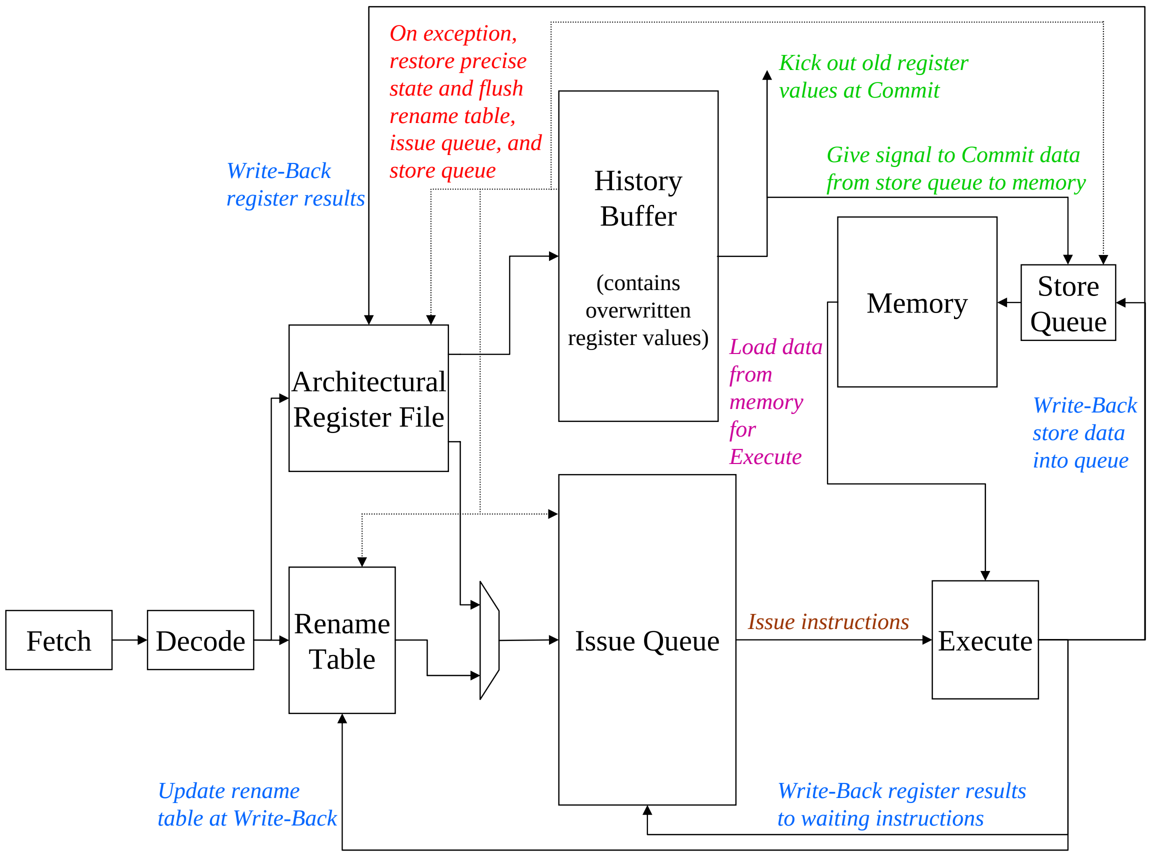

On exception, restore precise state and flush rename table, issue queue, and store queue
Kick out old register values at Commit
Give signal to Commit data from store queue to memory
Write-Back register results
History Buffer
(contains overwritten register values)
Memory
Store Queue
Architectural Register File
Load data from memory for Execute
Write-Back store data into queue
Fetch
Decode
Rename Table
Issue Queue
Issue instructions
Execute
Update rename table at Write-Back
Write-Back register results to waiting instructions

Figure 2-7: A history buffer scheme.

2.3.3 Future File

The future file scheme [SP85] is similar to the reorder buffer approach in that architectural state is updated in program order. Figure 2-8 shows a design incorporating a future file, which is a separate register file from the architectural register file. The design is very similar to Figure 2-6. The main difference is that if an instruction completes and it produces the latest update of a particular register, its result will be written to the future file. The purpose of this update is to remove the path present in Figure 2-6 from the reorder buffer to the multiplexer which feeds source values to the reorder buffer. In the future file design, when instructions are renamed, they no longer need to obtain any source values from the reorder buffer in this stage: the future file will provide values of instructions that have completed but not yet committed. In the event of an exception, the future file will be flushed. Apart from those changes, the future file design is identical to the reorder buffer design.

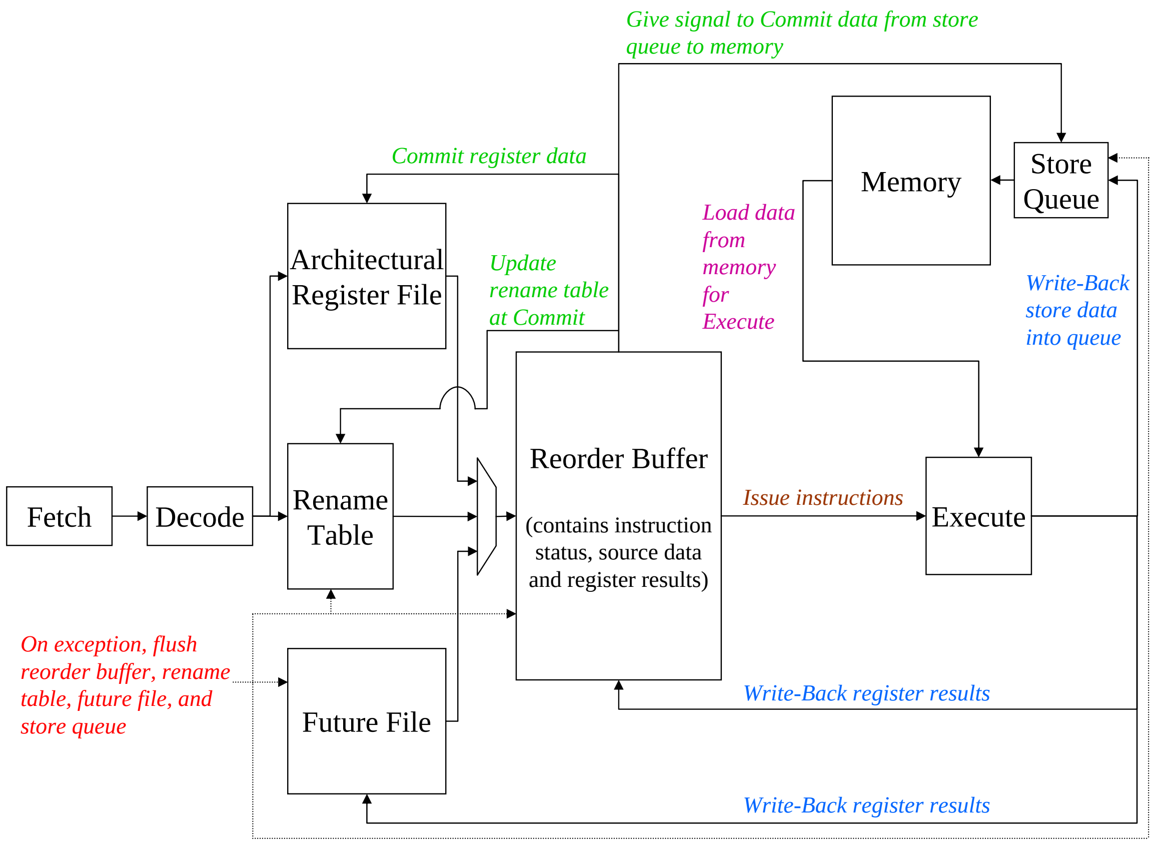

Give signal to Commit data from store queue to memory
Commit register data
Memory
Store Queue
Architectural Register File
Update rename table at Commit
Load data from memory for Execute
Write-Back store data into queue
Reorder Buffer
(contains instruction status, source data and register results)
Fetch
Decode
Rename Table
Issue instructions
Execute
On exception, flush reorder buffer, rename table, future file, and store queue
Future File
Write-Back register results
Write-Back register results

Figure 2-8: A future file scheme.

The design of Figure 2-8 only uses the future file to provide uncommitted values, which is based on Johnson's modification of the original future file design [Joh91]. In the original scheme described by Smith and Pleszkun [SP85], all source register operands are obtained from the future file, including values that have been committed to architectural state—i.e. the reorder buffer no longer holds any source operands. Since the future file acts as a working register file in this latter design, its values have to be restored from the architectural register file in the event of an exception, which can add performance overhead. However, for an in-order processor, using the future file to provide all source operands can be beneficial to avoid a complex bypass network. For example, Sun's UltraSPARC-III [HL99] is a superscalar processor that issues instructions in program order. It extends the integer pipeline to match the depth of the floating-point pipeline so that floating-point exceptions will be precise. While precise exception support is therefore straightforward and no reorder buffer is needed, avoiding a performance penalty is a significant concern since the UltraSPARC-III has a 14-stage pipeline, and implementing a full bypass network would significantly increase the complexity of the design. To avoid this problem, the UltraSPARC-III's integer pipeline uses a working and architectural register file (WARF) scheme—essentially a future file approach—to significantly reduce the bypassing complexity, enabling a short cycle time. In this case, the working register file, or future file, replaces the result bypass buses that would otherwise be used to prevent pipeline stalls.

2.3.4 Checkpoint Repair

Checkpoint repair [HP87] is somewhat similar to the history buffer idea of allowing instructions to update state and then recreating a precise state in the event of an exception. In this approach, shown in Figure 2-9, the processor makes backup copies of architectural state at certain points during program execution. Each instruction is associated with a particular register file, and it will write its result to that register file. The register file corresponding to the oldest checkpoint holds a precise state—all instructions preceding that checkpoint have committed. When all instructions between the oldest checkpoint and the second-oldest checkpoint have completed without exception, those instructions can be committed by freeing the register file for the oldest checkpoint and writing appropriate values from the store queue to memory. If an instruction causes an exception, the processor flushes the pipeline and restores a precise state from the oldest register file. It then executes instructions sequentially in program order until the exception is recreated. This guarantees that the process state will be precise when handling the exception.

2.3.5 Reorder Buffer With Unified Physical Register File

In the scheme proposed by Smith and Pleszkun [SP85], the reorder buffer holds instruction results and then commits them to architectural state. This general approach of having a dedicated architectural register file is also employed in the alternative schemes described above such as the history buffer design. An alternative scheme is to use a unified physical register file [MPV93] instead of having a separate structure to hold architectural state. This approach is illustrated in Figure 2-10. In this design, the reorder buffer no longer holds source operands or instruction results. Instead, it contains each instruction's operand mappings from architectural registers to physical registers. An instruction that is renamed

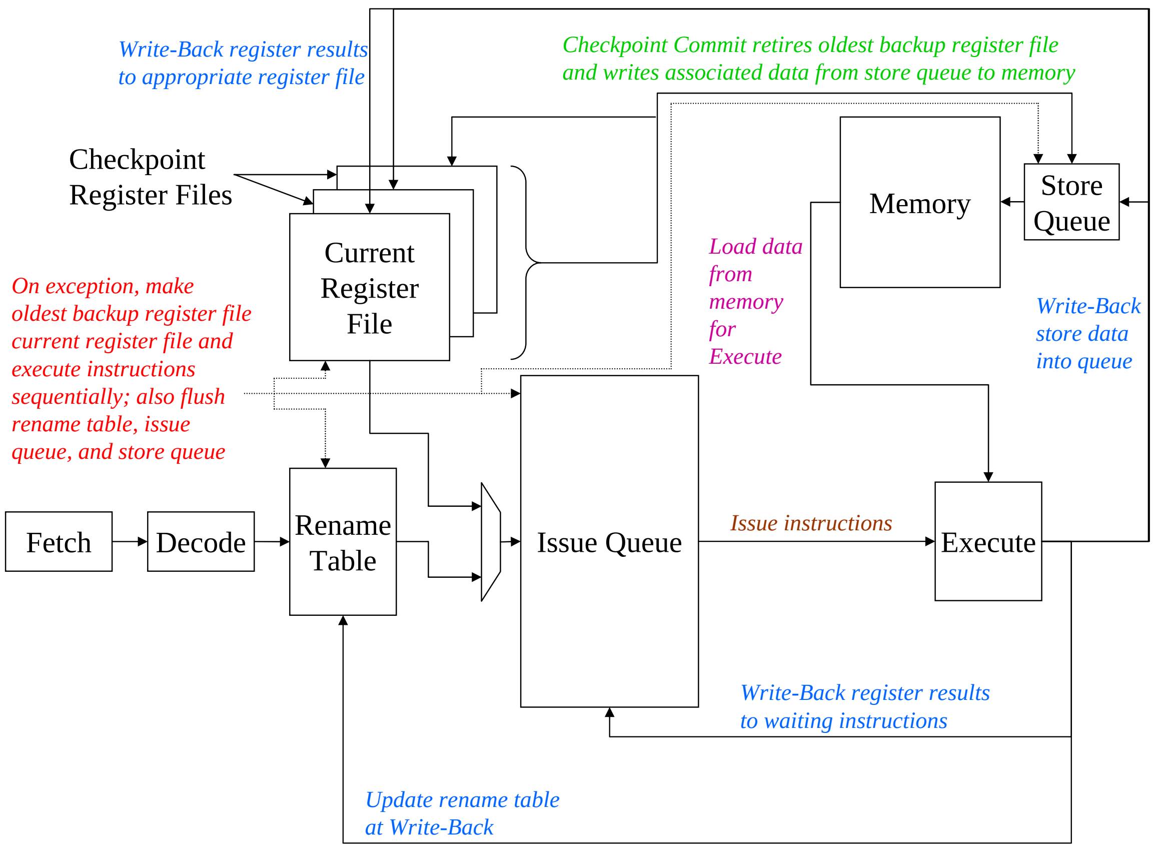


Figure 2-9: A checkpoint repair scheme.

will obtain the most recent mappings for its source operands, and will also update the mapping for its destination operand to point to a physical register from the free list. The old mapping of the destination register is copied into the reorder buffer as part of the instruction status. When an instruction is ready to execute, it uses its operand mappings to obtain its source data from the physical register file. When an instruction completes, it writes back its result to the physical register assigned during the rename stage; it also updates the reorder buffer to indicate it has finished execution. An instruction commits when it reaches the head of the reorder buffer; at this point, there are no longer any instructions in the pipeline that need the value pointed to by the old mapping for the instruction's destination register, so that physical register can be added to the free list. In the event of an exception—which is handled at commit time—the old mappings contained in the reorder buffer can be restored so that the rename table points to a precise architectural state. This design avoids the need to copy values from working registers to architectural registers. Also, the rename table now becomes part of architectural state, as the location of architectural register values can change throughout the duration of a program.

It should be noted that the description in [MPV93] uses a unified physical register file in conjunction with a history buffer, not a reorder buffer. However, the optimized scheme that the authors present is identical to the approach illustrated in Figure 2-10, which is described as using a reorder buffer. Both labels are appropriate in this context: although not shown in the figure, the buffer keeps a history of old physical register mappings, making it somewhat similar to the history buffer approach; however, it does not actually allow old values to be overwritten until commit time, making it somewhat similar to the reorder buffer approach. Given the more widespread usage of the term "reorder buffer" as opposed to "history buffer," the former label is used in Figure 2-10; the term "reorder buffer" is also used in the description of the Pentium 4's unified physical register file design [HSU$^+$01].

2.3.6 Other Approaches

There are several other proposals for handling exceptions precisely; in certain cases the designs may have different names but are actually similar or identical to the approaches already described in terms of exception handling. For example, Smith and Pleszkun's original reorder buffer scheme is described solely within the context of implementing precise exceptions in a superscalar processor with in-order issue. The register update unit [Soh90] extends the concept of the reorder buffer so that the same structure can also be used to support register renaming. The idea of using the reorder buffer for speculative execution is touched on in that work, but is more extensively addressed elsewhere, notably with the deferred-scheduling, register-renaming instruction shelf (DRIS) technique [PSS$^+$91], as well as the schedule table design [PM93]. Manohar, Nyström, and Martin [MNM01] use an approach similar to the reorder buffer scheme to enforce precise exceptions in an asynchronous processor. They insert a queue that causes instruction results to be written back to archi-

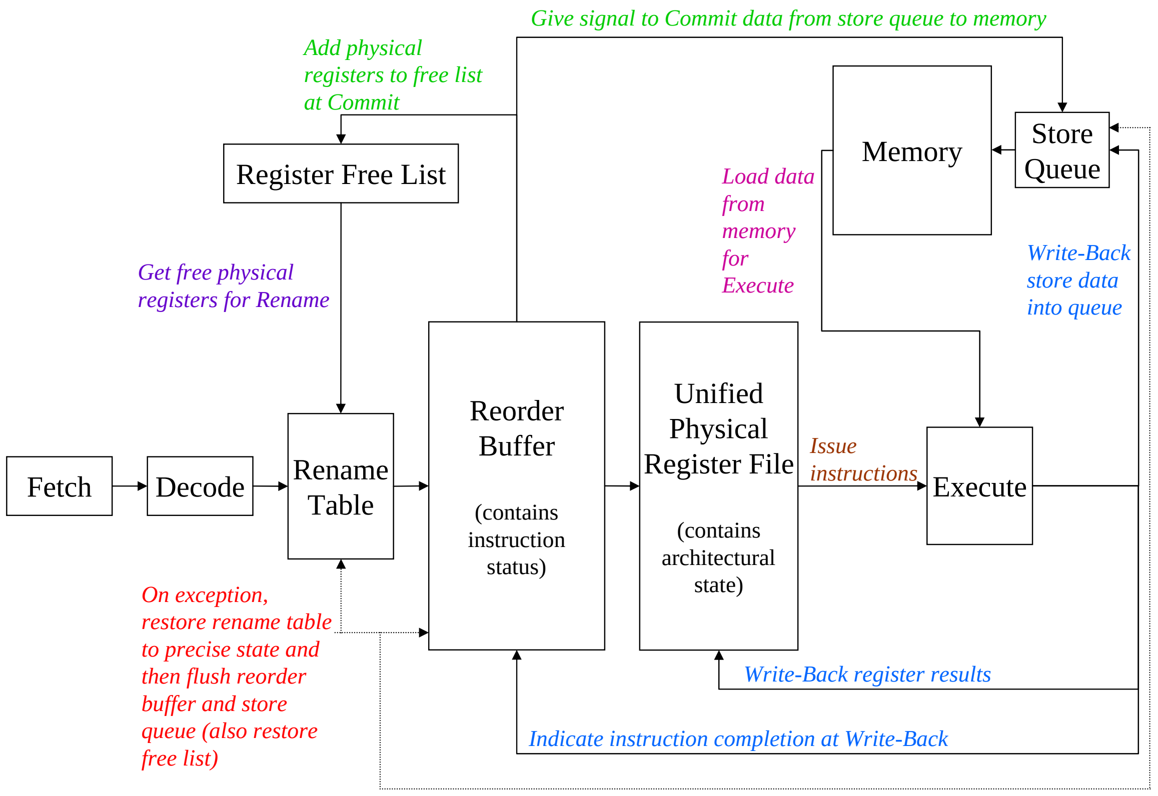

Give signal to Commit data from store queue to memory
Add physical registers to free list at Commit
Register Free List
Memory
Store Queue
Load data from memory for Execute
Get free physical registers for Rename
Write-Back store data into queue
Reorder Buffer
(contains instruction status)
Unified Physical Register File
(contains architectural state)
Issue instructions
Fetch
Decode
Rename Table
Execute
On exception, restore rename table to precise state and then flush reorder buffer and store queue (also restore free list)
Write-Back register results
Indicate instruction completion at Write-Back

Figure 2-10: A reorder buffer with a unified physical register file.

tectural state in program order. However, regardless of the name, the same basic approach is used in all of these cases to implement precise exceptions.

Smith and Pleszkun [SP85] also describe another technique in which a newly-issued instruction that takes i cycles to execute will prevent any subsequent instructions from issuing for i cycles by "reserving" pipeline stages. However, this approach can significantly degrade performance.

The WISQ architecture [PGH^{+}87] uses a reorder buffer to implement precise exceptions, but it presents an additional challenge because it makes internal pipeline details programmer-visible. The exposing of the pipeline provides multiple benefits: old register values can be used while new ones are being computed (due to the delay between issuing an instruction and the result being written back); results in the reorder buffer can be explicitly sourced; and instructions that have not yet committed can be selectively undone (as opposed to flushing all of the instructions past a certain point). However, the fact that the reorder buffer is programmer-visible means that it has to be saved in the event of an exception. When the process restarts, any previously-issued instructions that had not written back their results into the reorder buffer will be reissued. The authors argue that reorder buffer size (and thus the overhead of saving it) will be limited due to the fact that the number of entries only needs to equal the number of stages in the longest-latency functional unit, but they do not account for cache misses. Also, WISQ imposes a restriction that an instruction can only commit when a new instruction enters the reorder buffer. This is done to keep a constant buffer length, but it can potentially cause performance problems if a long-latency load causes the commit stage to get backed up.

The Fast Dispatch Stack (FDS) system [DT92] is similar to the future file scheme in that a second “working” register file is used in addition to the architectural register file. The key difference is that unlike the future file scheme, in which results to be committed are stored in the reorder buffer, the issue unit in the FDS system does not store any results. Instead, when an instruction commits, a signal is given that the result stored in the working register can be copied to its corresponding architectural register. To prevent the problems caused by the fact that multiple instructions may target the same register, an instruction cannot be issued if there is a previous instruction that has the same register destination; however, this approach can limit performance. The system also uses a copy-back cache in which each cache line has a duplicate. A store instruction that executes causes the line it updates to be marked as “Pending.” When the store actually commits, the line switches from “Pending” to “Current.” The authors point out that in their scheme they can issue and commit more than one instruction at a time and can execute stores out-of-order, as compared to the various designs described by Smith and Pleszkun [SP85]. However, it should be noted that the designs proposed by Smith and Pleszkun are not inherently limited to only single-issue or committing a single instruction at a time.

Hwang and Kyung [HK94] propose a content-addressable register file in which the architectural registers do not have fixed locations. Their scheme uses an “in-order buffer” to keep track of old register mappings that will be restored in the event of an exception. Overall, their approach is similar to the unified physical register file scheme with the primary exception being the content-addressable nature of their register file.

The FLIP processor [Hen94] takes advantage of existing branch mechanisms to handle exceptions while in user mode. An asynchronous exception is treated like a jump instruction which allows the processor to begin fetching exception handler instructions immediately without having to invalidate uncompleted instructions in the pipeline. A synchronous exception is handled in the same manner as a branch misprediction, causing all instructions after the excepting instruction to be flushed. Kernel-mode exceptions are handled by adding a mode bit to each instruction to indicate its protection domain. FLIP uses a template buffer that is similar to a reorder buffer which holds instruction results. However, it does not use a rename table, instead keeping register-tag bindings in the template buffer, and using a scan circuit to find the most recent tag for a particular register.

Alli and Bailey [AB04] propose a physical register file scheme in which an age tag is associated with each instruction. Each newly allocated physical destination register also stores the age tag of the instruction that will generate the result. Each rename table entry contains both the actual architectural-to-physical register mapping as well as the age tag of the last instruction to read the register. For each entry, a FIFO queue is maintained of the last `n` pairs of mappings and age tags. The design also contains a completion table that holds age tags and is used to commit the corresponding instructions in-order as they complete. When an instruction commits, the corresponding tag is broadcasted to the rename table so that mappings for physical registers which are no longer needed will be freed. In the event of an exception, the faulting instruction first signals the register file that it has excepted and broadcasts its age tag. If any registers have an associated tag that is less than the broadcasted tag—i.e. an older instruction is still uncompleted—the excepting instruction will wait for some number of cycles and re-broadcast its tag. Once all preceding instructions have completed, the excepting instruction then broadcasts its tag to the rename table so that all previous instructions will commit, releasing any unneeded architectural-to-physical register mappings.

Finally, Walker and Cragon [WC95] provide a comprehensive survey of exception handling mechanisms and describe many of the techniques discussed in this section as well as others such as shadow registers.

2.3.7 Precise Exceptions in Commercial Processors

To this point, I have described several research proposals for precise exception implementations. This section illustrates how various commercial processors have implemented the precise exception model. The purpose of this section is not to cover all processors that have ever been created, but rather to show that in general, industrial designs tend to use only a few basic approaches to implement precise exceptions. To limit the scope of this section, I focus on the major commercial architectures developed by companies that either are or were significant forces in the industry. Also, the emphasis of this section is on designs that allow out-of-order completion—as was previously mentioned, it is relatively straightforward to implement precise exceptions in processors with in-order completion. Finally, most of the processors in this section are for desktop systems, due to the fact that embedded processors often forego precise exception support, or alternatively because some embedded systems are simple in-order designs.

Intel's P6 microarchitecture [Gwe95]—which is employed in the Pentium Pro, Pentium II, and Pentium III processors—uses a reorder buffer that holds operation results and commits them to the retirement register file (RRF) in program order. The subsequent Netburst microarchitecture used in the Pentium 4 [HSU$^+$01] also uses a reorder buffer scheme, but with a unified physical register file.

Like Intel, AMD has been consistent in its precise exception implementations. The AMD K5 [Chr96] uses a reorder buffer that stores instruction results until commit time. Similarly, reorder buffers are used in the K6 [Hal96], K7 (Athlon) [Die98], and K8 (Opteron being the first implementation) [KMAC03] cores.

The IBM RISC System/6000 [OG90] handles floating-point exceptions imprecisely in the normal mode of execution, but also offers a precise mode in which only one floating-point instruction can execute at a time. Many of IBM's later processors use a *completion buffer* to ensure in-order instruction commit. A completion buffer is another name for a reorder buffer that controls when instruction results in "rename" registers can be committed to architectural state. Examples of processors that have a completion buffer include chips in the PowerPC series—which was jointly designed with Apple and Motorola—such as the PowerPC 603 [BAH$^+$94], 604 [SDC94], 620 [LTT95], and 740/750 (G3 "Arthur") [KAF$^+$97]. The POWER3 [OW00], POWER4 [TDJSF$^+$02], and POWER5 [SKT$^+$05] processors also incorporate completion buffers.

The Alpha AXP architecture specifies that virtual memory exceptions are handled precisely, but that arithmetic exceptions are imprecise by default [Sit93]. If the user requires precision for arithmetic exceptions, trap barrier instructions can be inserted. The use of trap barriers in the 21064 implementation of the architecture causes a performance degradation of between 3% to 25% in floating-point code, but this software-based approach also enables shorter cycle times [McL93]. The Alpha 21164 also handles virtual memory exceptions precisely and arithmetic exceptions imprecisely [ERPR95]. Both the 21064 and 21164 processors issue instructions in program order; by contrast, the Alpha 21264 supports out-of-order issue [Kes99]. To support precise exceptions, it employs a reorder buffer with a unified physical register file.

The Hewlett-Packard PA-8000 [Hun95] uses a reorder buffer to enforce in-order commit. When an instruction retires, its result is copied from its associated rename register to the appropriate architectural register.

The MIPS R8000 (TFP) [Hsu94]—which is the first superscalar implementation of the MIPS architecture—decouples the integer and floating-point pipelines. Thus, floating-point exceptions are imprecise with respect to integer instructions. The R8000 has a "precise-exception" mode which stalls the integer pipeline during the execution of a floating-point instruction. The MIPS R10000 [Yea96] uses a reorder buffer (referred to as an "active list") with a unified physical register file.

The SPARC architecture specifies a floating-point queue, which is essentially a reorder buffer that is visible to the exception handler. The HaL R1 CPU [PKL$^+$95] implements the SPARC version 9 architecture. It contains a Precise State Unit (PSU) that uses checkpointing to implement precise exceptions. Sun's UltraSPARC-I [TO96] and UltraSPARC-II [GT96] are in-order issue machines that enforce precise exceptions for long-latency floating-point instructions by artificially lengthening the integer pipeline so that it is the same length as the floating-point pipeline. By contrast, earlier SPARC processors like the Texas Instruments SuperSPARC [BK92] use the SPARC deferred floating point trap model in which there is a hardware floating-point queue that is visible to the exception handler. The use of the floating-point queue in these systems allows exceptions to be restartable. Sun's UltraSPARC-III [HL99] uses a future file scheme to implement precise exceptions for integer instructions. It also uses the approach from the UltraSPARC-I and Ultra-SPARC-II of lengthening the integer pipeline so that floating-point exceptions will be precise. Metaflow's Lightning [LH91] is a SPARC processor that uses a Dataflow Content-Addressable FIFO (DCAF) to implement the functionality of the Deferred-scheduling, Register-renaming Instruction Shelf (DRIS)—essentially a reorder buffer—described in [PSS$^+$91].

As a final example, the Motorola MC88110 uses a history buffer approach to implement precise exceptions [UH93].

2.4 Sources of Overhead in Precise Exception Implementations

As the previous section showed, over the last 15 years, commercial designs have used only a few different techniques to support precise exceptions, with the reorder buffer being the most widespread approach. While this has been acceptable until recently, the constant push to increase performance has exposed certain problems with existing precise exception implementations—namely, they introduce several sources of overhead. This section describes the different types of overhead associated with precise exception handling mechanisms, discusses various proposals to reduce that overhead, and shows that further research into the area of exception handling is warranted.

2.4.1 Overhead Unique to Precise Exceptions

One obvious source of overhead is the chip area required for the hardware structures used to support precise exceptions. Implementations of precise exceptions are typically hardware-intensive and make little use of software support [SG01]. All of the techniques that have been discussed rely on the use of buffers, whether that entails buffering new values before allowing them to be committed to architectural state, or buffering old values until they are no longer needed. For most of these implementations, a buffer is required for every result-producing instruction in the instruction window. The buffer could be a physical register, a history buffer entry, or some other form of storage. Even in the checkpointing approach, which does not have to buffer every register value, every store result has to be buffered. Buffers can also be found in a simple in-order pipeline in the form of pipeline latches between each stage that hold values for uncompleted instructions. As a result, the amount of buffering in all of these schemes is proportional to the size of the instruction window. As performance increases and the instruction window becomes larger, the number of buffers required will increase as well. Wang and Emnett [WE93] evaluate the designs proposed by Smith and Pleszkun [SP88] by generating VLSI implementations in a pipelined RISC processor. They conclude that the reorder buffer and history buffer schemes each introduce 15% area overhead, while the future file scheme introduces 46% area overhead. The actual sizes of the buffers used are not specified, but a size of 8 might be likely given the examples presented in the paper as well as the time period of this work. Although this evaluation is somewhat dated, it provides an indication that the area overhead of these approaches is not trivial. Also, a parallel architecture might require significantly more area to buffer instruction results—e.g. a single instruction in a vector architecture could produce on the order of 100 scalar results.

Adding large amounts of hardware also introduces overhead in the form of energy consumption. These structures consume energy in every clock cycle, not just when an exception occurs. The support needed for precise exceptions can lead to significant total energy costs; for example, in a Pentium III-style microarchitecture, 27% of the processor energy is dissipated in the reorder buffer [FG01]. This type of overhead conflicts with the push toward more energy-efficient computing.

Precise exception implementations also incur significant overhead in terms of performance. Aside from the fact that hardware structures such as a large physical register file can constrain cycle time, the fact that instructions are committed in order can lead to resources being tied up by completed but uncommitted instructions—e.g. there could be a long-latency load at the head of the instruction window that blocks newer instructions from committing. In Wang and Emnett's evaluation [WE93], the reorder buffer degrades performance by 20% versus an imprecise exception model, while the history buffer and future file degrade performance by 17%. Bell and Lipasti [BL04] also evaluate the overhead of in-order commit, and discover that permitting out-of-order commit in certain "safe" cases can improve performance by up to 68% for floating-point benchmarks, and by up to 8% for integer benchmarks. Performance (and energy) overhead also stems from the fact that precise exception implementations typically flush all subsequent instructions from the pipeline when handling an exception, even if those instructions have completed execution. This can lead to significant amounts of wasted work, as those instructions later have to be re-executed. Again, parallel architectures exacerbate these problems, as they typically have a larger number of in-flight instructions than conventional out-of-order superscalars.

Note that the above sources of performance overhead are independent of the time required to actually execute the exception handler instructions. In fact, the number of cycles required to actually deal with many exceptions is often insignificant compared to the actual program execution time, because most exceptions are designed to be "exceptional" events—i.e. they occur infrequently. There are a few exceptions whose handling can have a substantial impact on performance, notably TLB misses as well as swappable exceptions including page faults and timer interrupts. In general, TLB misses are by far the most frequent type of exception in a system with a software-managed TLB. For example, Rosenblum et al. [RBH$^+$95] execute program development, database, and engineering workloads on a 1994 MIPS R4400 processor-based machine model and present data showing that the frequency of TLB misses is more than an order of magnitude greater than the frequency of the second-most common type of exception, “VM faults” (which might include both page faults and memory protection violations). Other exceptions in this study such as system calls or interrupts are even more infrequent. As a more recent example, Caşcaval et al. [CDSW05] execute scientific applications on an Apple Xserve G5 system and show that on average a data TLB miss might occur as frequently as once every 1000 cycles or so, while there are typically many millions of cycles between page faults, even when using small pages. This TLB miss frequency combined with the actual time to handle the miss can significantly impact performance: Huck and Hays [HH93] state that database applications can incur 5–18% performance overhead from TLB management, with extreme cases incurring 40% overhead; Kandiraju and Sivasubramaniam [KS02a] show that one-fourth of the SPEC CPU2000 benchmarks have TLB miss rates higher than 1%, and with an “optimistic” 30-cycle miss penalty and a CPI of 1.0, this can result in 10% performance overhead; Peng et al. [PLW$^+$06] find that TLB misses can account for 24% to 50% of the total execution time when running Java applications on embedded processors. Depending on the type of system and the workload, the performance penalty of handling a software-managed TLB miss can be substantial. Swappable exceptions such as page faults may be less frequent, but they incur another form of overhead due to the fact that the process is swapped out, typically for millions of cycles. This can drastically increase the total execution time of a program. Although TLB misses and swappable exceptions can be costly, the performance overhead of actually handling most exceptions is often not that significant. This is particularly true in a system that handles TLB misses in hardware, or one that does not even support virtual memory. But if precise exceptions are supported, the other sources of overhead not related to handling the exception—such as area and energy consumption—*are incurred regardless of whether an exception occurs.* In a sense exception handling implementations can be viewed as “just in case” mechanisms. Although they are only infrequently required for an actual exception, they need to be available just in case an exception does need to be handled. And this can lead to significant costs with respect to other aspects of the system.

One caveat that should be mentioned at this point is that precise exception implementations such as the reorder buffer approach are frequently used to implement a variety of other performance-enhancing features in superscalar processors, such as register renaming to eliminate false dependences, branch misprediction recovery, and load speculation. Thus, depending on the particular design, it could be misleading to attribute all of these sources of overhead to the implementation of precise exceptions. However, as mentioned in the introduction to this thesis, there has been a shift away from using increasingly complex superscalar processors. Also, even in superscalar designs, the amount of overhead that is introduced by the structures used in precise exception implementations has recently begun to outweigh the benefits. The next section addresses various attempts to reduce that overhead.

2.4.2 Approaches to Reduce Precise Exception Management Overhead

A key issue that has arisen in recent years is the problem of scalability: designers want to support thousands of in-flight instructions in order to better tolerate memory access latency, but the typical superscalar designs which rely on the reorder buffer have hit a wall, only supporting on the order of 100 in-flight instructions. While there have been schemes devised to reduce the complexity of a reorder buffer that holds instruction results [KPEG04], there are still other issues to consider, such as large store queues. Akkary, Rajwar, and Srinivasan [ARS03b, ARS03a] argue that it is not actually the reorder buffer that limits scalability—since it is typically implemented as a FIFO—but instead the mechanisms that use the ROB to handle misprediction recovery and register file management. In their approach, called Checkpoint Processing and Recovery (CPR), they create checkpoints at low-confidence branches to allow for fast recovery with low overhead. They use a hierarchical store queue to avoid increasing cycle time. Also, instead of using in-order commit semantics to determine when to free a register, they reclaim a register as soon as all the readers have read the value and the logical register has been renamed to a different physical register. The follow-on Continual Flow Pipelines (CFP) work [SRA$^+$04] uses a non-blocking register file and scheduler to attain the performance of a large instruction window while still keeping the register file and scheduler small.

There have been many other proposals to tolerate memory access latency without introducing excessive overhead due to the various structures involved in implementing precise exceptions. Several of these techniques are based to some extent on the work of Dundas and Mudge [DM97], who introduce *runahead processing* as a means of pre-executing instructions when a cache miss occurs. Cristal et al. [CSC$^+$05] present a survey of much of the recent work, including the aforementioned CPR and CFP work, as well as Cherry [MRHP02], runahead execution for out-of-order processors [MSWP03], out-of-order commit processors [COLV04], and kilo-instruction processors [CSVM04]. As a recent example from industry, Sun uses hardware scouting [CCYT05] to improve single-thread performance in a multithreaded processor. When a performance-reducing event such as a load miss or a full store buffer occurs, the processor takes a checkpoint and then the hardware scout continues to process instructions in the instruction stream until the event is resolved. A shadow register file is used to avoid updating architectural state. When execution resumes from the checkpoint, additional data that will be needed later will have been prefetched by the scout; additionally, on-chip structures such as the branch predictor will have been warmed up.

One thing to note is that the extra energy consumption overhead due to these techniques is not measured. Also, these proposals rely on checkpointing: while they are suitable for sequential architectures, which typically have a small amount of architectural state, they would not be readily adaptable to systems with a large amount of architectural state such as vector machines because of the massive checkpoints that would be required.

As mentioned previously, another problem with precise exception implementations is the fact that they typically flush the pipeline before handling an exception. Keckler et al. [KCL$^+$99] propose *concurrent event handling* for multithreaded architectures to reduce the overhead associated with handling exceptions. In this approach, which is used in the MIT Multi-ALU Processor, the handler runs in a thread slot dedicated to dealing with exceptions. This design avoids the need to stop the faulting thread and flush its instructions, saving wasted work. Additionally, since the exception handler has its own register state, the faulting thread's registers do not have to be saved and restored. Finally, running the handler concurrently with the faulting thread can improve performance. A similar approach is proposed in [ZES99] for a multithreaded processor. Jaleel and Jacob [JJ06] develop a method for a single-threaded processor to allow the exception handler to run concurrently with user code. Their approach relies on Henry's technique [Hen94] of attaching a privilege bit to each instruction. Also, the hardware has to know the length of the exception handler to permit nonspeculative execution (there has to be enough space in the reorder buffer for the handler). Although these schemes can be advantageous, they do not provide any benefit for events which would cause a thread to be swapped out, such as page faults.

2.4.3 The Problem of Context Switch Overhead

The aforementioned proposals are designed to reduce the various sources of overhead attributed to precise exception support. As mentioned previously, in typical out-of-order superscalar designs, not all of the overhead can be solely attributed to precise exceptions—e.g. a large physical register file is also used for register renaming. However, there is an additional source of overhead that is unique to exception handling, one that is common to both precise and imprecise exception handling mechanisms: context switch overhead. When handling a swappable exception, a context switch is performed, so the process state needs to be saved to memory and later restored. One of the frequently-listed benefits of the precise exception model is that the only state which needs to be preserved across a swappable exception is the architectural state—there is no need to save any internal pipeline state. While it is certainly true that only saving and restoring the architectural state can keep context switch overheads manageable in a sequential architecture (which typically only has a relatively small number of programmer-visible registers), things can rapidly spiral out of control in an explicitly parallel architecture that has a much larger amount of state to be preserved. This section discusses two particular processor examples that illustrate this point, and also addresses the issue of why previous proposals to reduce context switch overhead have not proven completely satisfactory.

Processor Examples

The Cydra 5 minisupercomputer [BYA93] has a numeric processor that uses a VLIW architecture. The cost just to enter and exit the exception handler (not including actual time to run the handler) is about 2000 cycles, mainly due to the large amount of state that needs to be saved and restored. Rudd [Rud97] presents an analysis of the average per-operation performance cost that can be attributed to this exception handling overhead. Based on exception frequencies in [RBH$^+$95], Rudd concludes that the exception handling overhead is approximately 10 cycles per operation on the Cydra 5. This overhead is for each operation in the program, not just operations that cause exceptions. Although the typical usage of the Cydra 5 rarely results in context switches—as it is a supercomputer with a large main memory designed to make demand paging infrequent—Rudd points out that aggressive processors in other processor domains can have similarly high context switch overheads. It is also worth noting that Rudd's analysis apparently understates the actual frequency of exceptions per operation by factors ranging from 4.1× to 8.5×, depending on the workload. Table 4.2 in [RBH$^+$95], from which Rudd derives his data, lists exception rates using the metric of events per second; however, Rudd's analysis treats the rates as the total number of events for the entire duration of the program. Using the correct data substantially increases the exception handling overhead per operation, which only serves to further highlight the problem of context switching.

As a more recent example to illustrate Rudd's point, the Cell processor [Hof05] is designed to avoid many of the sources of overhead found in modern superscalar processors, such as structures to support out-of-order execution, hardware branch prediction, and deep pipelines. It is a multicore processor that contains eight Synergistic Processing Elements (SPEs) which work in conjuction with a Power Architecture core. Cell is able to exploit data-level, instruction-level, thread-level, memory-level, and compute-transfer parallelism [Gsc07]. To enable a power-efficient design, it relies heavily on large architectural register files and statically-scheduled parallelism as opposed to the register renaming and dynamic extraction of parallelism found in superscalar processors. However, this approach comes at a significant cost in the form of context switch overhead. Saving the context of a single SPE—which includes a local store that resides between the processing element and memory—requires over 258 KB of storage space [IBM07], and this leads to a substantial performance overhead when swapping out a context. (For reference, a typical sequential architecture might have a fraction of a kilobyte of architectural register state.) As a result, the recommended programming model for Cell is one that is "serially reusable"—i.e. an SPE executes a task to completion, and then another task is assigned to it in a serial manner. However, avoiding the use of multiprogramming in the SPEs can lead to decreased resource utilization if there are frequent stalls due to waiting for data from memory.

Proposals to Reduce Context Switch Overhead

Snyder, Whalley, and Baker [SWB95] describe a method in which the compiler inserts a bit for each instruction to indicate whether it is a fast context switch point, which is defined as a point at which no caller-saved registers ("scratch" registers) are live. (Unless interprocedural analysis is used, the compiler cannot determine whether callee-saved registers are live.) This means that only callee-saved registers ("non-scratch" registers) have to be saved when performing a context switch. If an exception occurs and a processor does not reach a fast context switch point within a predetermined amount of time, it will simply save all registers. A register remapping technique can also be used to remap scratch registers to non-scratch registers when possible, potentially increasing the number of fast context switch points. While the approach described in this work can be useful for asynchronous exceptions such as timer interrupts, it does not typically improve context switch overhead for synchronous exceptions such as page faults, unless the faulting instruction is also a fast context switch point.

A similar technique is proposed by Zhou and Petrov [ZP06], who pick "switch points" (typically loop boundaries) where there are very few live registers and create special register/save restore routines for these points. If an exception occurs, execution will continue to the switch point and then the exception will be handled. As with the above proposal, this approach only works for asynchronous exceptions.

Saulsbury and Rice [SR01] use dirty bit registers to indicate whether a working register (or group of working registers) is live in the event of a context switch. If only a few registers are live, the time to save and restore the values can be significantly reduced. One drawback of this approach is that the programmer has to insert an `mnop` instruction to indicate a dead value, which can lead to bloated code. While there are alternative methods to mark whether values are live or dead—such as a dirty bit mask at the boundary of a loop or function—another problem is that the exception handling routine has to check each dirty bit individually. If there are only a few live values, there will probably be a net benefit to this technique even with the dirty bit checks due to the avoidance of load and store operations. However, if most of the values are live, then not only will the majority of registers have to be saved and restored, but there will also be additional overhead from having to check the dirty bit of each register. When dealing with a large number of registers, this overhead could be substantial.

An idea that is not specifically targeted toward reducing context switch overhead for swappable exceptions but is similar in spirit involves support for user-mode exception handlers. Although exceptions are typically handled in kernel mode (and that is the approach used throughout this thesis), certain exceptions could be handled at the user level. This can improve exception handling efficiency in systems that use a general-purpose handler which does not differentiate between restartable and swappable exceptions—i.e. it saves and restores process state even when it is not required, incurring the overhead of a context switch although the process is not swapped out. Thekkath and Levy [TL94] show how user-mode exception handling can avoid this unnecessary saving and restoring of state and also reduce the number of crossings between protection domain boundaries. They demonstrate the use of their scheme with memory protection violations and misaligned accesses, and show how it can be used in applications such as distributed shared memory, unbounded data structures, and pointer swizzling. Similarly, Appel and Li [AL91] discuss several algorithms that are used in systems which allow user-mode handling of memory protection violations. Applications that they cover include concurrent and generational garbage collection, shared virtual memory, concurrent checkpointing, and persistent stores. Although these schemes can avoid saving and restoring registers for certain exceptions and have a variety of applications, they do not address the issue of swappable exceptions.

2.5 Imprecise Exception Handling Mechanisms

Given the sources of overhead associated with precise exception implementations, it is worth considering the ramifications of handling exceptions imprecisely. This notion dates back to at least the time of the IBM System/360 Model 91 [AST67], which allowed an "imprecise interrupt" in certain situations in order to avoid a significant performance reduction. Section 2.3.7 listed additional processors that have an "imprecise" mode for performance reasons—e.g. when handling floating-point exceptions. However, there have been relatively few approaches to handling exceptions imprecisely while still supporting swappable exceptions such as page faults. This section discusses the primary techniques.

Smith [Smi98] describes the approach used by the CDC STAR-100 and Cyber 200 to implement virtual memory, which is the same technique used by the DEC Vector VAX [BSH91]. When an exception occurs, those processors save the entire pipeline contents in an "invisible exchange package"—a technique I refer to as *microarchitectural swapping*. However, there are certain problems with this approach, as brought out by Rudd (who refers to this as the Snapshot approach) [Rud97] as well as Moudgill and Vassiliadis [MV96]. One drawback is that the amount of state to be saved can easily become unmanageable, particularly in complex pipelines. Also, a substantial amount of logic and wiring is required so that every pipeline latch can be saved and restored; besides the area and complexity that is involved, this requirement can also lead to an increase in the processor's cycle time. Another problem is that a global signal is needed to freeze the state of the processor in the event of an exception; delivering that global signal across a large processor could prove problematic. Finally, the operating system's view of the machine is made more complicated by the need to preserve internal pipeline state.

The instruction window scheme [TD93] is another alternative to precise exceptions. The actual instruction window structure has entries that each contain a tag, the instruction, and whether the instruction has been issued. If an exception occurs, only instructions that have reached a "no return point" (which can be set to either the final machine cycle of an instruction for fast exception handling, or the start of instruction execution to allow all executing instructions to complete, or somewhere in between) can complete and update state. The processor then handles the exception and saves the contents of the instruction window, providing a "modified precise" exception model. (This approach is also referred to as "functionally precise" by Richardson and Brunvand [RB95], who use it to handle exceptions in a self-timed, decoupled processor that supports out-of-order completion. Moudgill and Vassiliadis generalize this scheme and refer to it as "sparse restart" [MV96].) When the process resumes, the instruction window and register state are restored, and execution can continue where it left off. Thus, only uncompleted instructions will be executed after the exception; although not specified in the paper, this scheme requires each instruction's source operands to be stored in the instruction window in case a later instruction overwrites a required value and then an exception occurs. Also, only instructions at the top of the instruction window are considered for issue, because the window does not store any instruction results: the architectural register file is updated as instructions complete. While the instruction window approach localizes the state that has to be saved (versus saving the entire pipeline contents), it can still incur significant overhead when dealing with a large number of in-flight instructions.

Rudd [Rud97] proposes the use of *replay buffers* to handle exceptions in architectures with exposed pipelines, such as VLIW processors. The results of in-flight instructions can be drained into the replay buffers in the event of an exception, and then later replayed into the pipeline after handling the exception to preserve the original order of instruction execution. However, besides the fact that this approach requires enough buffer space to hold each in-flight result, it also does nothing to reduce context switch overhead. In fact, context switch overhead is increased since the replay buffer has to be saved and restored when handling a swappable exception.

Smith and Pleszkun [SP88] argue that instead of trying to make a parallel implementation conform to a sequential architectural model by supporting precise exceptions, a better approach might be to have the exception handler assume that the implementation is parallel. However, their proposals for future research still incur significant hardware overhead. They mention the possibility of saving all the pipeline state (as in the CDC STAR-100 and Cyber 200), but recognize the problems of a global signal and an implementation-dependent exception handler. They suggest a variant of this approach for vector architectures in which the intermediate state of vector instructions is saved (using length counters). The IBM System/370 vector facility [Buc86] uses this length counter technique to track how many elements of a vector have been processed; after handling an exception, execution will be resumed from the point indicated by the counter. However, the IBM design only executes one vector instruction at a time. The length counter technique can become unwieldy if a large number of in-flight vector instructions are permitted, particularly since each counter has to be saved in the event of a swappable exception. Another solution proposed by Smith and Pleszkun is saving a series of program counter values, each one pointing to an uncompleted instruction that has to be executed after handling an exception. They also mention a variant in which the actual uncompleted instructions are saved, rather than just their program counter values. These approaches are similar to the instruction window approach mentioned earlier and have the same scalability issues.

2.6 Classification of Exception Handling Mechanisms

Section 2.4 illustrated the various sources of overhead associated with precise exception implementations and provided the motivation for research into precise exception alternatives. As shown in Section 2.5, existing imprecise exception handling mechanisms have certain disadvantages. These drawbacks have prevented these proposals from becoming widely adopted. In fact, the imprecise techniques listed earlier have frequently been ignored due to their complexity or simply overlooked, as evidenced by the fact that there have been some incorrect assertions regarding exceptions in certain computer architecture publications, even in ones that are established references in the field. For example, although the approaches described in Section 2.5 can be used to implement demand-paged virtual memory, Hennessy and Patterson state that demand paging requires precise exception support [HP07, Appendix A]. Fisher, Faraboschi, and Young make a similar claim and use that as part of the justification for their argument that future embedded processor designs should support precise exceptions [FFY05]. These statements might be attributable to the fact that the vast majority of systems which support demand-paged virtual memory also implement precise exceptions. However, part of the problem may also be due to the traditional view of exception handling mechanisms as being either precise or imprecise, which tends to obscure whether a particular feature actually requires precise exception handling. To address this issue, the following two sections of this thesis are designed to more systematically illustrate what is needed to support various exceptions. In this section, I present a new classification of exception handling mechanisms that goes beyond simply labeling an approach as either "precise" or "imprecise."

The first axis along which an exception handling mechanism can be classified is its *level of restartability*, which is defined as its support for terminating, restartable, and swappable exceptions. Since a terminating exception only requires the ability to kill the process, any basic mechanism supports terminating exceptions. A mechanism that only supports terminating exceptions has the lowest level of restartability, as it never resumes the process after an exception. The next level involves support for restartable exceptions as well as terminating exceptions: this requires a sufficient interface to the handler so that the process can be restarted after the exception is handled. A mechanism at the highest level of restartability supports swappable exceptions, which also implies support for restartable and terminating exceptions.

The second axis of classification for exception handling mechanisms is the *condition of the process state* when handling an exception. To simplify the comparison to precise exceptions, the possible values for this category are defined according to whether the mechanism conforms to the expected behavior of a sequential architecture. One possibility is that the condition of the process state reflects what would be expected from a sequential architecture—i.e. it is identical to the state that would result in a processor which executes instructions both sequentially and serially. A second possibility is that the process state is identical to what would be found in a system that supports sequential execution but not serial execution—i.e. instructions are executed one at a time but not in order. For example, the instruction window approach [TD93], in which instructions can commit out of order, would fall into this category. A third possibility is that the process state reflects serial execution but not sequential execution. An example of this would be a VLIW architecture that executes instructions in order, but handles exceptions at the granularity of an entire instruction rather than an individual operation. The final possibility is that the process state does not reflect either sequential or serial execution.

An exception handling mechanism can also be classified in terms of its *exception reporting order*. Exceptions can either be reported in program order or out of program order. Note that this ordering applies to synchronous exceptions. Because the timing of asynchronous events is independent of the actual program execution, ordering constraints do not apply in those situations.

The fourth axis of classification specifies the *inputs to the exception handler*. When supporting restartable or swappable exceptions, the handler requires at the very least the address of the faulting instruction, which is stored in the exception program counter. If the handler is provided with the EPC, it can deal with the exception; however, in order for the latency of the handler to be reasonable, the specific cause of the exception should also be supplied as an input. For example, when a page fault occurs in the Intel i860, the exception handler is only provided with the address of the faulting instruction and the information that some kind of memory fault has occurred. As a result, the processor must execute an extra 26 instructions to interpret the faulting instruction in order to determine the specific cause of the exception [ALBL91]. Typically, the combination of the EPC and the exception cause form the optimal set of inputs to the exception handler in terms of performance. Additional inputs are typically provided only when the handler has to do additional work, such as in the situation where some instructions before the faulting instruction have not yet completed execution.

The final axis of classification of an exception handling mechanism is the *amount of overhead* required to handle the exception. This category encompasses the various sources of overhead discussed in Section 2.4. Precise exceptions are attractive in many cases as there is little work needed by software to deal with the exception state. The standard architectural state can be saved and restored using conventional instructions, and execution can then resume with a jump back to the exception PC.

2.7 Exception Needs

By definition, an exception handling mechanism that supports precise exceptions at the granularity of an operation ensures that the process state reflects sequential and serial execution when the handler is invoked, and also that exceptions are reported in program order. (Parallel architectures such as vector or VLIW with instruction-level precision rather than operation-level precision do not require the process state to reflect sequential execution.) An advantage of this approach is that providing the exception handler with the EPC and the exception cause is sufficient for it to deal with the exception. However, as discussed earlier, this approach can lead to implementation challenges due to the need to buffer results to enforce in-order commit. These issues make it worthwhile to consider which types of exceptions actually require precise exception support. (It should also be noted that although it is not explicitly required by the definition, precise exception implementations typically handle swappable exceptions.) This section presents a general overview of the characteristics that various exceptions require to be true of an exception handling mechanism, although it does not go too deeply into the issue because certain requirements may vary depending on the overall system.

Asynchronous events such as timer or I/O interrupts may require support for swappable exceptions, as these events might trigger a context switch. However, in general they can be taken at any convenient time after being detected, and thus do not need to be reported in order. Additionally, the process state does not need to reflect sequential or serial execution when handling these types of exceptions, as long as the process can later be resumed.

Virtual memory events such as software-managed TLB misses or page faults do not need the process state to reflect sequential or serial execution, or for exceptions to be reported in order. This is because these events affect programmer-invisible state such as the page table, and thus the precision of the architectural state is unimportant. However, it is desirable—but not required—to have a simple interface to the exception handler because of the frequency of TLB misses. Since TLB misses are frequent relative to other exceptions, it is also important that the handler overhead be low. Additionally, a system with demand paging requires support for swappable exceptions, as otherwise the large number of clock cycles required to bring in a page from the disk will cause the processor to be idle for an unacceptably long period of time. Since swapping out a process in the event of a page fault is designed to improve processor resource utilization, it is also important to have low context switch overhead, or else the purpose of performing the swap will be defeated.

Arithmetic traps typically only require support for restartable exceptions, as the traps can often be handled within a short bounded period of time. Also, it is frequently desirable for the process state to reflect sequential and serial execution when handling an arithmetic trap, and for exceptions to be reported in order, as this simplifies the handling of events such as arithmetic overflow. The alternative would be to implement a structure such as the instruction window which preserves the inputs of the faulting instruction for the handler to examine and also keeps track of uncompleted instructions.

System calls may need full precise exception support, depending on what the particular operating system call needs to do. Finally, debugging events such as breakpoints or watchpoints also require precise exceptions so that the programmer can accurately examine the process state.

In general, precise exception support is only occasionally required for certain events. However, the hardware overhead associated with supporting precise exceptions affects the execution of all programs, regardless of the types of exceptions that actually occur.

2.8 Developing a New Exception Model

The previous section illustrates the fact that a precise exception model is not necessary to support many types of exceptions. Given the numerous sources of overhead associated with existing precise exception implementations—particularly in parallel architectures—it is worth exploring alternatives to the precise exception model. In order for a new model to achieve widespread usage, it should have certain characteristics:

- **The model should facilitate support for swappable exceptions, retain a simple interface to the exception handler, and reduce or eliminate the various sources of overhead associated with precise exceptions.** Swappable exception support is required to enable features such as multiprogramming and demand-paged virtual memory, which are necessary in any system that is designed to gain mainstream acceptance. A handler interface that retains the simplicity of precise exceptions is such an important characteristic that it could almost be treated as a requirement for any new model. Finally, unless the new model can improve on the disadvantages of precise exceptions by reducing the overhead discussed earlier, there will be little or no benefit to using it.

- **Implementing the model should not require unreasonable tradeoffs.** Alternatives to established approaches typically come at a cost, so a new model that reduces the overhead of precise exceptions while retaining many of its benefits will likely have some other disadvantages. Although the overall importance of these disadvantages will vary from system to system, there should be at least some designs in which these tradeoffs do not outweigh the advantages of the new exception model.

- **The new model should be applicable to a wide range of architectural paradigms.** Introducing a model that is targeted toward a specific design will limit its usefulness. Given the recent trend toward parallel computing, it would be especially appropriate if the new model could be used in different types of parallel architectures.

This thesis describes an exception model that satisfies the above constraints. It is not designed to completely replace precise exceptions, but rather to provide an acceptable alternative with reduced exception management overhead that can be used in a variety of situations.

2.9 Summary

This chapter presented an overview of exceptions and existing exception handling mechanisms. Exceptions are used to enable a variety of features, including multiprogramming and demand-paged virtual memory. Robust exception support is a requirement for any general-purpose system, and is likely to become increasingly important for future embedded designs. A key issue in dealing with exceptions is being able to restart the process after the exception is handled. In a processor that executes instructions both sequentially and serially, determining the restart point is simple. The precise exception model replicates that simplicity in other types of processors by ensuring that instructions commit in program order. While there have been various schemes proposed to implement precise exceptions, only a few have been used in commercial processors, with the reorder buffer being the primary approach. However, existing precise exception implementations introduce significant amounts of overhead with respect to chip area, energy consumption, and performance. Although the structures that enable precise exceptions in out-of-order superscalar processors are often used for other features such as register renaming, an additional source of overhead unique to exception handling is the time required to perform a context switch when dealing with a swappable exception. This is an issue for both precise and imprecise exception handling mechanisms. Given the various problems with precise exception implementations—and the fact that existing imprecise exception handling approaches have certain disadvantages which have prevented their widespread adoption—it is worth exploring alternatives to the precise exception model. This thesis has presented a new classification of exception handling mechanisms that avoids the typical approach of viewing techniques as either precise or imprecise. Additionally, an overview of what different exceptions require from a mechanism has been discussed, showing that certain key features such as demand-paged virtual memory do not actually require precise exception support. Finally, the characteristics that should be found in a new exception model were listed.

Chapter 3

Software Restart Markers

In this chapter, I present an alternative to the precise exception model for handling exceptions that is based on the usage of *software restart markers*. This approach relies on compile-time information to coarsen the granularity at which instructions are committed. Software restart markers have been presented in previous work within the context of *energy-exposed architectures* [Ham01, AHKW02]. However, that work focused on an exception handling approach designed to reduce energy consumption within a simple five-stage RISC pipeline for a sequential architecture. The focus of this thesis is on using software restart markers to reduce multiple sources of exception management overhead within parallel architectures while still supporting features such as demand-paged virtual memory. Additionally, some of the concepts that were previously introduced are extended and refined within this chapter.

I begin by building on the overview presented in Chapter 2 to show the basis for the software restart markers model. I present the fundamental concepts behind the model and the advantages that it provides. I then discuss the disadvantages of this approach and set the stage for future chapters by presenting three different usage models for software restart markers.

3.1 Determining Where to Restart a Process

Section 2.3 discussed an important issue when supporting restartable or swappable exceptions: knowing where the process should be restarted after the exception is handled. Imprecise exception handling mechanisms can complicate the situation because there may be instructions before the faulting instruction that have not yet completed, or instructions after the faulting instruction that have completed. For example, consider the following instruction (Note: all scalar instruction examples used throughout this thesis follow the MIPS architecture format):

```
add r1, r1, 1
```

Suppose that another instruction occurs before the above `add` in program order and causes an exception, but that the `add` updates architectural state before the exception is handled. When the process resumes, if the `add` is executed again, then the final value of `r1` will be incorrect.

The problem with the `add` instruction is that it overwrites one of its input values. If that input value was preserved, it would be safe to re-execute the instruction, as the same result would be produced. To illustrate this point, consider Figure 3-1. The value of each element in array A is to be copied into the corresponding element in array B, which initially contains random values. Since A and B are disjoint, the copy can be performed in parallel—e.g. by using a vectorized `memcpy` function in a system that supports vector instructions. If the arrays are large enough, the number of elements that could be copied in parallel would be limited in an actual implementation (bounded by the maximum vector length in a vector processor); however, for the purpose of this example, assume that there are sufficient resources to copy all elements simultaneously. Now suppose that a page fault occurs when copying one of the elements, as shown in Figure 3-2. In the figure, elements that have been successfully copied at the time the fault is handled are indicated by the lack of arrows between the two arrays. Note that the processor has completed copying elements after the faulting element. If the page fault is handled and the process is later restarted from the faulting element, it would be safe to re-execute the copies of those completed elements since array A has not been modified—the last three elements of B which contain 5, 8, and 1 will simply have the values 5, 8, and 1 copied into them once again, producing the same result.

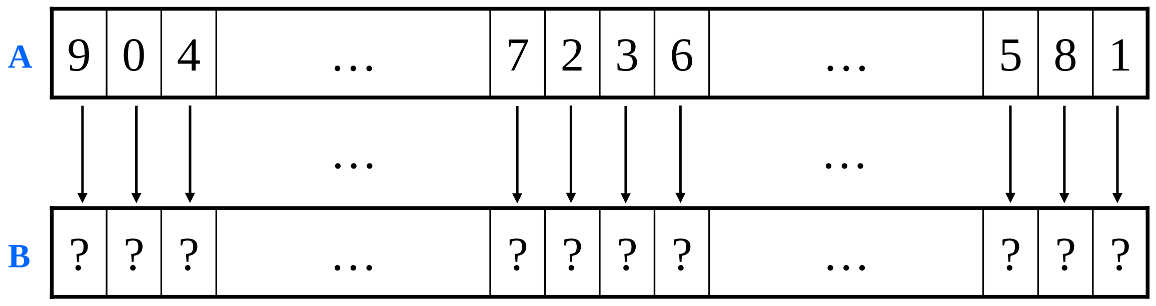


Figure 3-1: The contents of array A are to be copied into array B.

While this example avoids the problem of overwriting the input values of completed instructions which will later be re-executed, another problem still has to be addressed: certain elements before the faulting instruction have not completed execution at the time of the exception. In Figure 3-2, at least the first three elements of array A are still being copied when the page fault is handled. This could be due to an event such as a cache miss. Regardless of the cause, if all instructions are flushed from the pipeline before handling the exception, those values will not have been copied when the process resumes. Thus, if execution restarts from the faulting element, array B will end up missing some values when

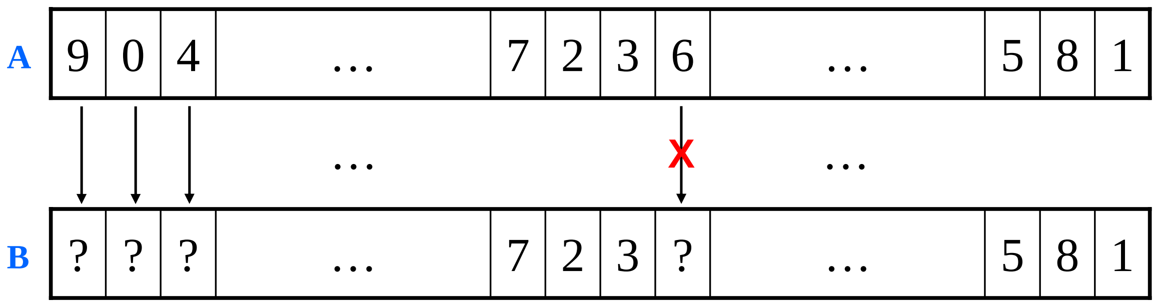


Figure 3-2: A page fault occurs when copying the element with a value of 6 from array A to array B. Some of the elements both before and after the faulting element have been successfully copied at the time of the exception, as indicated by the lack of arrows between the arrays. Also, some of the elements before the faulting element are still being copied.

the copy is finally completed.

As discussed in Chapter 2, previous exception handling proposals have suggested keeping track of which instructions before the faulting instruction have not yet completed, and selectively executing those after handling the exception. This avoids re-executing previously completed instructions, but it can also incur significant hardware overhead. We can improve upon this approach by noting that if it is safe to re-execute copies of elements after the faulting element in Figure 3-2, it is also safe to re-execute copies of elements before the faulting element. Once the page fault is handled, if the process restarts execution from the beginning of the array copy code, then that will provide the opportunity to successfully copy the first three elements of array A, as well as the faulting element. Although some elements will be copied more than once as a result of restarting from the beginning, this is safe because of the fact that the contents of array A are preserved.

The key property of this example is that the array copy operation is an *idempotent* task—in other words, it can be executed multiple times and still produce the same result. This characteristic simplifies the issue of restarting the process after handling the exception. The precise exception model also simplifies the issue of process restart, but it does so by requiring that instructions commit in order. This approach necessitates the use of structures such as physical registers and store queues to buffer intermediate values. By contrast, in the approach just described of restarting the array copy from the beginning, no intermediate values have to be buffered. If an exception occurs, all intermediate values will be recreated after the exception is handled. Thus, there is no need to place the values being written to array B into a store queue—they can be sent directly to memory. While this scheme does introduce additional issues, they will be discussed later in the chapter. First, it is important to show how these idempotent tasks can be detected.

3.2 Marking Idempotent Regions of Code

In order to enable the processor to detect idempotent tasks, two options are possible: either develop a hardware mechanism that can examine the program (which introduces the type of overhead that is a problem with precise exception implementations); or provide annotations in software indicating idempotent sections of the program. The latter approach is the foundation of the new exception model proposed in this thesis, based on *software restart markers*. Figure 3-3 shows a comparison between code executed under the precise exception model and code executed under this new approach. In the precise exception model, there is an implicit "commit" barrier attached to each instruction, as indicated by the fact that each instruction resides within its own colored region in Figure 3-3(a). (Previous work on software restart markers referred to this as a "trap" barrier, indicating that the barrier would be lifted if all earlier instructions had cleared exception checks. However, this label does not properly encompass the precise exception requirement that all earlier instructions also have to update architectural state.) If an instruction causes an exception, the implicit barrier on each instruction ensures that the exception will only be handled once all previous instructions have committed, and also that no subsequent instructions have committed. Each barrier indicates the end of a region. The beginning of a region serves as a restart point for the instructions within that region. In the event of an exception, the process will resume from the associated restart point once the exception is handled, unless the handler also processes the region's commit barrier—e.g. when emulating an instruction, the exception handler will actually perform the commit and update architectural state, and then restart from the subsequent instruction.

Figure 3-3(b) shows how code is viewed in the software restart markers model. The number of colored regions has been significantly reduced: rather than a commit barrier being associated with each instruction, it can be attached to a group of instructions. The only requirement is that each region be idempotent—i.e. the instructions in the region can be executed over and over while still producing the same final result. This model coarsens the granularity of commit and makes it possible for instructions to *commit out-of-order* within a region, without requiring any buffering of values. For example, the instructions composing the array copy task from the previous section could be placed entirely within a single region, avoiding the need to buffer the store data for array B. Since there is no longer an implicit barrier on each instruction, it is the responsibility of software to explicitly mark restart points. It does this by inserting software restart markers in the code to delimit the boundary of each region, referred to in this thesis as a *restart region*. If an instruction within a particular region causes a restartable or swappable exception, all in-flight instructions will be flushed from the pipeline, the exception will be handled, and then the process will restart execution from the head of the region. Any previously completed instructions will have their results reproduced, while any uncompleted instructions will have the opportunity to finish execution. Note that conventional precise exception semantics can be replicated by simply placing each instruction within its own region.

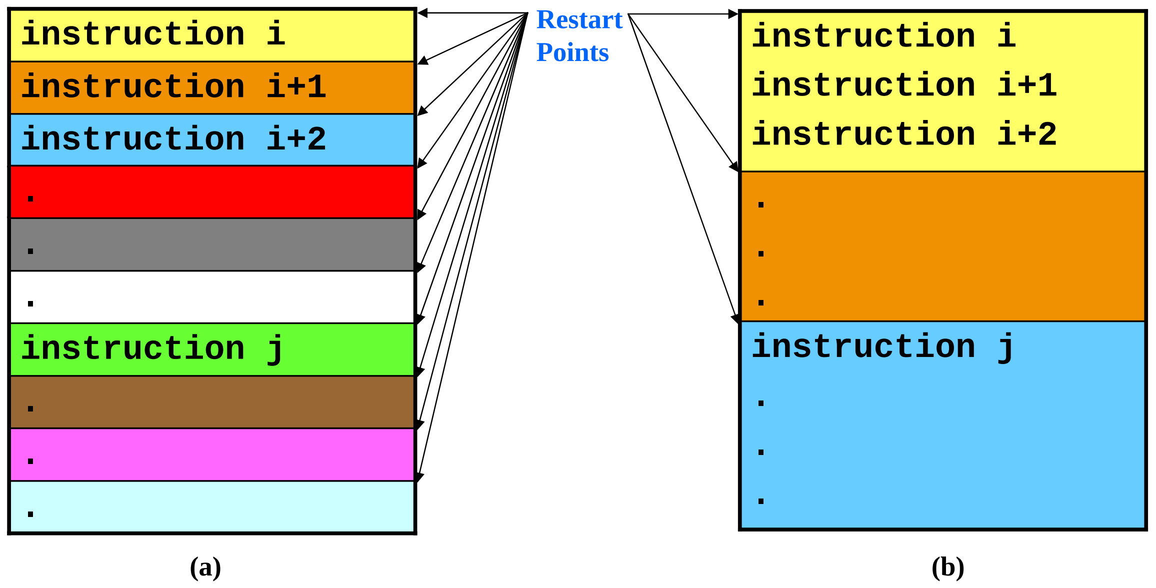


Figure 3-3: (a) Code executed under the precise exception model. There is an implicit restart point associated with each instruction. Each colored region corresponds to a commit barrier; for the precise exception model, every instruction has a barrier to enforce in-order commit. (b) Code executed under the software restart markers model. Software is responsible for explicitly marking restart points. This enables a significant reduction in the number of commit barriers, and allows greater freedom of execution within each region.

In order for the exception handler to know where the beginning of a restart region is, the address of the first instruction in the region must be stored somewhere. This address can either be held in a kernel-visible register or in memory. Since storing the address would only require a single register, and this would be the only additional hardware required for the basic software restart markers model, the approach employed in this thesis work is to hold the address in a register, called the *restart PC*. The use of a register avoids the possibility of increasing the exception handler latency if there is a cache miss when retrieving the restart address. When the commit barrier for a region is processed, the restart PC will be updated with the address of the instruction at the head of the subsequent region. To enable support for swappable exceptions, the restart PC is counted as part of the process state and is saved in the event of a context switch. As long as the exception handler has access to the restart address, the simple interface associated with the precise exception model can be retained: the only additional inputs the handler needs are the exception program counter and cause of the exception. An exception handler in the software restart markers model can thus be almost identical to a corresponding handler in the precise exception model. The only change required would be that the handler jumps back to the address in the restart PC after dealing with the exception.

Besides the fact that instructions can commit out-of-order within a region, another key distinction between the software restart markers model and the precise exception model is that the former approach divides programmer-visible state into three different categories with respect to exception handling: *checkpointed state*, *stable state*, and *temporary state*. Checkpointed state is updated each time a commit barrier is executed, and is preserved across an exception by the kernel. In the software restart markers model, the only checkpointed state is the restart PC. Stable state can be updated throughout a restart region, and is also preserved across an exception: conventional user-level architectural state such as general-purpose registers and memory fall into this category. These first two categories can also be seen in systems that implement precise exceptions; the third category of temporary state is a distinguishing feature of the software restart markers model. Temporary state is defined as state that is only valid within a restart region, and is not live across region boundaries. For example, in the array copy task from the previous section, registers would typically be used to hold intermediate values when copying elements from array A to array B. If the array copy task is placed within a single restart region, then those register values are initially defined within the region and are not used outside of the region. Thus, those registers can be treated as temporary state while the region is active. The advantage of this classification is that in the event of an exception, temporary state values will be recreated when the process restarts from the head of the region, and therefore do not need to be preserved. If the compiler can set up restart regions in a program such that certain registers are always mapped to temporary state, the overhead involved in saving and restoring those registers when handling an exception can be completely avoided. This can produce substantial performance benefits—e.g. Section 4.2 shows how an entire vector register file can be mapped to temporary state, drastically reducing context switch overhead. Additionally, the concept of temporary state makes it possible to expose internal microarchitectural state to the compiler without requiring access paths to save and restore that state. Making hidden pipeline state visible can provide benefits with respect to performance, area, and energy consumption—e.g. Section 5.4 shows how exposing ALU inputs in a VLIW architecture can reduce architectural register requirements, while Section 6.2 discusses the use of programmer-visible ALU inputs to improve performance.

Although the category of temporary state is novel with respect to exception handling, that does not mean that the use of temporary state automatically changes the architecture's programming model. For example, when dealing with a vector processor, mapping the vector register file to temporary state has no effect on the compilation process. What changes in this situation is the fact that the operating system can now ignore the vector register file when performing a context switch. The "temporary" nature of the vector register file is a run-time characteristic indicating that the values do not need to be preserved across an exception. From the standpoint of the compiler, the vector register file is simply architectural state that falls into the same general category as the scalar register file which is preserved across exceptions. Even when the concept of temporary state is used to expose pipeline details to the compiler, the changes to the programming model do not necessarily pose any novel challenges. For example, making ALU inputs programmer-visible essentially extends the number of registers the compiler can target. To maximize the benefit from this approach, the compiler should be modified so that it tries to use the ALU inputs as frequently as possible, but whether the compiler actually uses the inputs does not affect correctness, just as failing to use certain general-purpose registers would not affect correctness.

Note that although the per-instruction barriers in the precise exception model make it appear as though a single instruction is completely executed before the next one begins execution, implementations frequently work around this and actually execute instructions in parallel or out-of-order. Similarly, although each region has a barrier in the software restart markers model, it would be possible to execute instructions from multiple regions simultaneously. However, just as in precise exception implementations, this would typically incur significant hardware overhead when instructions can complete out-of-order. Only instructions from the oldest region would be allowed to update architectural state, while instructions from later regions would have their results buffered and any exception reporting would be delayed. Since this thesis is designed to develop an exception model that reduces or avoids the types of overhead seen in precise exception implementations, throughout this work the restriction is imposed that for processors which permit out-of-order instruction completion, only a single restart region can have active instructions at any given point in time. (For processors with in-order instruction completion, it is safe to have multiple active regions in the system.) This restriction means that regions must be sufficiently large to allow parallel or out-of-order execution; otherwise, performance could be significantly degraded.

Figure 3-4 summarizes the differences in the exception handling process between the precise exception model and the software restart markers model. To present the most general case, the steps shown in the figure are for dealing with a swappable exception; a restartable or terminating exception would not typically require all of the steps to be executed. Software restart markers reduce the amount of buffering in the system by allowing the processor to simply abort pending instructions in the event of an exception, rather than enforcing in-order instruction commit. Immediately flushing the pipeline also avoids the time required to undo imprecise state changes, such as in a history buffer scheme. Although software restart markers require an additional register to be saved and restored in the event of a context switch—the restart PC—the notion of temporary state can drastically reduce the overall amount of state to be preserved, leading to decreased exception handling time.

3.3 Concerns with the Use of Software Restart Markers

I have shown how the software restart markers model satisfies the first criterion for a new exception model presented in Section 2.8: saving the restart PC enables support for

Hardware must…	**Software must…**
1. Detect exception	
2. Commit earlier pending instructions	
3. Undo imprecise state changes	
	4. Save architectural state (including exception PC)
	5. Run exception handler
	6. Restore architectural state
	7. Resume process by jumping to exception PC

(a)

Before handler

During handler

After handler

Hardware must…	**Software must…**
1. Detect exception	
2. Abort pending instructions	
	3. Save checkpointed and stable state (including exception PC and restart PC)—temporary state can reduce amount of state to save and restore
	4. Run exception handler
	5. Restore checkpointed and stable state
	6. Resume process by jumping to restart PC

(b)

Figure 3-4: (a) The exception handling process for the precise exception model when dealing with a swappable exception. (b) The exception handling process for the software restart markers model when dealing with a swappable exception.

swappable exceptions; the exception handler has a simple interface; and the overhead due to buffering or context switches can be significantly reduced. The second criterion from Section 2.8 was that implementing the new model should not require unreasonable tradeoffs. I have already mentioned the fact that the only hardware required to implement software restart markers is the restart PC. This section presents the other issues with the model.

As discussed in the last section, the fact that only instructions from a single restart region at a time are executed in a processor with out-of-order completion means that small restart regions can significantly hurt performance. At the other extreme, attempting to make restart regions as large as possible may not always be the best approach, as there can potentially be significant performance overhead from executing the same instructions more than once in the event of multiple exceptions within a region. The issue of the performance impact of software restart markers will be considered throughout this thesis. However, to give an indication of the feasibility of creating large restart regions, consider that a sufficient condition for a region to be idempotent is that the inputs to the region are preserved throughout its execution. This condition means that functions which do not overwrite their input arguments are idempotent (with a notable exception being code that uses memory-mapped I/O, as it is not possible in that case to read or write the same values multiple times). Many parallelizable functions that are the focus of this work such as `memcpy()` do not overwrite their input arguments at the source code level. However, even functions that do modify their inputs can typically be made idempotent by simply copying any overwritten input values. The question to consider in that situation is whether the overhead involved in copying those values—which could be stored in registers or memory—is worth the potential benefits of using software restart markers.

Another concern with software restart markers is ensuring forward progress in the face of finite resources—i.e. avoiding livelock. For example, in a demand-paged virtual memory system the number of memory pages touched in a restart region must be less than the number of available physical pages to allow execution to proceed through the entire region without a page fault. In practice, this particular restriction is not usually significant, as any program will have a minimum physical memory requirement, and the program's working set can be sized accordingly so that it will not exceed the minimum when using software restart markers. A more significant restriction arises if TLB refills are handled in software, as the number of different pages accessed within a region must be less than the number of available TLB entries so that the region can run from start to finish without incurring a TLB miss. Although a hardware page table walker is a more suitable method to perform TLB refill in processors that execute large numbers of concurrent operations [JM98c], software-managed TLBs have proven popular in the past because they give the operating system the flexibility of choosing a page table structure [UNS$^+$94]. Chapter 4 considers the issue of livelock in more detail.

Debugging usually requires precise exception support to provide maximum benefit to the programmer; otherwise it could be difficult to know which instructions have modified the process state. There are two types of debugging to consider. For application-level debugging, the programmer can have the compiler generate code with single-instruction restart regions—i.e. replicate the precise exception model. This approach of “turning off” software restart markers is similar in concept to turning off compiler optimizations when debugging programs. Debugging optimized code continues to be problematic despite the introduction of various techniques designed to address the issue [JGS99]. Thus, the fact that it could be difficult to debug code with restart regions is not really a fundamentally new problem from that seen in existing systems. The second area to consider is compiler debugging, in which it is necessary to provide debugging support for code with software restart markers so that the compiler writer can track down the source of any incorrect restart regions. A possible solution to this problem is to have the operating system provide a software emulation of the processor when executing a buggy region. This emulation approach could also potentially be used for application-level debugging of code with restart regions, although it might be the case that the compiler writer has access to a more comprehensive debugging environment than the application-level programmer (e.g. when dealing with industrial designs). An additional observation that applies to both compilers and applications is that maintaining a precise state for debugging is motivated by the fact that users frequently deal with sequential architectures and the corresponding programming models. The recent shift toward parallel computing could lead to an increased usage of parallel programming models and architectures that do not specify execution ordering at the granularity of an instruction. If tools are developed in the future to facilitate debugging of programs even when execution has an element of nondeterminism, the same techniques could be applied to debugging code with software restart markers.

A final concern with the software restart markers approach is that it changes the memory model in an SMP system. One solution is to add fences around memory operations used for inter-thread communication. For example, consider a multithreaded version of the `quicksort` routine, in which each thread checks a shared work queue in memory for array partitions that need to be sorted. The only communication between threads occurs when accessing the work queue; once a thread pops a task from the queue and begins sorting a partition, it proceeds independently of the other threads until it needs a new task. Placing fences around the memory operations used to access the work queue allows the memory model to remain unchanged. While this topic is an interesting one to explore, as mentioned in the thesis introduction, this work focuses on exception handling within a single processor.

3.4 Usage Models for Software Restart Markers

The third criterion for a new exception model presented in Section 2.8 was that it should be applicable to a wide range of architectural paradigms. To that end, this section provides three different usage models for software restart markers. This list is not meant to exhaustively classify all possible ways to use software restart markers, but it does contain some key applications, and each model corresponds to a design that I illustrate in this thesis. Also, although the differences between the models may appear somewhat subtle, they have significant implications for the benefits that can be achieved by using software restart markers.

One point needs to be emphasized about the general usage of software restart markers. This exception model is primarily designed for explicitly parallel architectures that rely on compiler techniques to achieve a high level of performance. Architectures that implicitly exploit parallelism by using dynamic techniques such as out-of-order execution are typically less suitable for the software restart markers model, for two main reasons. First, these processors typically already incorporate hardware structures such as store queues that would be needed for precise exception support, so the additional amount of overhead required to implement exceptions precisely is reduced. Second, the model of only executing instructions within a single restart region could lead to an inefficient use of hardware resources in sections of code with small regions. For example, if the processor supports 128 in-flight instructions, and it is executing a region that only contains 3 instructions, its capabilities are being wasted. Thus, in processors such as out-of-order superscalar designs, the precise exception model would probably be more appropriate. The focus of this thesis is on parallel architectures that do not implement techniques such as out-of-order execution.

3.4.1 Black Box Model

Perhaps the most natural application of software restart markers is to implement a "black box" model. In this approach, illustrated in Figure 3-5, the system contains a main processor and a coprocessor. The main processor implements the precise exception model; for example, it could be an out-of-order superscalar design that uses a reorder buffer. The coprocessor uses the software restart markers model for exception handling. A section of code such as a function or loop that gets mapped to the coprocessor will be placed within a restart region. This provides two advantages. First, a coprocessor is frequently a specialized design such as a vector unit intended to accelerate the execution of certain types of code. However, under the precise exception model, this performance improvement can incur a substantial hardware cost in the form of buffers to hold uncommitted results. By exclusively using software restart markers to handle coprocessor exceptions, the need to buffer any results of coprocessor instructions is completely eliminated. The second advantage is that by placing a coprocessor code section within its own restart region (and ensuring that no coprocessor registers are live across region boundaries), the entire state of the coprocessor can be mapped to temporary state, avoiding the need to save and restore it in the event of an exception. This characteristic is the motivation for labeling this type of design the "black box" model: from the perspective of the exception handler, it does not matter how the coprocessor is designed or what kind of state it contains; in fact, since the coprocessor state does not need to be preserved in the event of an exception, it can actually be completely invisible to the operating system. Any type of processing element could be plugged into this black box, and it would make no difference in terms of handling the exception. Chapter 4 presents an example of this usage model in vector architectures.

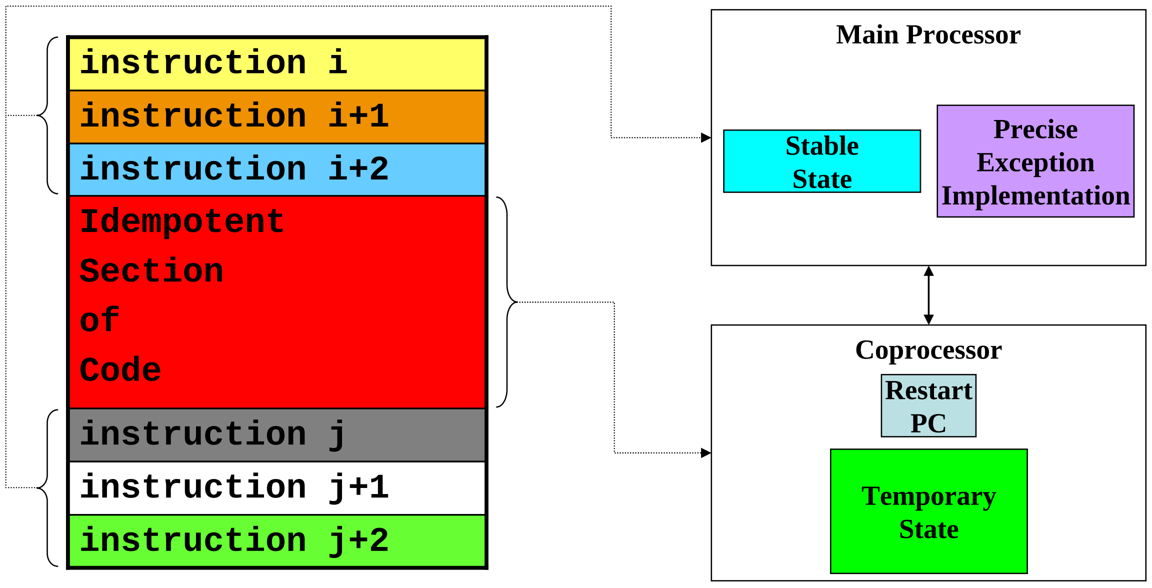


Figure 3-5: In the black box usage model, there is a main processor which implements exceptions precisely, and a coprocessor which uses software restart markers. Any segment of code that is mapped to the coprocessor will be placed within a restart region. All of the coprocessor state is temporary state in this model.

3.4.2 Partial Model

Software restart markers can also be used in a system that only has a single processor, as shown in Figure 3-6. In this model, the processor implements exceptions precisely during certain parts of the program. However, there are also certain sections of code that are grouped into restart regions. When execution enters a restart region, only instructions from within the region will be issued, and an exception will cause the process to resume from the address in the restart PC. Upon reentering a “precise” section of code, the processor will return to the approach of in-order instruction commit. Since the processor needs to support precise exceptions in parts of the program, the same types of overhead are present in this design as would be seen in a system that implements exceptions precisely throughout the entire program. An advantage of using software restart markers on a partial basis is that the use of temporary state in certain key sections of code could enable the same level of performance with a reduced amount of stable state—i.e. registers that have to be preserved in the event of an exception. This approach permits a flexible switching between precise exceptions and software restart markers. An example of this "partial" usage model is shown in Chapter 5 for VLIW architectures. Note that although the black box model also uses software restart markers on a partial basis throughout the entire program, that partial usage is only visible to the main processor; from the perspective of the coprocessor, the only exception handling approach used is software restart markers, which is why a different label is applied to that usage model.

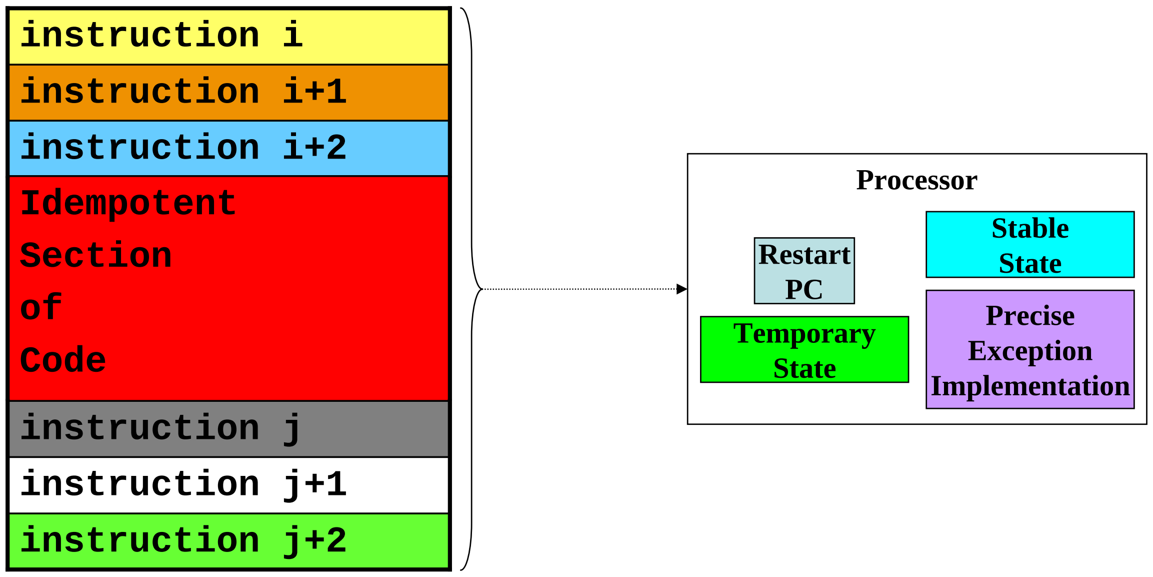


Figure 3-6: In the partial usage model, there is a single processor which has the ability to implement exceptions precisely and to incorporate software restart markers. The processor makes use of temporary state within restart regions to reduce the amount of stable state in the processor.

3.4.3 Complete Model

The partial usage model still requires support for precise exceptions. In a design such as an out-of-order superscalar, this would require the use of a reorder buffer or some other scheme. In the "complete" usage model shown in Figure 3-7, software restart markers are used throughout the entire program. Only using software restart markers eliminates the need for any mechanism to support precise exceptions; however, it also means that for sections of code with small restart regions, the processor will have about the same performance as a machine that executes instructions sequentially and serially. The benefit of using software restart markers in this model is that temporary state could enable a reduction in the use of stable state throughout the program. Chapter 6 shows how this usage model can be used in multithreaded architectures.

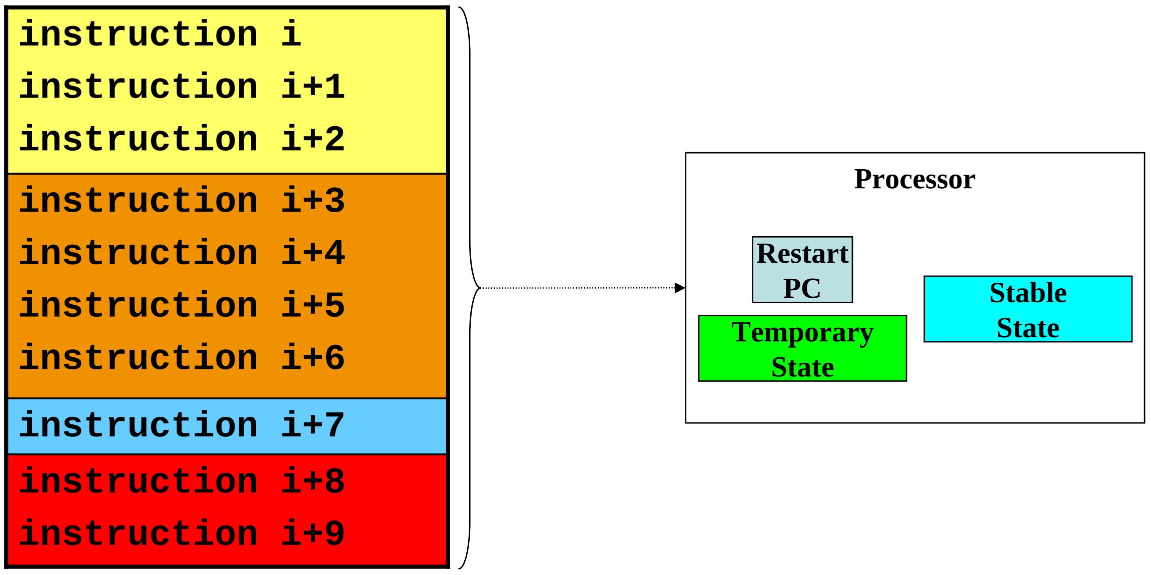


Figure 3-7: In the complete usage model, there is a processor which only uses software restart markers to deal with exceptions. Temporary state can reduce the use of stable state in the processor.

3.5 Related Work

Sentinel scheduling [MCH+92] is used to deal with exceptions in a software speculated architecture. In this work, a speculative instruction is associated with a later "sentinel" instruction, which checks whether an exception has occurred. If a correctly speculated instruction causes a restartable or swappable exception, the process will resume from that instruction, which means that all of the instructions up to the sentinel will be re-executed. This restart point imposes the requirement that the sequence of instructions between a speculative instruction and its sentinel must be idempotent, just as a restart region must be idempotent. However, the software restart markers exception model is more general in that it is designed to be used within a variety of parallel architectures, not just to recover from faulting speculative instructions. Also, software restart markers enable the use of temporary state, which can reduce context switch overhead.

Kim et al. [KOE+01] explore the notion of idempotent memory references within the context of a speculative multithreaded processor. They focus on non-parallelizable program sections—e.g. loops with cross-iteration dependences—and use compiler analysis to determine loads and stores which are idempotent within a segment. A segment is a unit of speculative execution, and multiple segments make up a region—e.g. a loop iteration could be a segment, while the entire loop could be a region. Regions execute and are committed in order, while segments can execute out of order. The purpose of their work is to reduce the pressure on speculative storage by allowing idempotent references to directly access non-speculative storage; if there is any error, the value will eventually be correctly recreated. By contrast, this thesis focuses on forming regions that are completely idempotent and using that property as the basis of an exception model for parallel architectures, regardless of whether speculation is used.

The Alpha architecture handles floating-point exceptions imprecisely by default, meaning that if an instruction causes an arithmetic trap, additional instructions past the excepting instruction might have executed—these instructions are referred to as the "trap shadow" of the excepting instruction [Cor98]. A trap shadow can be viewed as a type of restart region, although it is more restricted, as it is only used in conjunction with floating-point exceptions, and there are several conditions that must be satisfied. For example, the exception handler in the Alpha system has to determine which instruction caused the exception by beginning at the exception PC address and performing a backwards linear scan, comparing destination registers of trap shadow instructions with the registers specified by the arithmetic trap's register write mask parameter. For this linear scan to work correctly, the destination register of the excepting instruction cannot be used as the destination register of any trap shadow instruction; additionally, the trap shadow is limited to be within a basic block. By contrast, software restart markers permit arbitrary overwriting of registers within a region as long as the original input values are preserved, and regions can encompass entire functions. The Alpha's exception handler is also responsible for re-executing instructions in the trap shadow, so each of those instructions need to have their inputs preserved. By contrast, with software restart markers, only the inputs to a region—not an individual instruction—have to be preserved, and the exception handler can simply jump to the restart PC without having to deal with re-executing individual instructions.

Bell and Lipasti [BL04] examine the conditions that must be satisfied before an instruction can be committed in a superscalar processor. They implement techniques that can be used to commit instructions out of order, with the purpose of freeing up processor resources more quickly. However, their work relies on the fact that their design incorporates a reorder buffer, and they utilize the existing superscalar hardware to determine when it is safe to commit instructions out of order and introduce additional logic to collapse ROB entries. A parallel architecture that does not support techniques such as register renaming may not have existing structures to facilitate the detection of when instructions can be committed out of order. Additionally, their approach can only commit instructions out of order if it is guaranteed that no previous instructions will cause an exception. This restriction can limit potential performance gains—for example, in a parallelized loop, an iteration has to wait for all previous active iterations to clear exception checks before its instructions can commit. Finally, their work does nothing to reduce context switch overhead.

The software restart markers approach is somewhat similar to using the hardware-based checkpoint repair scheme [HP87] discussed in Section 2.3.4. In a sense, software restart markers act like checkpoints that are statically determined by the compiler: in the event of an exception, execution resumes from the last marker, just like an exception in the checkpoint repair scheme causes execution to resume from a checkpoint. The software restart markers model is less flexible than a hardware checkpointing scheme, as restart regions have to be idempotent and only one region is active at a time. However, software restart markers have the advantage of significantly reduced hardware overhead. Also, checkpoint repair incurs significant overhead for processors with a large amount of architectural state, as a backup is needed at every checkpoint. Software restart markers avoid this overhead and can also use the concept of temporary state to drastically reduce context switch time for processors with a significant amount of architectural state.

Melvin and Patt [MP89] discuss three types of atomic units and how the sizes of those units can affect performance. An architectural atomic unit, or AAU, is defined by the granularity at which an architectural-level exception can occur. For example, in a VLIW architecture that supports precise exceptions at the instruction level rather than at the operation level, an entire instruction would be an AAU. A compiler atomic unit, or CAU, is defined by the granularity at which a compiler can generate code; for a VLIW architecture, this would be a single operation. Finally, an execution atomic unit, or EAU, is implementation-specific, and is defined as an indivisible group of operations issued by the hardware. An EAU can be smaller than an AAU (e.g. in a microcoded machine), equivalent, or larger (if the hardware dynamically groups AAUs into indivisible units of work). However, supporting EAUs which are larger than AAUs can be complicated because of the need to handle exceptions at the granularity of an AAU, which requires the ability to switch "modes" in the event of an exception and handle one AAU at a time. For example, IBM's POWER4 [TDJSF$^+$02] and POWER5 [SKT$^+$05] processors group up to five instructions and dispatch them together. If an instruction within the group causes an exception, all of the group's instructions are flushed and then the instructions are individually replayed in the pipeline. Melvin and Patt argue that large EAUs can be beneficial for performance (due to the fact that a chunk of work will be issued at one time), which implies that it would be good to have large AAUs as well—i.e. the granularity of exception handling should be coarsened. This approach is similar to the software restart markers model: an AAU is somewhat analogous to a restart region, although in the latter case a region can be partially executed. A restart region can contain large EAUs such as VLIW or vector instructions, potentially improving performance. Melvin and Patt also state that another benefit of large AAUs is that they could potentially reduce the amount of architectural state in the processor, since temporary values within an AAU do not have to go through the register file—they can be referred to by their position within the block. This idea is similar to the concept of temporary state in a restart region. Finally, the authors advocate for small CAUs, which give the compiler greater control over the work to be done. Software restart markers do not impose any restriction on the granularity of compiler-visible units of work within a region, so small CAUs are possible.

There have been several designs which coarsen the granularity of commit. For example, the Transmeta Crusoe processor has software-controlled exception barriers at the borders of blocks of x86 code that have been translated into the native VLIW format [Kla00]. This scheme has a future file for registers and a speculative store buffer that allow state updates to be revoked if an exception is encountered in a translated block. A commit operation copies state updates into the architectural state. A similar approach is used in the TRIPS architecture [SNL$^+$03], in which programs are partitioned into blocks of instructions that commit atomically. Exceptions are "block precise" and are handled at block boundaries. Each active block has access to its own set of registers and execution resources. A third example is found in the use of transactional memory [HM93], which is designed to simplify parallel programming. In this scheme, a memory transaction is an atomic sequence of memory operations which either commit—i.e. execute completely—or abort—i.e. have no effect. To ensure that a transaction can be aborted, some form of backup state must be maintained to permit rollback. In general, all of these schemes rely on state updates being buffered in some manner, as a block must either be completely committed or have no effect. By contrast, software restart markers allow updates to architectural state within a restart region, eliminating the need for buffering.

The Inmos transputer family has a type of temporary state in the form of a three-register evaluation stack that is not preserved across context switches [WS85]. However, context switches are only allowed to occur when the stack contains no live values, which also prevents support for features such as demand paging. By contrast, software restart markers allow context switches to occur at any point in the program.

3.6 Summary

This chapter presented *software restart markers* as an alternative to implementing precise exceptions. In the software restart markers model, the compiler is responsible for marking restart points in the code; in the event of an exception, the process will resume from the most recent restart point. To ensure correct execution, the sequence of instructions between restart points—referred to as a *restart region*—must be idempotent, meaning that it can be executed multiple times and still produce the same result. This exception model has two primary benefits: it permits *out-of-order commit* within a region, eliminating the need for costly buffering mechanisms; and it introduces the notion of *temporary state*, which can significantly reduce context switch overhead. While using software restart markers does involve certain tradeoffs, these drawbacks are often acceptable, as will be brought out in more detail in subsequent chapters. This chapter also introduced three different usage models for software restart markers, which correspond to designs that will be developed in this thesis.

Chapter 4

Software Restart Markers in Vector Architectures

In this chapter, I illustrate the black box usage model of software restart markers by showing how it can be applied to vector architectures. I begin by presenting an overview of vector architectures and the problems with exception handling in these designs. I then show how software restart markers can be used in vectorized code. Finally, I discuss the issue of livelock and also present some optimizations that can help to reduce performance overhead from using software restart markers. While the livelock and performance issues are addressed within the context of vector architectures, the techniques that are presented in this chapter can be used in other designs which employ the software restart markers model.

4.1 Vector Architecture Background

A vector architecture encodes data-level parallelism, or DLP, by allowing a single instruction to indicate that multiple elements of data can be operated on independently. Figure 4-1 shows how a basic vector processor might execute a vector-add instruction. Since the compiler checks for dependences, the processor does not need complex hardware to perform this task—it can simply issue the independent operations specified by the vector-add. This also leads to reduced instruction fetch and decode overhead. Parallelism can be enhanced by incorporating multiple *lanes* [HP07, Appendix F], or parallel pipelines, which are used to execute multiple operations within an instruction simultaneously. Additionally, the regular access patterns seen in vector processing make it possible to perform datapath optimizations for both registers and memory.

With all of these benefits, it is easy to see why vector architectures can provide high performance with relatively low complexity for data parallel codes across a wide range of application domains ranging from supercomputing [Rus78, EAE$^+$02] to embedded media processing [Asa98, LS98, QCEV99, Koz02]. However, extensive vector capabilities have yet

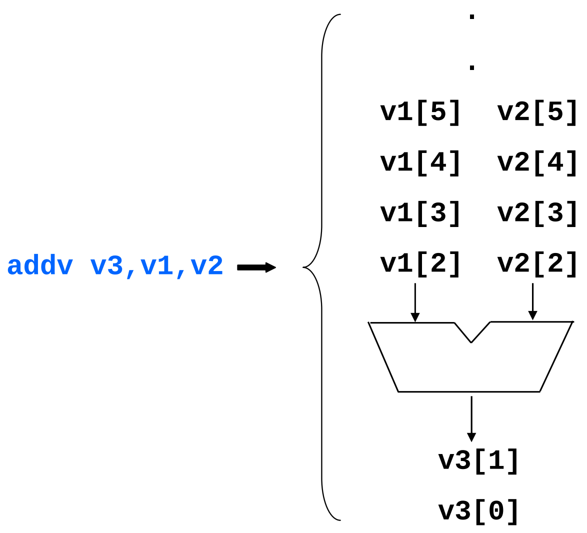


Figure 4-1: Execution of a vector-add instruction in a single-lane vector machine.

to appear in recent commercial products other than newer versions of traditional large-scale vector supercomputers [Bro02, KTHK03]. Although short-vector multimedia extensions such as Intel's MMX/SSE [PW96, RPK00] and Apple/IBM/Motorola's Altivec [DDHS00] provide some of the benefits of vector processing, they cannot provide the same degree of complexity reduction in instruction fetch, decode and register renaming, or the same improvement in sustained memory performance as a traditional long-vector architecture.

One of the factors that has hindered the widespread adoption of vector architectures is the difficulty of supporting demand-paged virtual memory in these machines. Virtual memory is typically implemented in superscalar designs with precise exception handling mechanisms, but vector machines introduce two major complications. First, many vector units are designed to run decoupled from the scalar unit, which runs far ahead fetching instructions for subsequent loop iterations [EV96], and vector instructions can be long running, so a vector instruction might complete hundreds of clock cycles after a scalar instruction that follows it in program order. The problem of out-of-order completion is further exacerbated when multiple vector instructions are executing simultaneously. This issue makes supporting precise exceptions through conventional register renaming and reorder buffer techniques very expensive due to the length and number of instructions in flight. Second, vector units have far more architectural state than scalar units. For example, a simple scalar processor with a 32-entry integer register file might have 128 bytes of architectural register state, not including any additional control registers. Even if floating-point registers are added, the amount of state would probably only increase by a factor of 2 or 3. By contrast, a vector register file is substantially larger than a scalar register file, as shown in Figure 4-2. Adding a vector unit to a processor could easily increase the amount of architectural register state by 2 orders of magnitude [HP07, Appendix F]. This dramatically lengthens context switch times for precise exceptions.

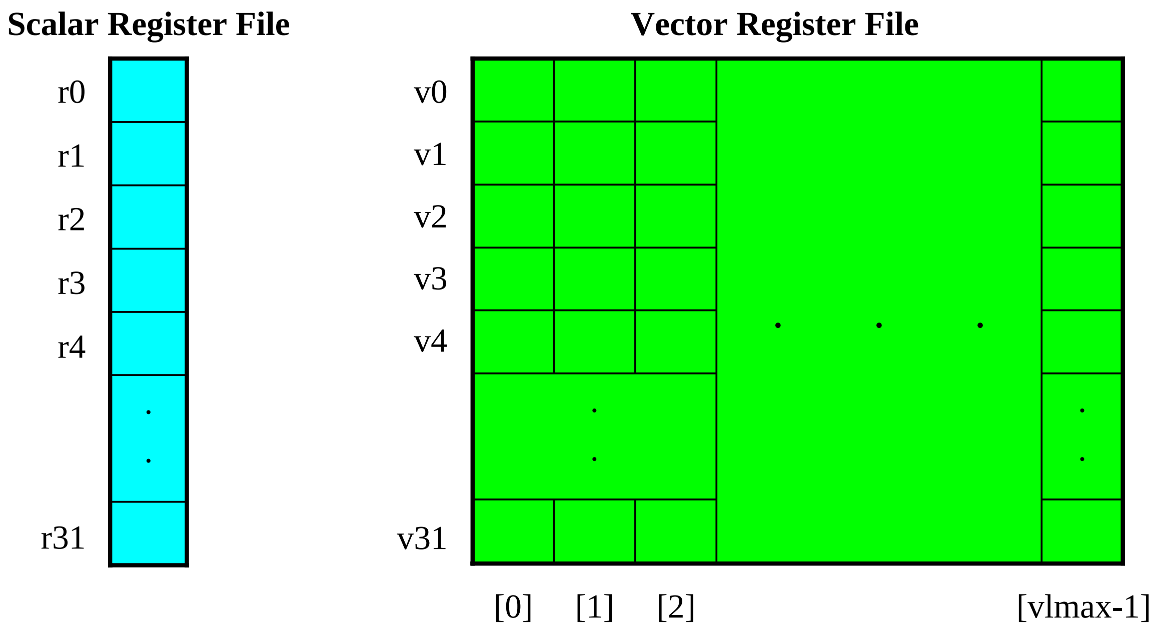


Figure 4-2: Illustration of the size difference between a scalar register file and a vector register file.

Despite these problems, there have been some implementations of precise exceptions in vector processors. The IBM System/370 vector facility [Buc86] only allows one vector instruction to execute at a time. Although this restriction avoids the problems caused by multiple in-flight vector instructions, it severely limits potential performance. The IBM design also maintains in-use and dirty bits for the vector registers to avoid unnecessary register saves and restores in the event of a context switch. However, this approach still requires a significant amount of state to be saved and restored when the vector unit is active. There are other vector designs which use register renaming to facilitate in-order commit [EVS97, EAE$^+$02, KP03]. This technique can enable virtual memory support but requires a substantial amount of buffering in the form of additional physical vector registers. Although register renaming can provide other benefits if it is used to support out-of-order execution with a limited number of architectural vector registers [EVS97], this performance advantage will be reduced for newer designs with more architectural vector registers.

An imprecise approach to virtual memory support is employed by the DEC Vector VAX [BSH91], which uses microarchitectural swapping to preserve the internal state of the vector unit in the event of an exception. However, saving pipeline state can hurt performance and energy consumption, and also complicates the OS interface. The DEC design attempts to reduce context switch overhead by only saving the vector unit state if a vector instruction is encountered in the new process. However, this technique provides little benefit unless the vector unit is used by very few processes.

Smith and Pleszkun [SP88] propose an approach in which each architectural vector register has two physical registers associated with it. A specific architectural register can only be targeted by one in-flight instruction at a time. One physical register holds the “old” value, while the other physical register will receive the updated value. If an exception occurs before the instruction can commit, the “old” physical register will become current again. The problem with this scheme is that it can limit performance if there are a small number of architectural vector registers; also, for maximum effectiveness it relies on the compiler using a round-robin register allocation policy. The authors also discuss the idea of allowing a vector instruction to partially complete, relying on length counters to restart from an intermediate state after an exception. This approach adds hardware overhead, particularly if multiple vector instructions are in-flight simultaneously.

Given the drawbacks of the above approaches, vector supercomputers typically have not supported virtual memory [Rus78, UIT94, KTHK03]. This lack of support has been acceptable in the supercomputing domain, since supercomputer jobs are designed to fit into physical memory so that expensive CPUs do not wait while pages are swapped in from disk. Also, supercomputers commonly use batch scheduling, where a queue manager ensures the set of running jobs can fit in physical memory. However, if traditional vector architectures are to become more widely used in general-purpose systems, they need to support virtual memory.

4.2 Restart Regions in Vectorized Code

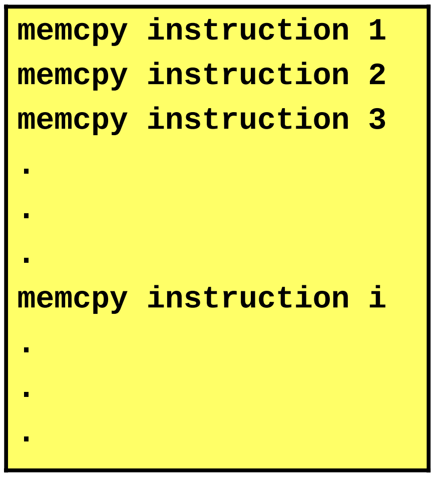
This section illustrates how restart regions can be formed in vectorizable functions. For the purpose of applying the software restart markers model, functions can be divided into two major categories: functions that do not modify any input memory arrays (including functions that have no input arrays at all, such as the `memset` routine); and functions that modify one or more input memory arrays. These two categories will be considered separately.

4.2.1 Functions With No Overwritten Input Memory Arrays

To illustrate how software restart markers can be applied to vector architectures for functions which do not modify any input memory arrays, consider the C `memcpy` function, for which the prototype is given below.

```
void* memcpy(void *out, const void *in, size_t n);
```

This function copies the first `n` bytes of `in` to the first `n` bytes of `out`. The input and output arrays do not overlap in memory, otherwise the behavior is undefined. The preservation of the input memory values means that the `memcpy` instructions can be placed within a single restart region, as shown in Figure 4-3. If an instruction within the function causes an exception, the process can be restarted from the beginning of the function.



```
memcpy instruction 1
memcpy instruction 2
memcpy instruction 3
.
.
.
memcpy instruction i
.
.
.
```

Figure 4-3: The entire `memcpy` function is placed within a single restart region.

Although the `memcpy` function may be idempotent at the source code level, things can change when the code is actually compiled. To illustrate this, consider the sample vectorized loop shown in Figure 4-4, which could be used as a partial implementation of `memcpy`. (The loop does not implement the complete functionality of `memcpy`, as it does not account for misaligned byte vectors and it can only be used when the number of bytes is a multiple of 512.) The code in this figure and all subsequent examples in this chapter use the VMIPS instruction set [HP07, Appendix F]. *Strip mining* [Lov77] is used to perform the work of multiple scalar loop iterations simultaneously. Each vector register holds 64 8-byte elements, so each strip-mined iteration copies 512 bytes. In the VMIPS code, the argument registers are overwritten in each loop iteration. This problem can be handled by copying the argument register values to other registers at the beginning of the restart region. The registers with the copied values can then be used to execute the loop. Figure 4-5 shows the `memcpy` loop inside a restart region. Although copying register values can increase the register pressure, the effect will often be insignificant—typically only the number of iterations and a few memory pointers will have to be preserved, and these will be held in scalar registers, not vector registers. Thus, any performance overhead will usually be small compared to the total execution time of the function.

Placing all of the vectorized code within a single restart region allows the vector instruction results to be committed to architectural state as soon as they are available. Since the `memcpy` code is a DOALL loop, its iterations can be executed in parallel, restricted only by the available execution resources, without requiring any buffers in the form of additional physical registers or store queues. Additionally, no vector registers are live across the boundaries of the restart region, which means that the vector register state will be recreated each time the region is restarted, and the vector register file can be treated as temporary state. In general, unless techniques such as interprocedural dependence analysis [PW86] are used, vector registers are typically only live within a single function (and usually only

```
# void* memcpy(void *out, const void *in, size_t n);
loop: lv v0, a1            # Load input vector
      sv a0, v0            # Copy the data to output vector
      addiu a1, 512        # Increment the input base register
      addiu a0, 512        # Increment the output base register
      subu a2, 512         # Decrement the number of bytes remaining
      bnez a2, loop        # Is loop done?
done:
```

Figure 4-4: Vectorized `memcpy` loop.

```
begin restart region
      move t0, a0          # Copy the argument registers
      move t1, a1
      move t2, a2
loop: lv v0, t1            # Load input vector
      sv t0, v0            # Copy the data to output vector
      addiu t1, 512        # Increment the input base register
      addiu t0, 512        # Increment the output base register
      subu t2, 512         # Decrement the number of bytes remaining
      bnez t2, loop        # Is loop done?
end restart region
done:
```

Figure 4-5: Vectorized `memcpy` loop placed inside a restart region. The argument registers are copied to working registers when the region is entered, and the working registers are used to execute the loop.

within a loop iteration). This allows the vector unit to be treated as a black box that can be ignored by the operating system. Figure 4-6 shows how the software restart markers model leads to a drastic reduction in context switch overhead by avoiding the need to preserve the vector register file.

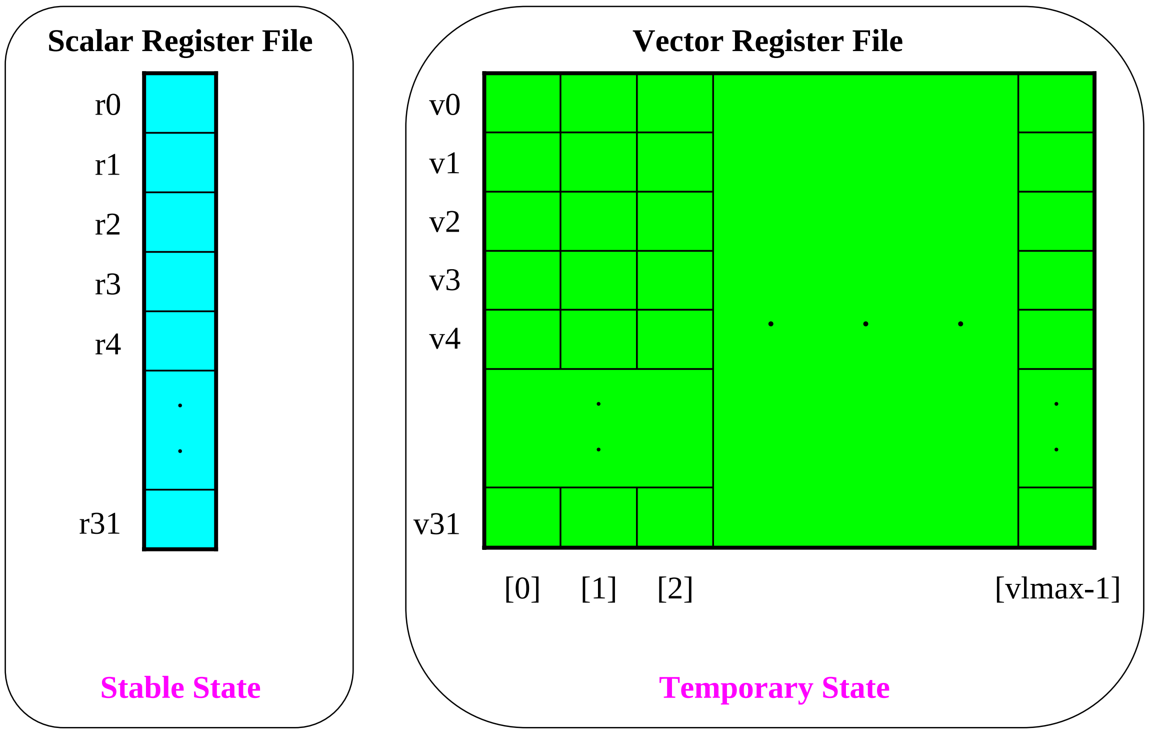


Figure 4-6: In the software restart markers model, the vector register file is treated as temporary state that does not have to be saved and restored across an exception.

4.2.2 Functions With Overwritten Memory Input Arrays

As shown in Figure 4-5, overwritten input registers to a region can simply be copied to backup registers so that the values will be preserved. Input memory values that are overwritten are more difficult to handle, as the amount of data is often much larger. Although a vectorizing compiler may apply transformations to remove dependences that inhibit restart region formation, certain types of code will still cause problems. Consider the vectorized function in Figure 4-7, which takes an input array `in` of size `n` and multiplies each element by 2, overwriting the input values. To form a restart region, the `in` array can be copied to a temporary buffer allocated in memory, as shown in Figure 4-8. The temporary buffer can then provide the input elements, while the original array will be updated with the output elements. In the event of an exception, the original input values will still be available and execution can resume from the beginning of the restart region. Using a temporary buffer transforms the function so that it has disjoint input and output arrays, making the problem similar to the last section.

```
# void multiply_2(char *in, size_t n);
loop: lv v0, a0            # Load input vector
      mulvs.d v0, v0, f0   # Multiply vector by 2
      sv a0, v0            # Store result
      addiu a0, 512        # Increment the base register
      subu a1, 512         # Decrement the number of bytes remaining
      bnez a1, loop        # Is loop done?
done:
```

Figure 4-7: Vectorized `multiply_2` loop which multiplies each element of an array by 2 and stores the result into the same array.

```
# void multiply_2(char *in, size_t n);
      Allocate temporary buffer of size n pointed to by t2
      memcpy(t2, a0, a1)  # Copy input values to temporary buffer
      move t3, t2         # Preserve start address of buffer
begin restart region
      move t0, a0         # Get original inputs
      move t1, a1
      move t2, t3
loop: lv v0, t2           # Load input vector from temporary buffer
      mulvs.d v0, v0, f0  # Multiply vector by 2
      sv t0, v0           # Store result into original array
      addiu t0, 512       # Increment first base register
      addiu t2, 512       # Increment second base register
      subu t1, 512        # Decrement the number of bytes remaining
      bnez t1, loop       # Is loop done?
end restart region
done:
```

Figure 4-8: Vectorized `multiply_2` loop with software restart markers that copies all of the original array elements into a temporary buffer and uses that buffer to supply the input values during execution.

While this approach does not require actual hardware buffers, it could still incur substantial space overhead in memory depending on the size of the dataset. This issue can be alleviated by using strip mining, where the single large restart region encompassing the entire loop is broken up into smaller regions containing a certain number of loop iterations. Another drawback of this scheme is the performance overhead due to copying all of the memory elements, which could be significant in functions that perform little computation on each element. However, although the extra copies could make the vectorized code slower, the modified code will often still be much faster than the alternative of executing scalar code. The primary purpose of applying the software restart markers model to vector architectures is to make it more practical to include a vector unit in a processor that might otherwise devote that chip area to more inefficient scalar execution resources. Even a vector unit that has reduced performance on certain types of codes because of restart region formation will often be preferable to having no vector unit at all. In situations where the overhead of creating idempotent regions is too high—e.g. code with a tiny dataset—the processor can revert to executing the scalar version of the code. To enable this, the compiler can generate both vector and scalar versions of the code, and add a test at the beginning of the function to determine which version to execute, as shown in Figure 4-9. Occasionally falling back to scalar code execution is acceptable since software restart markers are not intended to handle every single vectorized program for all datasets, but rather to handle the vast majority of code using a low-cost exception handling mechanism.

```
# void multiply_2(char *in, size_t n);
      sltiu t0, a1, 64        # Is n less than threshold?
      bnez t0, scalar_version # If so, use scalar version
# Vector version of function with restart regions here
      .
      .
      j done
scalar_version:
# Scalar code without restart regions here
      .
      .
done: # return from function
```

Figure 4-9: Code for the `multiply_2` loop that consists of both a vectorized implementation with software restart markers and a scalar implementation without software restart markers .

Although copying the array elements when necessary provides a suitable solution to dealing with overwritten input arrays, I present two other approaches for the sake of completeness. One alternative method of inserting software restart markers in the code for the `multiply_2` function is to divide each loop iteration into two restart regions, as shown in Figure 4-10. However, this technique sacrifices performance in machines that implement vector chaining, as the store will not be able to chain to the multiply. Additionally, performance will be degraded in systems that otherwise decouple the scalar unit and vector unit, as the scalar operations at the end of the loop have to wait for the vector store to complete, which in turn delays the vector load in the subsequent iteration.

```
loop:
begin restart region
      lv v0, a0            # Load input vector
      mulvs.d v0, v0, f0   # Multiply vector by 2
begin restart region
      sv v0, a0            # Store result
end restart region
      addiu a0, 512        # Increment the base register
      subu a1, 512         # Decrement the number of bytes remaining
      bnez a1, loop        # Is loop done?
done:
```

Figure 4-10: Vectorized multiply_2 loop with software restart markers that divides each loop iteration into distinct restart regions. An explicit *end restart region* instruction is not included for the first region since that region is automatically terminated by the *begin restart region* instruction for the second region.

A separate issue with the code in Figure 4-10 is that vector register values are live across restart region boundaries. Thus, for correctness the vector register state must now be saved and restored around a context switch, removing one of the primary advantages of using software restart markers. The context switch overhead could be somewhat alleviated by adapting existing techniques. For example, software can set a status bit on the *begin restart region* instruction to inform the kernel whether the vector register state is live on a fault. The status bit must then be saved along with the process state to indicate if the vector register state should be reloaded when swapping the process back in. However, as discussed in Sections 2.4.3 and 4.1, the benefits of this and other techniques to reduce context switch depend on multiple factors. As a result, the approach discussed earlier of copying the array elements is probably preferable.

One more alternative to copying array elements that is interesting to consider is to use a speculative vector-store instruction in the code of Figure 4-7, assuming speculative instructions are supported by the architecture. For example, a speculative bit could be associated with each cache line. A speculative vector-store instruction would be allowed to update the cache, but the speculative bits for the corresponding lines would be set. Only when the speculative bits are reset—at the end of a restart region—would the system allow updated cache lines to be written back to memory. In the event of an exception, all cache lines with speculative bits set would simply be flushed. To avoid having too many outstanding speculative memory operations, the strip mining technique described previously can be used so that a restart region only encompasses a small number of loop iterations. The problem with this approach is that it requires support for dealing with speculative stores and ensuring that the cache—or any other auxiliary store buffer that is included in the pipeline—does not overflow because of too many outstanding speculative operations. Note that the maximum number of speculative vector-stores within a restart region depends not just on the cache capacity, but also the associativity and the memory access pattern. While using speculation could be an approach to consider, the drawbacks will probably outweigh the benefits.

4.3 Avoiding Livelock for TLBs with Software Refill

One of the concerns with software restart markers discussed in the previous chapter was the issue of livelock. In particular, if TLB refills are handled in software, the memory operations in a restart region cannot touch more pages than the number of available TLB entries. There are two initial points to consider regarding this problem. First, most commercial architectures use hardware to refill the TLB, including Intel machines and PowerPCs [JM98a], and a TLB miss does not cause an exception in these systems, so TLB-refill livelock is not a concern. Since refilling TLBs in hardware is more appropriate for vector processors—and in general, for any design that supports a large number of in-flight operations [JM98c]—it is unlikely that TLB-refill livelock would actually be an issue in most systems that use software restart markers. (As mentioned in Section 3.3, the other potential source of livelock—allowing the number of unique virtual pages in a restart region to exceed the number of physical pages—is unlikely to be a concern in any reasonably designed program.) However, it is possible that a vector unit could be added to an existing design which uses a software-managed TLB, such as a MIPS or Alpha processor. For example, the Tarantula vector extension to the Alpha architecture [EAE$^+$02] employs software TLB refill. This leads to the second point, which is that the issue of livelock in vector processors with software-managed TLBs is not unique to software restart markers. Vector machines that implement precise exceptions at instruction-level granularity and use software TLB refill have to make the TLB large enough to allow a single vector memory instruction to complete without incurring a TLB miss. For example, if the maximum vector length is 128, the TLB has to support at least 128 different page mappings since each element of a vector memory access can touch a different page.

One approach to avoiding livelock with software restart markers when dealing with a software-managed TLB is to limit the number of memory operations within a restart region to be no greater than the number of available TLB entries. The problem with this solution is that performance could be severely degraded—e.g. if there are 128 available TLB entries, and the vector length is 128, then a restart region can only contain a single vector memory instruction. Since unit-stride vector memory accesses are so common [HP07, Appendix F] (around 80% of all vector memory accesses according to Espasa et al. [EAE$^+$02]), it is overly conservative to assume that each operation within a vector memory instruction touches a different page. In the `memcpy` function, copying a single page of data would typically require several vector memory instructions, since the memory accesses have a stride of 1. Thus, creating large restart regions is still possible by using compile-time analysis to determine an upper bound on the number of pages touched within a region. For example, if the program loads 128 KB of input data within each restart region, and the page size is 128 KB, then it can be statically determined that at most 2 distinct input data pages will be accessed in a region (the actual number of pages touched depends on the alignment of the data). Note that the page size and number of available TLB entries are design parameters that have to be exposed to the compiler.

In general, where the compiler can determine strided memory accesses are used and the stride is small, it can often create restart regions without having to account for the possibility of livelock. In order to handle datasets that are too large to be completely mapped by the TLB, strip mining can be employed so that a new restart region is started before there is any danger of overflowing the TLB. However, for code with indexed memory accesses or strided memory accesses with large strides, restart region formation may be significantly restricted if software TLB refill is used, possibly leading to performance degradation. This is a drawback of software restart markers that should be taken into consideration when deciding on an exception model.

4.4 Performance Optimizations

Another concern discussed in the previous chapter was the potential performance overhead if multiple exceptions occur within a restart region, causing the same instructions to be executed more than once. This re-execution can be particularly harmful in a system with a software-managed TLB if a restart region contains accesses to several different data pages. For example, if maximally sized restart regions are used with the `memcpy` routine, then there will be a significant overhead from performing the same memory transfers multiple times. After copying the first page of data, a TLB miss will occur when the second page of data is accessed, causing the program to begin from the head of the restart region and copy the first page again. The subsequent TLB miss on the third page of data will cause the first and second pages to be copied again, and this pattern will continue, with the overhead becoming worse as more of the restart region is processed. Additionally, if a context switch occurs, then when the original process resumes, all of the TLB mappings will have been flushed. This section discusses two optimizations that can be used to reduce the performance overhead of software restart markers: the first is specifically designed for systems with software-managed TLBs; while the second can be used regardless of the TLB refill method, and reduces performance overhead for other types of exceptions as well.

4.4.1 Prefetching for Software-Managed TLBs

One method to reduce performance overhead of repeated restarts with a software-managed TLB is to prefetch the mappings that will be needed for each region. The idea of prefetching TLB entries to avoid misses has been previously explored within various contexts [BKW94, PA95, SDS00, KS02b, KS02a, PLW+06], although many of those designs are hardware-based and intended for systems that refill the TLB using a finite state machine. A notable exception is the work of Park and Ahn [PA95], who discuss the approach of exploiting compile-time knowledge to statically determine points in the program where compulsory TLB misses would occur, and inserting explicit TLB prefetch instructions before those points to avoid any misses. The same technique can be used with software restart markers. As long as the data pages that will be accessed within each restart region can be statically determined, either explicit prefetch instructions can be used or a value can simply be loaded from each page, causing the corresponding mapping to be placed in the TLB. Since the use of a software-managed TLB already requires the compiler to perform an analysis of the pages accessed in each region to avoid livelock, this information can also be used for prefetching. Even if the memory accesses in a region are data-dependent, prefetching can still be used if the entire dataset can be mapped by the TLB—i.e. the dataset size is less than the available TLB capacity. The compiler can simply load a byte from each page in the dataset, and place all of the memory accesses within a single restart region.

Prefetching only helps with TLB misses, and does not avoid having to repeat work for other types of exceptions, such as timer interrupts. Also, the compiler must employ a suitable heuristic to determine when the overhead introduced by adding prefetching instructions outweighs the potential benefit.

4.4.2 Counted Loop Optimization

When dealing with counted DOALL loops, which make up the bulk of vectorizable code, another way of avoiding wasteful repeated restarts is to only restart execution from the *earliest uncompleted loop iteration* after handling an exception, instead of restarting at the beginning of a region. This can be accomplished by adapting the concept of *recovery blocks*, which are used in [ADM95] to alleviate the overhead associated with the original sentinel scheduling work. Recovery blocks can similarly be used with software restart markers to reduce performance overhead. To illustrate this, consider that if an instruction inside a vectorized DOALL loop triggers an exception, there may be multiple active loop iterations. Additionally, some number of iterations may have also fully completed and updated architectural state. Suppose the first i iterations have completed execution, and subsequent iterations are in various stages of completion. In the basic software restart markers model, an exception will cause the process to resume from the very beginning of the loop, causing iterations 1 through i to be re-executed. With the counted loop optimization, after the exception is handled, the process will restart at the head of a block of fixup code. This code will adjust the loop counters and pointers to equal the values associated with the *earliest uncompleted loop iteration*—i.e. iteration $i+1$. The end of the code then branches back to the head of the loop, avoiding the re-execution of previously completed iterations. Figure 4-11 presents a modified version of the vectorized `memcpy` loop which incorporates the counted loop optimization.

```
      move t0, a0          # Copy the argument registers
      move t1, a1
      move t2, a2
      move lc, 0           # Initialize loop iteration counter
begin restart region (put fixup code address into restart PC)
loop: lv v0, t1            # Load input vector
      sv t0, v0            # Copy the data to output vector
      addiu t1, 512        # Increment the input base register
      addiu t0, 512        # Increment the output base register
      subu t2, 512         # Decrement the number of bytes remaining
      commitbar_loop       # Non-blocking instruction that increments
                           # loop iteration counter when all previous
                           # instructions have completed
      bnez t2, loop        # Is loop done?
done:
end restart region
fixup_code:
      move t3, lc          # Get value of loop iteration counter
      sll t3, 9            # Each iteration copies 512 bytes
      addu t0, a0, t3      # Set output base register
      addu t1, a1, t3      # Set input base register
      subu t2, a2, t3      # Set loop counter
      j loop               # Restart loop
```

Figure 4-11: Vectorized memcpy loop that uses the counted loop optimization. A loop iteration counter is incremented by the `commitbar_loop` instruction upon the completion of each iteration. This counter can then be used to restore all of the induction variables in the event of an exception.

The loop in Figure 4-11 uses a special barrier instruction—`commitbar_loop`—that will increment a loop iteration counter once all previous instructions have completed execution. If an exception occurs, the counter can be used in conjunction with the original pre-loop values of the induction variables to skip past all of the completed iterations. Note that there is still only a single explicit restart region around the entire loop, and only a single restart address is used: the address of the head of the fixup code block. The `commitbar_loop` instruction does not need to update the restart PC because the same segment of code is being executed in a loop; however, the instruction effectively creates "mini-regions" within the loop, each of which correspond to a single iteration. Multiple mini-regions can be active simultaneously. With the basic software restart markers model, having multiple active restart regions would be problematic because results from later regions would have to be buffered. However, with the counted loop optimization, having multiple active mini-regions, or iterations, is acceptable because they satisfy two conditions. First, since the loop as a whole is idempotent, and only DOALL loops are considered (no values are communicated between iterations), each iteration is automatically idempotent and so it is safe to allow later iterations to update architectural state. Second, because the iterations are independent of each other, it is simple to create the fixup code block: updated induction variable values are easily reconstructed by using just a counter and the original pre-loop values; by contrast, it would be very difficult to deal with a non-induction variable that is live across iterations.

The counted loop optimization can reduce re-execution overhead regardless of the type of exception that is handled. It relies on the ability to determine when all instructions before an instance of the `commitbar_loop` instruction have completed, but this can typically be accomplished by using existing pipeline mechanisms to track the status of in-flight instructions. If the loop iteration counter is kept in a register, then that register also has to be preserved across a context switch. However, even a total hardware overhead of two registers—the other register being the restart PC—is a small cost for the other benefits provided by using software restart markers.

4.5 Summary

This chapter discussed the advantages of vector architectures and the problems with implementing precise exceptions in these designs. It showed how software restart markers can be used to treat the vector unit like a black box that can be ignored by the operating system. It is typically straightforward to form restart regions in vectorized functions that do not have any overwritten input memory arrays; however, even in functions that do modify their input arrays, restart regions can still often be created with little overhead. The issue of livelock is a concern for systems with software-managed TLBs, but since most code has unit-stride vector memory accesses, livelock can be easily avoided in many situations. Finally, the techniques of prefetching for software-managed TLBs and the counted loop optimization can be used to reduce any performance overhead from software restart markers.

Chapter 5

Software Restart Markers in VLIW Architectures

This chapter presents an example of the partial usage model of software restart markers within the context of VLIW architectures. I begin by presenting some background on VLIW architectures and discussing the issues with exception handling. I illustrate how software restart markers can be used to support virtual memory when executing software-pipelined loops in a VLIW machine that employs the EQ scheduling model, and I also show how the concept of temporary state can reduce the amount of architectural state in the processor, even in systems that do not use software restart markers.

5.1 VLIW Architecture Background

VLIW architectures are designed to exploit instruction-level parallelism, or ILP, without the significant control logic overhead found in out-of-order superscalar processors. To accomplish this, the compiler extracts independent operations from the program and places them in a single instruction, as shown in Figure 5-1. The advantage of this approach is that the processor does not need complex hardware to detect parallelism; instead, it can simply execute the statically scheduled program produced by the compiler. Also, the ability of the compiler to perform global analysis on a program makes many optimizations possible that cannot be easily accomplished in hardware.

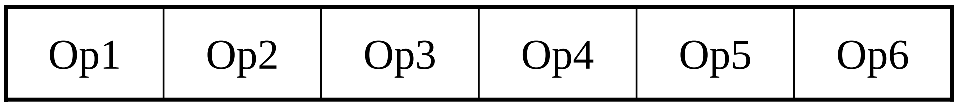


Figure 5-1: A sample VLIW instruction that contains six independent operations.

Implementations of VLIW architectures have appeared in multiple processing domains over the years. Fisher et al. [FFY05] present a historical perspective of various VLIW machines as well as processors that were designed to exploit ILP before the actual term "VLIW" was coined by Fisher in [Fis83]. In the 1980s, the Multiflow Trace [CNO+87, CHJ+90] and Cydrome Cydra 5 [RYYT89, BYA93] used VLIW architectures to try to approach the performance of supercomputers at a substantially reduced cost. Many of the ideas used in those machines were incorporated into later processors, notably Intel's Itanium designs [SA00, MS03], which are examples of the EPIC (explicitly parallel instruction computing) paradigm. Aside from Itanium, VLIW designs are primarily seen in the DSP and embedded processor domains. Examples include the Trimedia processors [ESV+99] from NXP (formerly Philips), the Texas Instruments C6x family of DSPs [Ses98], and the Lx family of embedded cores from Hewlett-Packard Laboratories and STMicroelectronics [FBF+00].

Since current VLIW architectures are largely associated with the embedded processor domain, it is common for these designs to employ an imprecise exception model. In general-purpose systems, it is essentially a requirement to support features such as demand-paged virtual memory and multiprogramming, so designers have been willing to incur the overhead of buffering mechanisms to enforce in-order commit (although as shown in Chapter 2, this overhead is becoming increasingly unacceptable). By contrast, embedded systems often do not need to support demand paging—there may be no secondary storage for swapping—and typically face stricter constraints with respect to area, cost, and power consumption. Even in instances when features such as demand paging are desirable, VLIW processors frequently have such a large number of registers that support for swappable exceptions is prohibitively expensive due to the high context switch overhead. For example, as brought out in Section 2.4.3, exception handling in the Cydra 5 could incur an overhead of several cycles per operation. Thus, the drawbacks of the precise exception model have frequently not been worth the benefits in VLIW architectures.

Using an imprecise exception model introduces the issues discussed in Chapter 2 of having instructions in various stages of completion. For example, consider Figure 5-2, which shows a snippet of code mapped to a VLIW architecture that can bundle three integer operations, a load operation, and a store operation into a single instruction. Assume that both load and multiply operations take two cycles in this architecture. Suppose that the second load instruction causes an exception before the first load completes. In the precise exception model, the hardware would ensure that the first load completes execution and updates architectural state before the exception is handled. In an imprecise exception model, if the exception for the second load is handled immediately and the pipeline is flushed, the first load will be uncompleted. An alternative is to drain the pipeline and allow in-flight instructions to complete, but this could also cause problems if later instructions have issued—e.g. if the exception is detected after the first multiply has begun execution. The hardware-based solutions discussed in Section 2.5 can also be adapted to VLIW architectures. For example, the current-state buffer [OSM+98] is a structure that keeps track of which instructions are uncompleted. Upon resuming after an exception, a process can use this information to avoid re-executing instructions. Unlike the instruction window scheme [TD93], the current-state buffer does not store source operands, and it also does not buffer results. Instead, it relies on compiler scheduling restrictions to ensure that uncompleted instructions have their source operands preserved. While this avoids a substantial amount of hardware overhead, the actual buffer still needs to be saved and restored in the event of an exception, which can be significant in processors that support a large number of in-flight instructions. Replay buffers [Rud97] are designed to avoid compiler restrictions by buffering results, this approach is even more hardware-intensive.

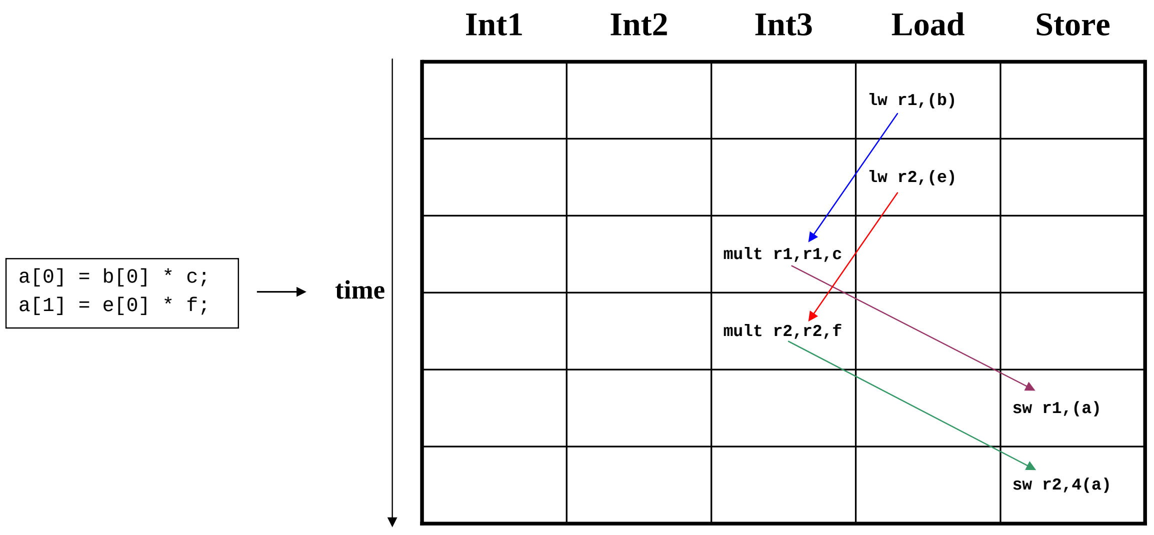


Figure 5-2: A code segment that is mapped to a VLIW architecture that can execute three integer operations, one load operation, and one store operation in parallel. Load and multiply operations have a two-cycle latency. The arrows between operations indicate the flow of data.

Since embedded processors frequently do not support features such as virtual memory, the problems discussed above are somewhat mitigated by the fact that often the primary type of exception which requires process restart in an embedded system is an asynchronous exception, such as a timer interrupt. An asynchronous exception is not correlated with a particular instruction, so it is safe to simply drain the pipeline and restart after the last completed instruction. For example, if a timer interrupt occurs after the first multiply has been issued in Figure 5-2, the processor can simply allow all instructions to complete and then restart after the interrupt from the instruction containing the second multiply operation. While there may also be support for restartable synchronous exceptions in embedded systems—e.g. for trap instructions or for the emulation of floating-point instructions in software—these are typically predictable at compile-time (a compiler knows when it inserts a trap instruction, while a TLB miss is often unpredictable), and the code can be appropriately scheduled so that it is safe to drain the pipeline in the event of an exception.

Although many existing embedded processors do not implement robust exception support, the blurring of the boundary between the embedded and general-purpose domains could lead to increased requirements for features such as demand paging in the future. However, even in systems that do not support virtual memory, there is another problem with asynchronous exception handling that is not illustrated in the example of Figure 5-2. Some compilers take advantage of the exposed pipelines in VLIW architectures and employ certain techniques which rely on specific sequences of instructions to be executed without interruption. To avoid any issue with unpredictable asynchronous exceptions, these designs may disable exception handling during these sections of code. Fisher et al. refer to this as an "unprotected state" [FFY05], and list some of the disadvantages of this approach, including debugging difficulties and complications for real-time applications. They argue that future embedded systems should support precise exceptions in order to have a "clean" programming model. The problem with this perspective is that precise exceptions can lead to unacceptable amounts of overhead from various sources. Due to the difficulties with handling synchronous exceptions, and the problems in some systems with asynchronous exceptions, it is worthwhile to consider the use of software restart markers in VLIW designs.

5.2 Issues with Software Restart Markers in VLIW Architectures

Chapter 4 showed how software restart markers could be used in vector architectures to substantially reduce exception management overhead. VLIW machines share some similarities with vector processors: both types of designs rely heavily on compile-time information to exploit parallelism and simplify hardware complexity; both types of designs excel at accelerating loops; and both types of designs tend to have large amounts of architectural state. At first glance, software restart markers might seem to be even more of a natural fit for VLIW architectures than for vector architectures, as a VLIW compiler's greater amount of control over the execution schedule would appear to mesh well with an exception model that also depends on the compiler. However, there are certain characteristics of VLIW machines which somewhat reduce the benefits of software restart markers seen in Chapter 4. This section discusses those issues.

A notable difference between vector and VLIW processors that affects the applicable software restart markers usage model is the fact that a vector unit is typically a coprocessor, while a VLIW design is often the main processor in the system. A vector machine normally has a scalar control processor that is used to handle non-vectorized code, making it a natural fit for the black box usage model: the control processor can support precise exceptions for scalar code, while all vectorized code is placed within restart regions. As a result, the entire vector register file can be treated as temporary state. It is also possible for a VLIW design to serve as a coprocessor, in which case the black box model could be applied. However, the fact that VLIW architectures exploit ILP—which can be targeted throughout the entire program, albeit with varying degrees of success—and vector architectures exploit DLP—which is primarily targeted in loops within leaf functions—makes it more likely for a VLIW design to be used as a main processor. In these situations, the black box usage model cannot be employed, meaning that the VLIW registers cannot be completely mapped to temporary state.

In a system with a single main processor, that processor is typically responsible for executing the various parts of the program that cannot be easily accelerated, or if there are no coprocessors, the main processor simply executes the entire program. This means that VLIW architectures are often used to execute portions of the program with little or no parallelism, such as control-intensive SPECint-style code. This type of code is likely to contain many non-idempotent updates. Thus, if software restart markers are to serve as the only exception model in a VLIW machine, there are two options that can be considered. One approach is to always try to create large restart regions using the techniques described in Chapter 4, such as copying memory arrays. However, a key reason why those techniques work in vector designs is that the speedup of vector execution often far exceeds any performance degradation from restart region formation. By contrast, when a VLIW architecture executes regions of code with little parallelism, there would probably be little performance gain and significant overhead from inserting software restart markers. A second approach is to simply have small restart regions when executing non-parallel code—so execution becomes primarily sequential and serial—and to use large restart regions where there is ample parallelism. This solution seems reasonable: since a lack of parallelism already keeps the processor from achieving a high level of performance, the overhead of small restart regions should not degrade performance much further. The overhead could be limited in architectures that support software speculation, as the compiler can enlarge restart regions by speculating non-idempotent operations and placing the actual exception checks in separate regions. This technique would be similar to the work with sentinel scheduling [MCH$^+$92], and might make the complete usage model of software restart markers more appealing. However, in VLIW architectures without speculation, the potential performance degradation caused by small restart regions may prove to be unacceptable to many designers, even when executing code that has little inherent parallelism.

Despite the issues with software restart markers in VLIW architectures, there are still benefits to be gained from using this exception handling approach. The next section discusses how the partial usage model of software restart markers can be applied to VLIW designs.

5.3 Handling Exceptions in Software-Pipelined Loops

When a VLIW compiler schedules code, it specifies a maximum latency for each operation, and it is the hardware's responsibility to handle situations when that latency is exceeded.

For example, if the compiler schedules a load operation with the assumption that the value will come from the L1 cache, and there is an L1 cache miss, the hardware has to stall the pipeline until the load value is returned. However, this restriction does not account for the possibility of values being produced sooner than the compiler's assumed latency—e.g. a value predicted to come from the L2 cache might actually be present in the L1 cache. Early values can be an issue depending on whether the compiler uses an *equals* (EQ) or *less-than-or-equals* (LEQ) scheduling model [FFY05]. In the EQ model, a value will not be written early; the hardware will enforce the compiler's specified latency for each operation. In the LEQ model, operations can complete at any time up to the compiler's specified latency. Scheduling under the LEQ model is similar to typical scheduling for a sequential architecture, in that two values with overlapping lifetimes—i.e. the time from a value's definition to its last use—cannot be mapped to the same register. Figure 5-2 is an example of the LEQ model. In that figure, two registers, `r1` and `r2`, are used to pass values between operations. By contrast, Figure 5-3 illustrates how the EQ scheduling model can be used to reduce register pressure. In Figure 5-3, only the `r1` register is used to pass values between operations. The reduction in register usage is possible because each operation writes its result to `r1` exactly at the compiler's specified time: the first load updates `r1` in the cycle that the first multiply is issued, and the multiply obtains the value of `r1` in that cycle; the second load then updates `r1` in the next cycle, and so on. In a sense, the lifetime of a register under the EQ scheduling model actually begins when the value is *produced*, not when the operation that produces the value is issued. This enables more efficient register allocation.

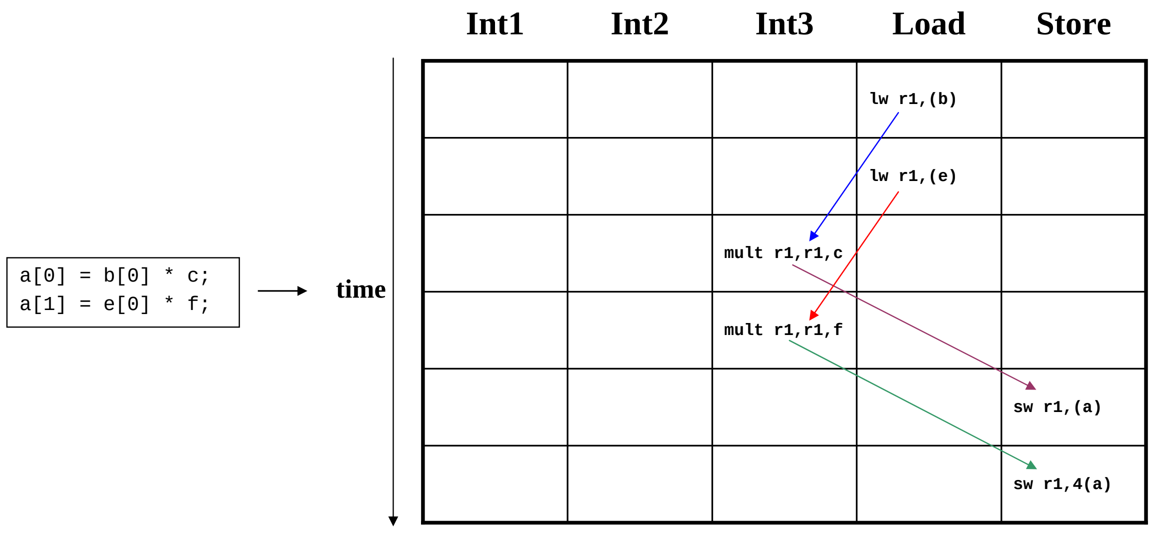
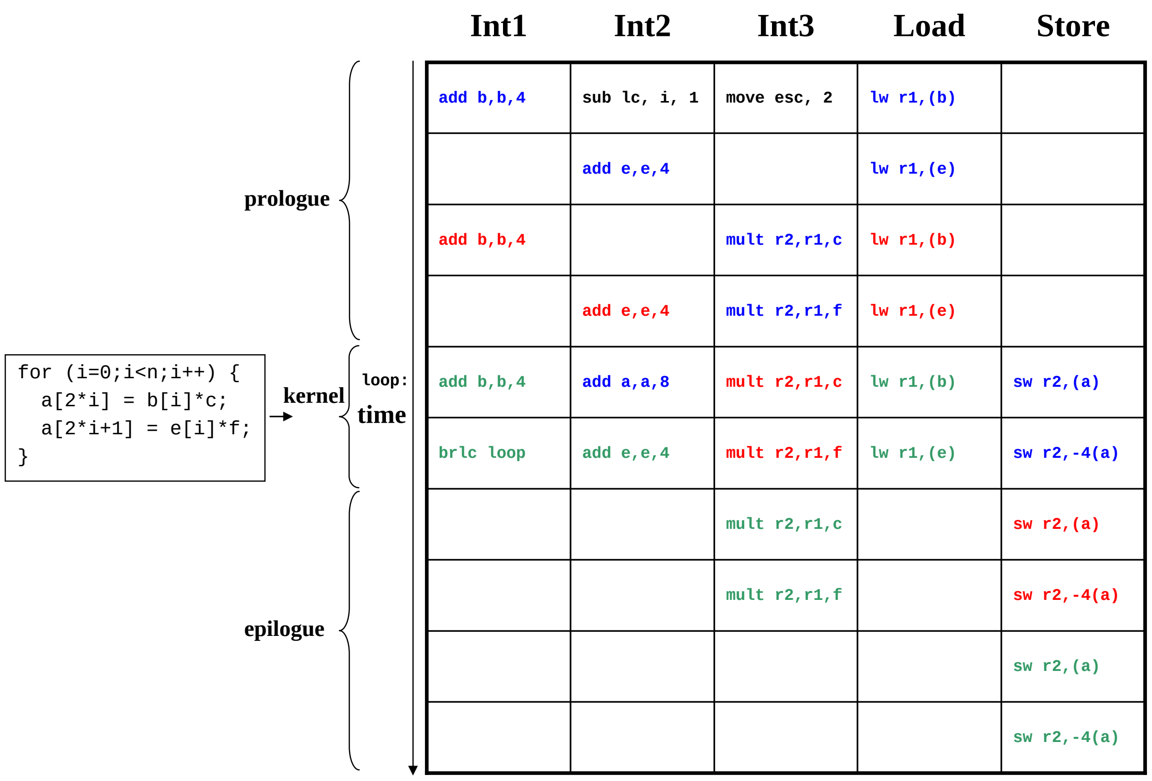


Figure 5-3: A code segment that is mapped to a VLIW architecture with an EQ scheduling model. Load and multiply operations have a two-cycle latency. The arrows between operations indicate the flow of data and also indicate exactly when each result is produced.

However, exception handling is complicated by using the EQ model. Suppose a timer interrupt occurs after the first multiply is issued in Figure 5-3. If instruction issue is halted and all in-flight instructions are allowed to complete, the value that will end up in `r1` when the interrupt is handled will be the result of the first multiply. If the process then resumes execution from the instruction containing the second multiply operation, the value of `r1` will be incorrect: while it should contain the result of the second load, it actually contains the result of the first multiply operation. One approach to resolve this issue is to simply disable all exceptions when processing long-latency operations under the EQ scheduling model. However, this solution can result in long periods of time without exceptions being enabled, particularly when dealing with loops.

A common technique to statically exploit ILP is to *software pipeline* loops [Cha81] so that multiple iterations will have their execution overlapped. There are a variety of methods to implement software pipelining [AJLA95], with modulo scheduling [RG81] probably being the most popular approach. While there has been a great deal of research in the field of software pipelining, and it can be very effective at accelerating loops, the fact that it causes multiple iterations to be in-flight simultaneously is problematic for VLIW architectures that use the EQ scheduling model. Consider the example shown in Figure 5-4. The C code is similar to the previous examples shown in this chapter, except now the statements are placed within a loop. A *kernel* executes the majority of the loop iterations, while a *prologue* is used to set up the software pipeline and an *epilogue* is used to exit the pipeline. The two load operations from a single iteration target the same register; similarly, the two multiply operations from a single iteration also target the same register. The problem of exception handling now lasts throughout the duration of the software-pipelined loop. The result of each instance of a load or multiply operation in a particular kernel iteration is supposed to be used by an operation that will be issued in the subsequent kernel iteration. Thus, if an exception is handled during the kernel, and in-flight instructions are allowed to drain, `r1` and `r2` will have incorrect values when the process resumes.

The Texas Instruments C6x architecture uses the EQ scheduling model, which is referred to as *multiple assignment* [SL99]. It deals with the exception handling problem by simply disabling exceptions during long-latency operations, including during software-pipelined loops. In order to bound the response time for interrupts under this scheme, the compiler has to set up the code so that the software pipeline periodically drains and the system can handle any interrupts. A similar approach devised by Texas Instruments [SS01] improves interrupt response time by setting up special code sections used to drain the pipeline as soon as an interrupt is detected, as opposed to waiting for a predetermined number of iterations to be executed before draining the pipeline and checking for interrupts. This solution effectively reduces the response time for an interrupt to the time required to execute a single loop iteration, but it does not enable support for synchronous exceptions within the loop. Modulo schedule buffers [MH01] are used in an alternative hardware-based approach that reduces interrupt response latency to the time for a single iteration, but again this work



```
for (i=0;i<n;i++) {
  a[2*i] = b[i]*c;
  a[2*i+1] = e[i]*f;
}
```

Int1	Int2	Int3	Load	Store
`add b,b,4`	`sub lc, i, 1`	`move esc, 2`	`lw r1,(b)`	
	`add e,e,4`		`lw r1,(e)`	
`add b,b,4`		`mult r2,r1,c`	`lw r1,(b)`	
	`add e,e,4`	`mult r2,r1,f`	`lw r1,(e)`	
`add b,b,4`	`add a,a,8`	`mult r2,r1,c`	`lw r1,(b)`	`sw r2,(a)`
`brlc loop`	`add e,e,4`	`mult r2,r1,f`	`lw r1,(e)`	`sw r2,-4(a)`
		`mult r2,r1,c`		`sw r2,(a)`
		`mult r2,r1,f`		`sw r2,-4(a)`
				`sw r2,(a)`
				`sw r2,-4(a)`

Figure 5-4: A software-pipelined loop in a VLIW architecture with an EQ scheduling model. Load and multiply operations have a two-cycle latency, while integer ALU operations have a single-cycle latency. Operations corresponding to a particular iteration have the same color. A modified version of the loop support in the Cydra 5 [DHB89] is used in this example. A special loop counter register, `lc`, holds the value of the number of uncompleted iterations. The epilogue stage counter (`esc`) register holds the value of the number of iterations that need to be executed in the epilogue. The `brlc` operation branches to the specified address if the loop counter is greater than the epilogue stage counter `esc`, and it also decrements the loop counter by 1. Assume that `i` is always greater than or equal to 3 in this example.

does not allow synchronous exceptions. Finally, replay buffers [Rud97] do allow synchronous exceptions, but increase hardware complexity.

Software restart markers can be used to enable both asynchronous and synchronous exception handling within a software-pipelined loop under the EQ scheduling model. By simply placing the loop within a restart region, the problems discussed above are avoided. In the event of an exception, there is no issue of performing incorrect register updates, since the loop will be restarted from a “safe” point. The analysis discussed in Chapter 4 can also be used for VLIW architectures to form restart regions. For example, the counted loop optimization can be used with the loop in Figure 5-4. All that is needed is code before the software-pipelined loop to preserve the induction variable values, and a recovery block to restart the software pipeline from the earliest uncompleted loop iteration in the event of an exception. Figure 5-5 shows these two code blocks. The pre-pipeline code that is inserted before the prologue preserves the original values of the induction variables. The fixup code that is branched to after handling an exception will adjust the induction variables by using the value contained in the `lc` register, which decreases from the trip count of the loop to 0. The `brlc` operation at the end of each loop iteration decrements the value of `lc`; thus, it is similar to the `commitbar_loop` instruction from Section 4.4.2.

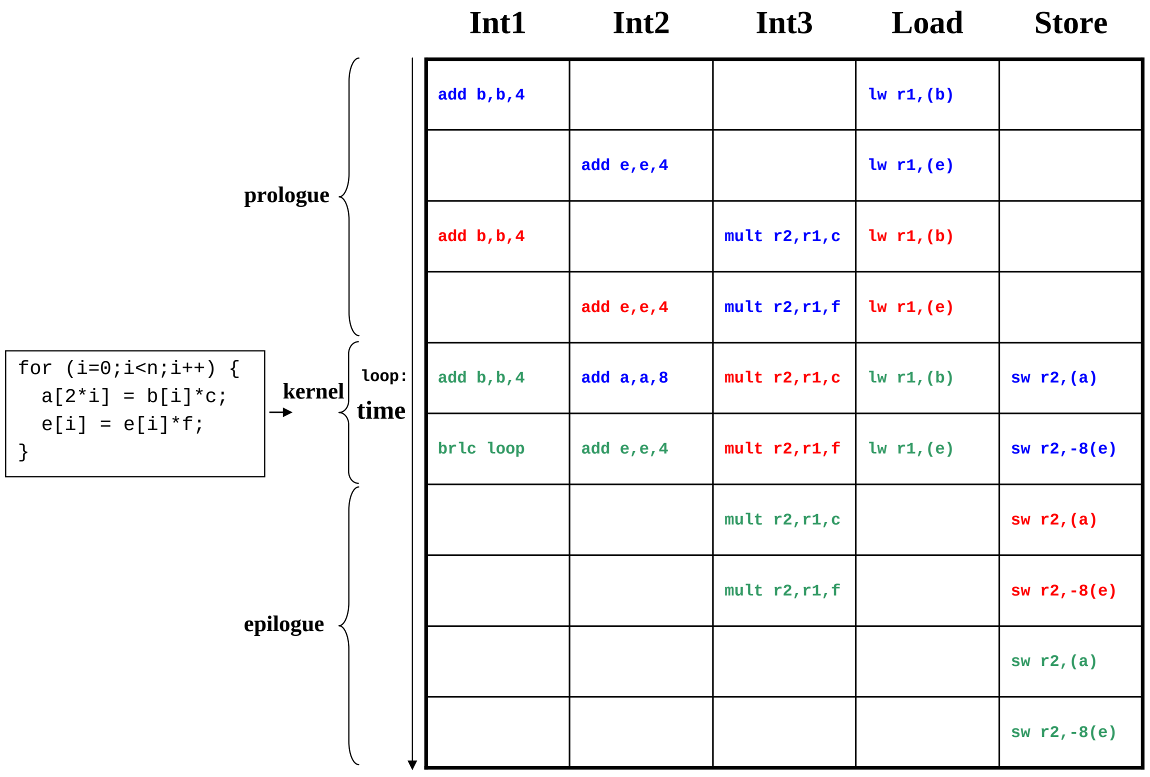
The loop in Figure 5-4 is idempotent, which makes it simple to form restart regions. As stated earlier, if this criterion is not satisfied, then the techniques described in Chapter 4 could be used to make the loop idempotent. However, an interesting option to consider is placing a restart region around a non-idempotent loop without performing any transformations such as copying memory arrays. This idea is possible for two reasons. First, a VLIW instruction contains multiple operations, so the `brlc` operation—which serves as a type of commit barrier—can issue simultaneously with other operations. Thus, the “barrier” instruction can perform other useful work besides simply switching to a new restart region. Second, in the EQ model, the compiler has complete control over the execution schedule, so any non-idempotent updates will occur according to the compiler’s specified latency. These two characteristics make it possible for the compiler to bundle a non-idempotent operation in the same instruction as the commit barrier for a restart region. Having the barrier instruction perform a non-idempotent update is not prohibited by the software restart markers model, as the only requirement for a barrier is that if it completes, the current restart region is ended. In fact, previous work on software restart markers [Ham01, AHKW02] adopted the approach of always attaching a barrier to an existing non-idempotent instruction in the program, rather than inserting a new instruction.

To illustrate how the same technique can be applied to VLIW architectures, consider the loop in Figure 5-6, which is a slightly modified version of the loop in Figure 5-4. In the new example, the second loop statement modifies `e[i]`, which is also used to provide an input value. Despite the fact that this statement makes the loop non-idempotent, the store operation is scheduled in the same cycle as the `brlc` operation for the corresponding

	Int1	Int2	Int3	Load	Store
`pre_pipeline:`	`move r3,a`	`move r4,b`	`move r5,e`		
	`sub lc,i,1`	`move esc, 2`	`sub i, i, 1`		
	begin restart region at prologue	*prologue no longer contains updates to the*	*lc or esc registers*		
	.				
	.				
time	.				
`fixup_code:`	`sub r1,i,lc`				
	`sll r1,r1,2`	`sll r2,r1,3`			
	`add a,r3,r2`	`add b,r4,r1`	`add e,r5,r1`		
	Execute the instructions from the prologue.	*Then if lc is greater than esc, jump to loop;*	*otherwise execute a special code segment that*	*will finish the software pipeline.*	

Figure 5-5: The pre-pipeline and fixup code needed to implement the counted loop optimization for the example of Figure 5-4. The pre-pipeline code makes copies of the induction variables and also relocates the initializations of the `lc` and `esc` registers to before the restart region, in case an exception occurs during the prologue. The value of `i` is also decremented by 1 to ensure the pointers are correctly updated in the fixup code. The fixup code restores the pointer values and restarts the software pipeline. Depending on the value of `lc`, the fixup code will either return to the loop kernel or clean up the software pipeline so that it completes successfully.

iteration, which means that the update of `e[i]` will only occur if the loop counter is also updated. Thus, the non-idempotent update will never be re-executed. Another requirement for correctness is that the store to `e[i]` is guaranteed to execute at some point in the loop. As long as stores are allowed to update memory once they are issued, then even if an exception occurs in the cycle after the `brlc` operation completes, `e[i]` will still receive the correct value. Although it is unlikely that all non-idempotent loops can be handled in this manner, the example shows an alternative method to use software restart markers without copying array values.



```
for (i=0;i<n;i++) {
  a[2*i] = b[i]*c;
  e[i] = e[i]*f;
}
```

Int1	Int2	Int3	Load	Store
add b,b,4			lw r1,(b)	
	add e,e,4		lw r1,(e)	
add b,b,4		mult r2,r1,c	lw r1,(b)	
	add e,e,4	mult r2,r1,f	lw r1,(e)	
add b,b,4	add a,a,8	mult r2,r1,c	lw r1,(b)	sw r2,(a)
brlc loop	add e,e,4	mult r2,r1,f	lw r1,(e)	sw r2,-8(e)
		mult r2,r1,c		sw r2,(a)
		mult r2,r1,f		sw r2,-8(e)
				sw r2,(a)
				sw r2,-8(e)

Figure 5-6: The loop of Figure 5-4 slightly modified to contain a non-idempotent update. The initializations of `lc` and `esc` now occur before the prologue and are not shown.

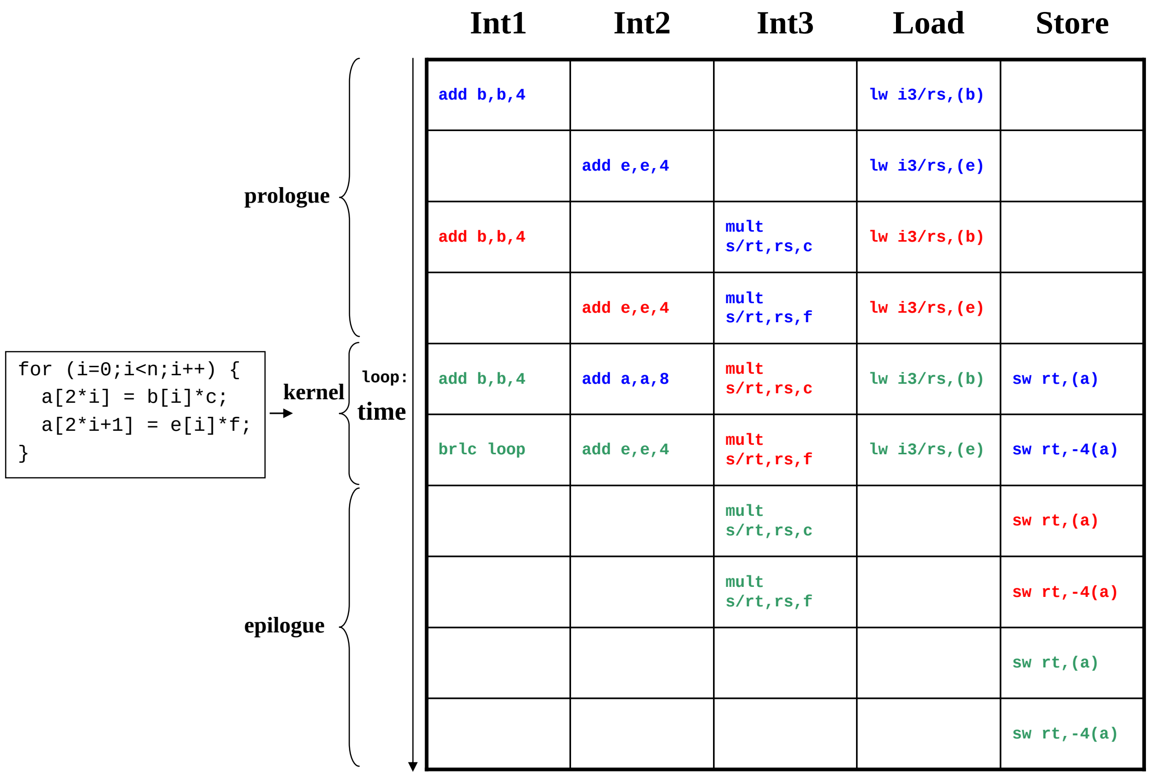
Although in the above examples, software restart markers are only used in part of the code—i.e. during the execution of software-pipelined loops—they provide a substantial benefit by allowing the compiler to reduce register pressure with the EQ scheduling model while still supporting both asynchronous and synchronous exceptions. The next section shows how the benefits can be increased by using temporary state.

5.4 Temporary State in VLIW Architectures

In the example in Figure 5-4, `r1` and `r2` are used to hold temporary values being passed between operations. Since these values are only used once, and restart regions allow them to be recreated in the event of an exception, it would be ideal if the values did not have to be mapped to registers at all. In fact, in an actual pipeline, the values would typically be read from the bypass network. If the architecture exposed the inputs to the ALU, the compiler could explicitly target them for short-lived values, avoiding the need to use the register file. This approach could allow the amount of architectural state in the system to be significantly reduced; for example, a general-purpose or rotating register file in a VLIW design could contain fewer registers without harming performance. Figure 5-7 shows how the values previously mapped to `r1` and `r2` are now mapped to the ALU inputs. While the given example only eliminates the use of two registers, more complex loops could obtain a significant benefit from exposing this temporary state. This technique would probably not require any additional bits for encoding, as there is often wasted space in a VLIW instruction; alternatively, the revised architectural specification could simply use the defunct encodings for the registers which have been removed to represent the ALU inputs. One disadvantage of this scheme is that a register will have to be used for a value which is needed by multiple operations, as the operation that produces the value can only target one ALU input. An alternative approach is to add a small number of accumulators in the pipeline which can be read by multiple operations. The compiler can then map short-lived values to these accumulators without having to preserve their state in the event of an exception.

An interesting point to note is that the concept of temporary state can be used in certain cases even in processors that do not employ the software restart markers model. For example, as stated earlier, existing VLIW designs that use the EQ scheduling model typically do not handle synchronous exceptions during the execution of a software-pipelined loop, and only handle asynchronous exceptions after the software pipeline is drained. This means that in those systems, an exception will only be handled at the boundary between loop iterations; thus, any temporary values which are only used within a single iteration will not be live once the exception is handled. The result of this observation is that it is safe for the compiler to explicitly target ALU inputs as in Figure 5-7, even without the use of software restart markers, as long as the produced values are not live across iterations. Therefore, even if a particular VLIW design does not need to support features such as demand paging, it is worth considering the use of the EQ scheduling model in conjunction with the concept of temporary state to potentially reduce the number of architectural registers in the system.

The idea of targeting ALU inputs or simply avoiding register file accesses has been explored in previous work. Energy-exposed architectures [Ham01, AHKW02] use software restart markers to expose pipeline bypass latches, but this work focuses on reducing energy consumption in a simple in-order pipeline, while the point of this thesis section is to show how the amount of architectural state in a VLIW design can be reduced. Instruction level distributed processing [KS02c] uses accumulators to hold short-lived values, but accumulator



```
for (i=0;i<n;i++) {
  a[2*i] = b[i]*c;
  a[2*i+1] = e[i]*f;
}
```

Int1	Int2	Int3	Load	Store
`add b,b,4`			`lw i3/rs,(b)`	
	`add e,e,4`		`lw i3/rs,(e)`	
`add b,b,4`		`mult s/rt,rs,c`	`lw i3/rs,(b)`	
	`add e,e,4`	`mult s/rt,rs,f`	`lw i3/rs,(e)`	
`add b,b,4`	`add a,a,8`	`mult s/rt,rs,c`	`lw i3/rs,(b)`	`sw rt,(a)`
`brlc loop`	`add e,e,4`	`mult s/rt,rs,f`	`lw i3/rs,(e)`	`sw rt,-4(a)`
		`mult s/rt,rs,c`		`sw rt,(a)`
		`mult s/rt,rs,f`		`sw rt,-4(a)`
				`sw rt,(a)`
				`sw rt,-4(a)`

Figure 5-7: The software-pipelined loop of Figure 5-4 is modified so that temporary values are explicitly mapped to ALU inputs instead of the register file. Similar to the MIPS architecture, `rs` corresponds to the ALU input that holds an instruction's first encoded source operand, while `rt` corresponds to the ALU input that holds an instruction's second encoded source operand. In a `sw` instruction, the store-data register is actually the second encoded source operand, which is why `rt` is used for the stores in the example.

values have to be buffered, while software restart markers enable the targeting of ALU inputs without any buffering. Sami et al. [SSS+02] discuss a design for embedded VLIW architectures in which short-lived values that can be exclusively provided by the bypass network will not be written back to the register file. However, their technique relies on modifying the pipeline logic, and if it is detected that an instruction will cause an exception, all previously issued instructions must write their values to the register file even if those values are short-lived. Static strands [SWL05] are chains of dependent instructions that are created at compile-time, avoiding the need to map temporary values to registers. However, each strand has to be treated as an atomic unit, so a strand cannot contain an instruction which might cause a synchronous exception. Also, since each instruction in a chain is scheduled immediately after the instruction on which it depends, this can lead to delays if there are long-latency instructions in the chain. By contrast, as shown in the example of Figure 5-7, the concept of temporary state can be used while still allowing the compiler to break apart dependence chains during scheduling.

5.5 Summary

This chapter presented a brief overview of VLIW architectures, which are primarily used in embedded systems. Typically, these VLIW machines do not support features such as demand paging and sometimes disable exception handling altogether for certain sections of code. Software restart markers would seem to be a natural fit for VLIW architectures, which rely heavily on the compiler. However, the fact that a VLIW processor is often the main processor in the system makes it more appropriate to employ the partial usage model of software restart markers, in which restart regions are formed around software-pipelined loops. This approach can enable support for both synchronous and asynchronous exceptions even when using the EQ scheduling model. Additionally, the notion of temporary state can be used to expose the inputs to the ALU, reducing the amount of architectural state needed to achieve a given level of performance.

Chapter 6

Software Restart Markers in Multithreaded Architectures

This chapter shows how the complete usage model of software restart markers can be applied to multithreaded architectures. I first present a brief overview of multithreading, and then illustrate how the concept of temporary state can be used to support a greater number of active threads.

6.1 Multithreaded Architecture Background

A multithreaded architecture exploits thread-level parallelism by enabling multiple threads to be active simultaneously. This can be accomplished by allowing more than one thread to execute in the same cycle using independent resources (i.e. multiple cores or multiple pipelines within a core), or by interleaving the execution of threads in different cycles on the same pipeline. Multithreading can help to alleviate the performance impact of stalls caused by long-latency instructions. A potential drawback of interleaving multiple threads is that the execution time for a single thread may be lengthened; this is a case of favoring throughput over per-thread latency.

There are several different types of designs for multithreaded architectures. *Fine-grained multithreading* can switch to a new thread after each instruction; examples of this approach include the Denelcor HEP [Smi81], the Horizon processor [TS88], and the Tera Multithreaded Architecture [ACC$^+$90]. *Coarse-grained multithreading* executes a thread until some long-latency event occurs, such as an L2 cache miss; the MIT Alewife machine [ABC$^+$95] and the IBM PowerPC RS64 IV processor [BEKK00] use this approach. *Simultaneous multithreading* (SMT) [TEL95] allows instructions from multiple threads to issue in the same cycle; this technique is incorporated in Intel's Pentium 4 (where it is referred to as *Hyper-Threading*) [MBH$^+$02] and IBM's POWER5 [SKT$^+$05]. All of these forms of multithreading are surveyed in greater detail by Ungerer, Robič and Šilc [URv03], who label them as types of *explicit multithreading*, where each thread is user-defined. The authors contrast this category with *implicit multithreading*, in which a processor executes speculative threads generated from a single program. Proposals that use thread-level speculation include single-program speculative multithreading [DOOB95], multiscalar processors [SBV95], and the superthreaded processor architecture [THA$^+$99].

As mentioned earlier, multithreaded architectures often focus more on increasing throughput rather than reducing per-thread latency. If there are a sufficient number of active threads in the system to utilize the processor resources, employing a complex out-of-order superscalar design in an attempt to improve ILP might not be worth the associated drawbacks, particularly for certain application domains with little ILP, such as server programs. For example, Sun's Niagara processor [KAO05], which is designed for server systems, contains eight single-issue in-order pipelines, each of which supports four threads. Even for simultaneous multithreading, which is typically associated with complex superscalars, previous work has shown that using in-order pipelines in a system with four threads can approach the performance of using out-of-order execution [HS99]. In general, in-order execution for multithreaded processors is more energy-efficient than out-of-order execution while often sacrificing little in terms of performance.

Chapter 2 showed that implementing precise exceptions is straightforward in simple in-order pipelines, and does not require excessive buffering of values. Thus, there would seem to be little opportunity to apply software restart markers to multithreading architectures: in-order execution is often the preferred approach in multithreading processors, simplifying precise exception support; and even multithreaded processors that execute instructions out-of-order likely have existing structures such as reorder buffers which can also be used to implement precise exceptions. However, one characteristic common to all multithreaded architectures is that they have a much greater amount of architectural state than a typical sequential architecture. Each thread needs its own user state, including general-purpose registers, as shown in Figure 6-1. A system that supports a large number of threads may enable greater throughput, but this advantage comes at the cost of the area required for the threads' state. Additionally, the amount of individual state for each thread can have a significant effect on context switch overhead depending on the type of multithreading design that is used. These issues point to the idea of using temporary state to reduce the per-thread resource requirements.

6.2 Temporary State in Multithreaded Architectures

Just as in Section 5.4, temporary state can be used for short-lived values instead of architectural registers. Consider the C code statements in Figure 6-2(a), which are mapped to a single thread in a multithreaded architecture (which could use coarse-grained, simultaneous, or speculative multithreading). Figure 6-2(b) shows the assembly code for these statements,

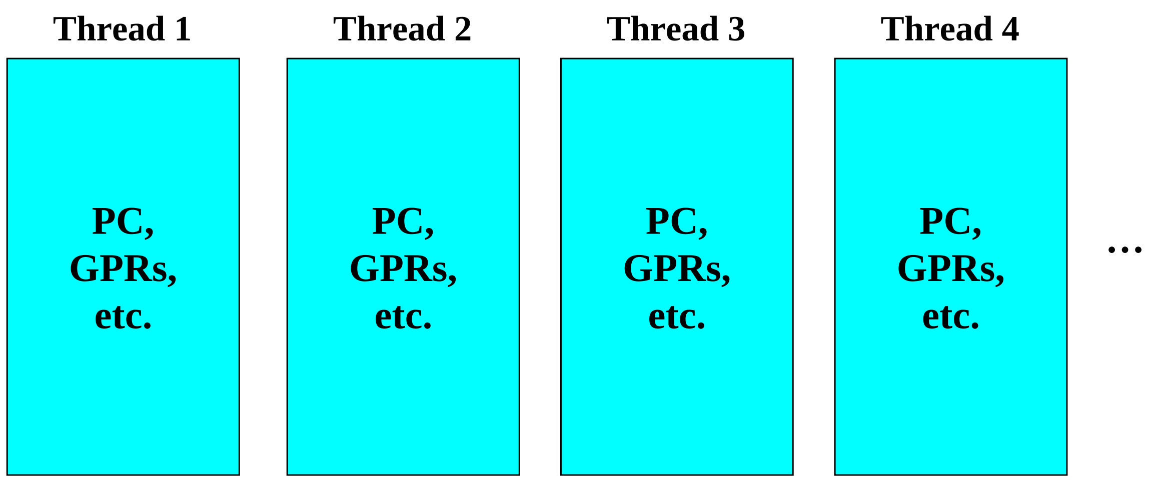
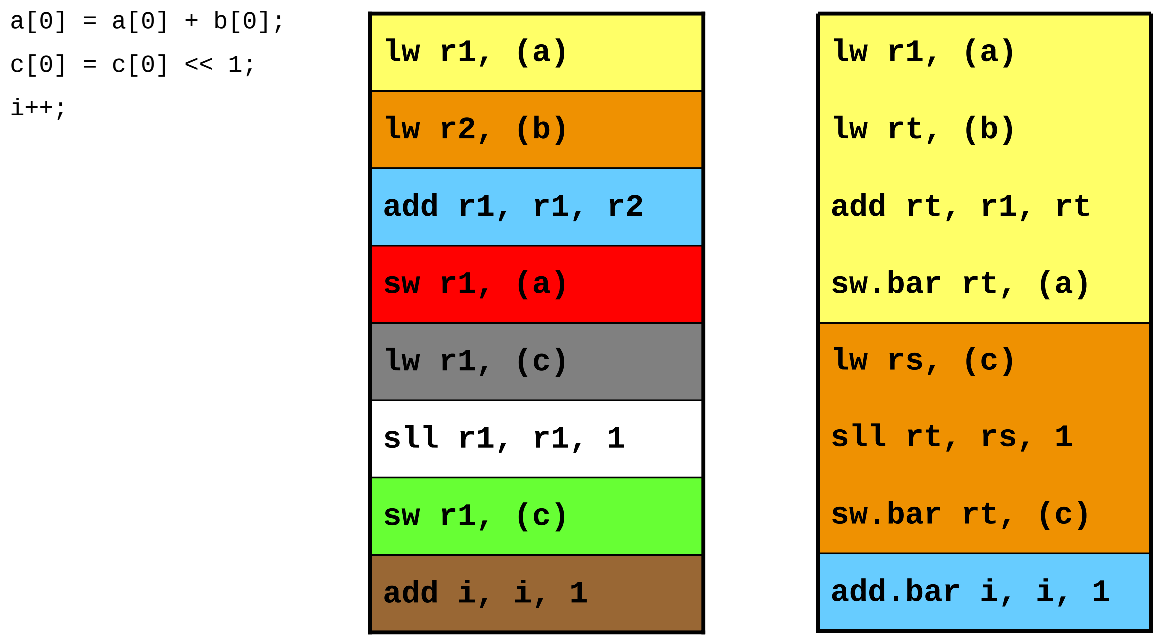


Figure 6-1: In a multithreaded design, each thread has its own architectural state.

compiled for a single-issue in-order pipeline that implements the precise exception model. Figure 6-2(c) shows the assembly code for the same segment, but compiled for a system that uses software restart markers. As in Section 5.4, ALU inputs are explicitly targeted by the compiler in this example, avoiding the use of the `r2` register.

The example of temporary state for VLIW architectures focused on exposing the ALU inputs to reduce the number of architectural registers required to achieve a certain level of single-thread performance, potentially enabling a reduction in the size of the register file. Since multithreaded architectures focus more on improving throughput rather than decreasing per-thread latency, a more natural application of temporary state to multithreaded designs is to use the architectural register reduction to increase the number of threads in a system with a fixed number of physical registers. Suppose that the code segment from Figure 6-2(a) was a single iteration from the loop shown in Figure 6-3(a). If this loop is mapped to a multithreaded architecture that enables a loop to be parallelized by having each thread execute an iteration, the resulting code might look like Figure 6-3(b). Multiple threads are spawned so that iterations can execute in parallel. The induction variable `i` can then be updated at the join point (not shown in the figure). Figure 6-3(b) shows 16 simultaneously active threads. If the threads share a set of physical registers, then reducing the per-thread physical register requirements can allow more threads to be spawned. For example, suppose there are 64 total physical registers in the system, with 32 of those registers restricted to hold global shared values such as constants or pointers, and the other 32 registers usable by individual threads to hold private values. The total number of simultaneously active threads is then determined by the number of private registers required by each thread. In Figure 6-3(b), the `a`, `b`, and `c` pointers will be mapped to shared registers, while the values held in `r1` and `r2` are private to each thread. Thus, each thread requires 2 private registers, which limits the total number of threads to 16. When temporary state is used in Figure 6-3(c), each thread only needs a single private register, enabling 32 threads



```
a[0] = a[0] + b[0];
c[0] = c[0] << 1;
i++;
```

(a)

```
lw r1, (a)
lw r2, (b)
add r1, r1, r2
sw r1, (a)
lw r1, (c)
sll r1, r1, 1
sw r1, (c)
add i, i, 1
```

(b)

```
lw r1, (a)
lw rt, (b)
add rt, r1, rt
sw.bar rt, (a)

lw rs, (c)
sll rt, rs, 1
sw.bar rt, (c)

add.bar i, i, 1
```

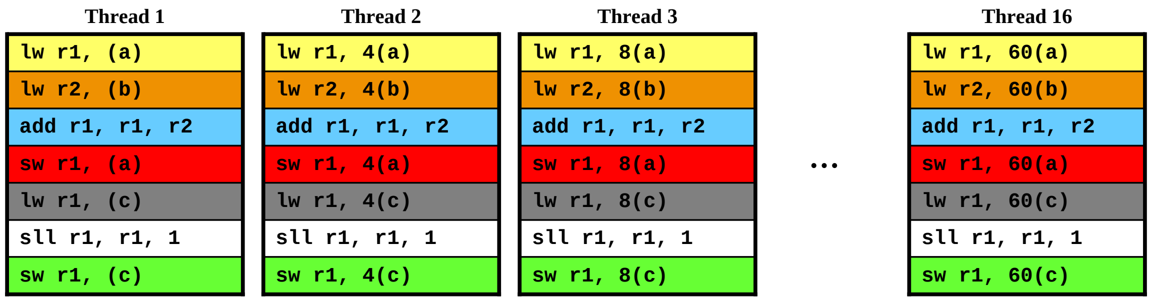
(c)

Figure 6-2: (a) A C code fragment that is to be compiled for a single-issue in-order MIPS processor. (b) Assembly code for the fragment when using the precise exception model. (c) Assembly code for the fragment when using software restart markers. Short-lived values are explicitly mapped to the ALU inputs `rs` and `rt`. The last instruction in each restart region serves as a commit barrier.

to be active simultaneously. Even if the processor has fewer than 32 pipelines and threads have to share execution resources, a greater number of threads can help to hide stalls due to events such as cache misses.

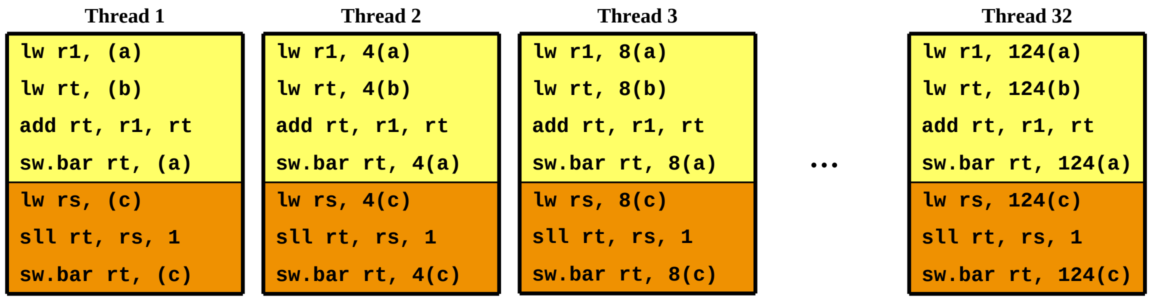
```
for (i = 0; i < n; i++) {
  a[i] = a[i] + b[i];
  c[i] = c[i] << 1;
}
```

(a)



```
Thread 1          Thread 2          Thread 3               Thread 16
lw r1, (a)        lw r1, 4(a)       lw r1, 8(a)            lw r1, 60(a)
lw r2, (b)        lw r2, 4(b)       lw r2, 8(b)            lw r2, 60(b)
add r1, r1, r2    add r1, r1, r2    add r1, r1, r2         add r1, r1, r2
sw r1, (a)        sw r1, 4(a)       sw r1, 8(a)      ...   sw r1, 60(a)
lw r1, (c)        lw r1, 4(c)       lw r1, 8(c)            lw r1, 60(c)
sll r1, r1, 1     sll r1, r1, 1     sll r1, r1, 1          sll r1, r1, 1
sw r1, (c)        sw r1, 4(c)       sw r1, 8(c)            sw r1, 60(c)
```

(b)



```
Thread 1          Thread 2          Thread 3               Thread 32
lw r1, (a)        lw r1, 4(a)       lw r1, 8(a)            lw r1, 124(a)
lw rt, (b)        lw rt, 4(b)       lw rt, 8(b)            lw rt, 124(b)
add rt, r1, rt    add rt, r1, rt    add rt, r1, rt         add rt, r1, rt
sw.bar rt, (a)    sw.bar rt, 4(a)   sw.bar rt, 8(a)  ...   sw.bar rt, 124(a)
lw rs, (c)        lw rs, 4(c)       lw rs, 8(c)            lw rs, 124(c)
sll rt, rs, 1     sll rt, rs, 1     sll rt, rs, 1          sll rt, rs, 1
sw.bar rt, (c)    sw.bar rt, 4(c)   sw.bar rt, 8(c)        sw.bar rt, 124(c)
```

(c)

Figure 6-3: (a) A C loop that is mapped to a multithreaded architecture, with each thread executing an iteration. (b) Assembly code for each thread under the precise exception model. Each thread requires two private registers, `r1` and `r2`. (c) Assembly code for each thread under the software restart markers model. Each thread only requires one private register, enabling a greater number of threads.

Note that the combination of restart regions and multithreading can create an interaction with the L2 cache design and replacement policy when using coarse-grained multithreading. To guarantee forward progress, there needs to be some mechanism to ensure that all of the cache lines accessed by a thread in a particular region can coexist in the L2 cache simultaneously (i.e. they map to different sets or ways within a set), and also to prevent other threads' memory accesses from causing those lines to continually be evicted. Another point of interest is that the complete usage model of software restart markers is adopted throughout the program, so many of the restart regions may be small. This issue is not a concern for single-issue in-order pipelines, as it would not be possible to exploit ILP even in larger regions. The use of small restart regions also limits the re-execution overhead in the event of an exception, although since simple pipelines do not employ techniques seen in superscalar or vector designs to hide memory access latency, cache misses would probably have a far greater effect on performance than the need to occasionally re-execute a few instructions. The complete usage model discussed here is the same as the one described in the work on energy-exposed architectures [Ham01, AHKW02].

The idea of using variable-size hardware contexts for each thread has been explored in other work. Register relocation [WW93] allows a physical register file to be flexibly partitioned into variable-size contexts. The compiler allocates registers for each thread and provides information to the processor about the total number of registers used by each thread; a special hardware register holds a relocation mask which then provides each thread's offset into the physical register file. The named-state register file [ND95] is an associative register file that serves as a cache for thread contexts. Live variables are brought into the register file as needed, and can be placed anywhere within the structure. Thus, threads with large numbers of active variables will be allocated a greater fraction of the physical register resources than threads with only a few live variables. Mini-threads [REL03] are used to increase the number of supported threads in an SMT design without increasing the size of the physical register file. Each hardware context can be partitioned among multiple mini-threads—e.g. if a regular context has 32 architectural registers, then 10 architectural registers can be allocated to one mini-thread while 22 architectural registers can be allocated to a second mini-thread with greater register pressure. The authors' initial evaluation of the technique restricts the partitioning so that each context is statically partitioned in half between two-mini-threads, but this is not a general requirement. The use of mini-threads can potentially increase per-thread latency due to the fewer registers, but can also increase thread-level parallelism. Rodrigues et al. [RMKU04] also discuss the sharing of a register context between very fine grained threads, or "threadlets." Software restart markers can be used in conjunction with the above techniques to further reduce the per-thread register requirements.

It is also interesting to consider the potential application of temporary state to multithreaded architectures with fixed-size contexts for each thread. As with a VLIW architecture, a possible approach is to try to decrease the total number of registers in the system by reducing the per-thread architectural register usage (and maintaining a constant number of threads). However, the actual magnitude of this reduction is questionable. Section 5.4 targets a VLIW design, in which there may be multiple pipelines and the compiler has complete control over the execution schedule, so even long-latency operations can target ALU inputs. By contrast, this section considers a single pipeline for each thread in which each instruction will overwrite the ALU inputs as it enters the execution stage (although *read caching* [Ham01, AHKW02] allows an instruction to avoid overwriting an ALU input if the previous instruction shares the same source operand). Thus, only short-lived values that are part of an unbroken dependence chain will be mapped to temporary state. As shown in Figure 6-2(b), even under the precise exception model the register allocator can simply reuse the same architectural registers for short-lived values. If the default configuration with the precise exception model uses 32 architectural registers for each thread, then the explicit targeting of ALU inputs might eliminate the need for one or two of those registers without requiring the remaining registers to be re-allocated. Additionally, any benefit is reduced by the fact that each thread requires a restart PC, although the use of registers to hold restart addresses can be avoided by storing the addresses in memory. Thus, using software restart markers in multithreaded architectures with fixed-size contexts may not provide any significant advantage over the precise exception model.

6.3 Summary

This chapter considered the issue of exception handling in multithreaded architectures. It is often more cost-effective for a multithreaded design to use simple pipelines and focus on exploiting thread-level parallelism rather than using the complex ILP-enhancing out-of-order execution techniques found in modern superscalar processors. Simple in-order pipelines make it straightforward to implement precise exceptions; however, they do not address the issue of the number of registers required by each thread. Software restart markers can be used to map short-lived values to temporary state, which can enable a greater number of active threads in the system.

Chapter 7

Software Restart Marker Compiler Implementation for Vector-Thread Architectures

As stated in the introduction, this thesis was motivated by a desire to create a low-overhead exception model for explicitly parallel architectures, and software restart markers are the basis of that model. Chapters 4, 5, and 6 discussed conceptual software restart marker designs for vector, VLIW, and multithreaded architectures. In the following two chapters, I present an actual implementation of software restart markers within the context of *vector-thread (VT) architectures* [KBH+04a, KBH+04b], specifically using the Scale VT architecture as a target platform. Scale simultaneously exploits multiple forms of parallelism, enabling it to target a wide variety of applications. It incorporates several features that are used to achieve a high level of performance in an energy-efficient manner. However, these same features also make it difficult to implement conventional exception handling approaches. Scale supports a large number of in-flight operations, contains a significant amount of architectural state, and exposes many pipeline details to the compiler. All of these characteristics introduce a substantial amount of overhead for the exception handling mechanisms discussed in Chapter 2. By using software restart markers as a low-overhead exception model, it becomes possible to retain the advantages of Scale with respect to performance, energy consumption, and area while still enabling support for features such as demand-paged virtual memory.

This chapter focuses on the compiler implementation of software restart markers for the Scale design. I first present an overview of the general vector-thread architectural paradigm as well as the Scale vector-thread architecture, and then I discuss the characteristics of Scale that make precise exception support impractical. Next, I describe the compiler analysis used to form restart regions for vector-style Scale code. Finally, I discuss how software restart markers can be used with threaded code in Scale.

7.1 Vector-Thread Architecture Background

Vector-thread architectures unify the vector and multithreaded execution models. This paradigm enables the efficient exploitation of multiple forms of parallelism within a single design. In this section, I present a brief overview of the vector-thread architectural paradigm and the Scale VT architecture.

7.1.1 Vector-Thread Architectural Paradigm

In the VT architectural model, a control processor interacts with a vector of virtual processors. Each virtual processor (VP) is a thread that contains a set of registers and execution resources. VPs execute RISC-like instructions that are grouped into atomic instruction blocks (AIBs). The control processor can *vector-fetch* an AIB that will be executed by all of the VPs in parallel, thus exploiting data-level parallelism (DLP). The control processor can also execute *vector-load* and *vector-store* commands to transfer blocks of data between VPs and memory. Each VP can also direct its own control flow with a *thread-fetch* of the next AIB, thus enabling thread-level parallelism (TLP). A VP thread halts and returns control to the control processor after executing an AIB that does not issue a thread-fetch instruction. The execution of vector-fetched AIBs and vector memory commands can be freely intermingled with the execution of thread-fetched AIBs, allowing DLP and TLP to be exploited simultaneously within the architecture. The VPs are also connected by a unidirectional ring, the *cross-VP network*, which allows each VP to send values to its neighbor.

7.1.2 The Scale Vector-Thread Architecture

The Scale VT architecture is the first instantiation of the vector-thread architectural paradigm. The Scale processor prototype [Kra07] is a low-power, high-performance design for embedded systems, and it demonstrates that the VT paradigm is well-suited for this processor domain. Figure 7-1(a) is a simplified high-level diagram of the Scale microarchitecture. Scale contains a scalar MIPS-based control processor and a vector-thread unit (VTU) with four parallel lanes. Virtual processors are striped across the physical lanes and are time-multiplexed within each lane to share the physical execution resources. The example configuration presented in Figure 7-1(a) has a vector length of 16, but Scale can support up to 128 VPs. A special vector memory unit (VMU) handles vector-load and vector-store commands.

Figure 7-1(b) is a more detailed view of one lane, showing how it is actually partitioned into four heterogeneous clusters to support instruction-level parallelism (ILP). All clusters can handle standard integer ALU instructions. Additionally, cluster 0 supports VP memory operations, cluster 1 supports VP fetch instructions, and cluster 3 supports integer multiply and divide. Clustering is exposed to the compiler, which is responsible for partitioning VP instructions between the clusters. Scale makes extensive use of decoupling to tolerate latencies, so different clusters within a lane can be executing code for different VPs simultaneously. There are four separate cross-VP networks in Scale, connecting sibling clusters in different lanes, and these networks contain queues to provide decoupling between neighboring VPs.

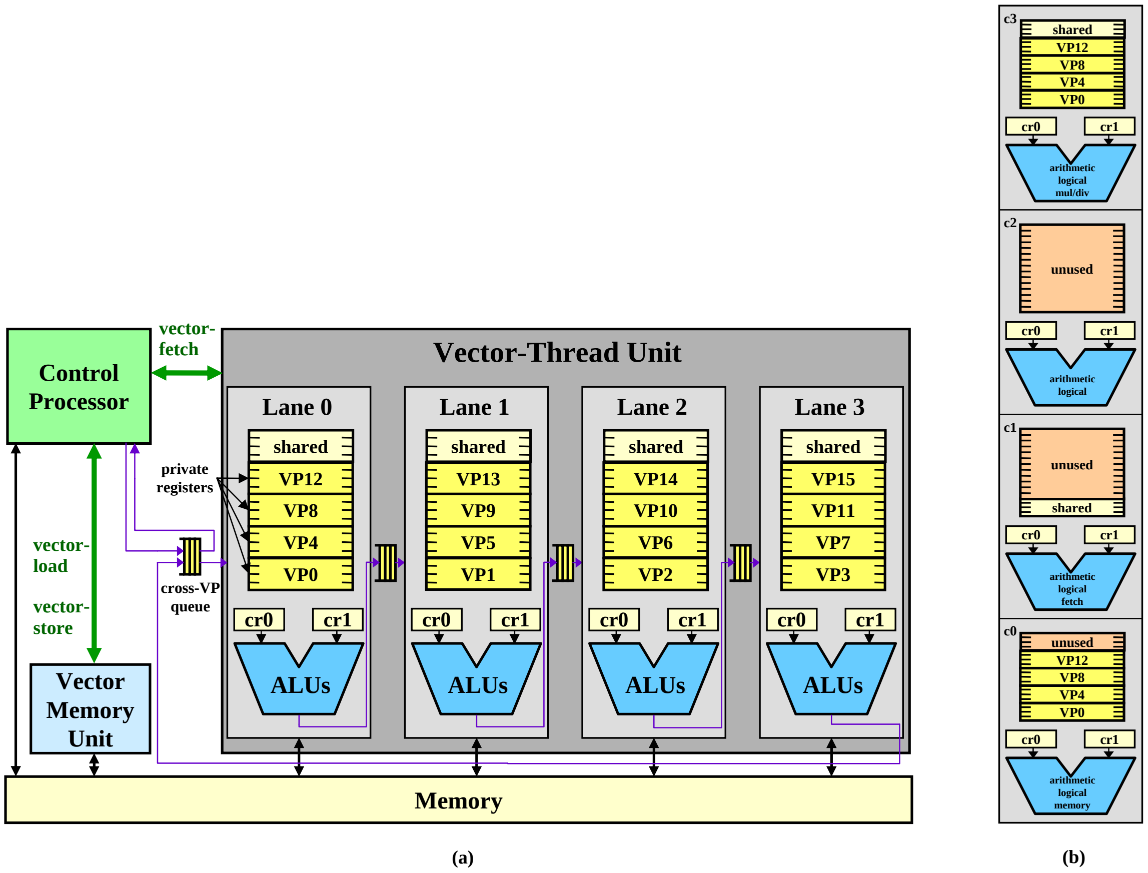


Figure 7-1: (a) High-level overview of Scale microarchitecture. For simplicity, certain details are omitted, such as the use of clustering. (b) Expanded diagram of the clusters within a single lane.

A VP's general registers in each cluster are either *private*—if they are only used by that VP—or *shared*—if they are used by other VPs. The shared registers can be used to hold constants set by the control processor, or to hold temporary values within the execution of a single AIB (the execution of each AIB is atomic with respect to other AIBs, but VPs can interleave execution at AIB boundaries), or to implement efficient reduction operations across VPs. Although not shown in Figure 7-1(b), cluster 0 also contains a separate set of *store-data* registers which are similar to private registers but are only used for holding values to be stored to memory. Additionally, each ALU's input operands are exposed as programmer-visible *chain registers*, which are used to avoid accessing the register files for short-lived values to reduce energy. The use of chain registers and shared registers also reduces the per-VP register requirements. For example, as shown in Figure 7-1(b), cluster

2 does not use the physical register file; this could be due to the fact that instructions on that cluster only use chain registers for their computations. By contrast, cluster 3 uses all available physical registers. The maximum length of the virtual processor vector is dependent on the cluster with the highest per-VP physical register requirements, as discussed in the next section. In Figure 7-1(b), cluster 3 determines the maximum vector length. An important Scale code optimization is to try to use the chain registers and shared registers in each cluster to reduce private register usage, hence allowing increased vector length and potentially greater performance.

7.1.3 Scale Code Example

The typical programming model for a VT architecture is to map loops to the virtual processor vector, with each VP executing an iteration. Figure 7-2 shows a simple vectorizable loop, in which it is assumed that the input and output arrays are guaranteed to be disjoint. The Scale code for this function is shown in Figure 7-3. The code contains both control processor instructions as well as VP instructions, which are delimited by `.aib begin`/`.aib end` directives. The control processor is responsible for issuing a `vcfgvl` configuration command to specify the number of private and shared registers used in each cluster, which determines the maximum vector length, `vlmax`. Note that for the purposes of the configuration, the number of private registers listed for cluster 0—which also supports memory operations—is the larger of the number of private registers used and the number of store-data registers used. To determine `vlmax`, first the following expression is computed for each cluster: $\lfloor (32 - \#\textit{shared regs}) / \#\textit{private regs} \rfloor$, where 32 is the number of physical registers in each Scale cluster. The minimum value of that expression across all clusters (which is the number of VPs supported in a single lane) is then multiplied by the number of lanes to obtain `vlmax`. In Figure 7-3, the register requirements of cluster 3 determine `vlmax`: $(\lfloor (32 - 1)/1 \rfloor) \times 4 = 124$. If a configuration command does not use any private registers, the maximum vector length is set to 128.

The `vcfgvl` command in the example causes the active vector length, `vl`, to be written to the `t0` register. The active vector length is the minimum of `vlmax` and the value of `a0`, which holds the number of loop iterations to be executed. Since the multiply coefficient is a constant value, the control processor writes it into a shared register on cluster 3. It then uses strip mining to launch multiple loop iterations simultaneously. In each strip-mined loop iteration, the control processor sets the vector length, and then performs two vector loads (with an auto increment addressing mode) to obtain the inputs. The actual computation is performed by vector-fetching the AIB, causing the VPs to execute the multiply-add sequence on the input elements. Each VP places its result in a store-data register, which is used by the auto-incrementing vector store. The strip-mined loop continues until all elements have been processed.

Note that the given code example is very similar to what might be generated for a

```
void mult_add(int len, int *in1, int *in2, int *out) {
  int i;
  for (i = 0; i < len; i++)
    out[i] = COEFF*in1[i] + in2[i];
}
```

Figure 7-2: C code for mult_add function.

```
mult_add: #a0=len, a1=in1, a2=in2, a3=out
  # vcfgvl command below configures the number of private and
  # shared registers for each cluster to determine vlmax; the
  # command also sets both the active vector length (vl) and
  # the t0 register to the minimum of a0 and vlmax
  # configuration format: c0:p,s c1:p,s c2:p,s c3:p,s
  vcfgvl  t0, a0,            1,0,   0,0,   1,0,   1,1
  sll     t0, t0, 2         # input/output stride
  vwrsh   COEFF, c3/sr0     # write constant to shared reg
stripmineloop:
  setvl   t1, a0            # (vl, t1) = min(a0, vlmax)
  vlwai   a1, t0, c3/pr0    # vector-load in1, inc ptr
  vlwai   a2, t0, c2/pr0    # vector-load in2, inc ptr
  vf      scale_add_aib     # vector-fetch AIB
  vswai   a3, t0, c0/sd0    # vector-store out, inc ptr
  subu    a0, t1            # decrement counter
  bnez    a0, stripmineloop # loop until done
  vsync                     # allow VPs to finish
  jr ra                     # return

scale_add_aib:
  .aib begin
  c3      mult.lo pr0, sr0      -> c2/cr0   # mult in1 by COEFF
  c2      addu    cr0, pr0      -> c0/sd0   # add to in2
  .aib end
```

Figure 7-3: Scale code for mult_add function.

traditional vector architecture. A key difference is that VP instructions are grouped within an AIB that the control processor issues in one vector-fetch instruction, whereas a traditional vector machine would issue separate vector-multiply and vector-add instructions. The use of AIBs improves locality over conventional vector designs. Additional architectural features not illustrated by the code example—such as the ability for each VP to fetch its own instructions—enable the compiler to map certain types of loops to Scale that would be difficult to handle in standard vectorizing compilers. Appendix A presents more detail about how the compiler targets the various features of Scale. However, the similarity of the `mult_add` example to conventional vectorized code is sufficient to point to the idea that the black box usage model of software restart markers could be adapted to Scale. Before examining the implementation of software restart markers in Scale, I first consider the feasibility of a precise exception approach for this architecture.

7.2 Issues with Precise Exception Handling in Scale

Chapter 4 illustrated the fact that precise exceptions are costly to implement in traditional vector architectures because of the buffering required for in-flight instructions as well as the substantial context switch overhead. This section demonstrates the fact that precise exception support would be even more expensive in Scale due to the features of the architecture. One problem with employing the precise exception model in Scale is the use of AIBs. AIBs improve locality, but they also make it more difficult to implement exceptions precisely. The straightforward approach for precise exceptions would be to enforce the requirement that if an exception occurs within an AIB, none of the instructions within that AIB will have updated architectural state when the exception is handled. This model is somewhat similar to handling exceptions precisely at the instruction level in a vector architecture rather than the operation level; however, a key difference is that an AIB can contain many independent operations. A single Scale vector-fetch VTU command might correspond to over 100 vector instructions in a conventional vector machine. The potentially large number of operations in an AIB complicates the precise handling of synchronous VP exceptions—e.g. debugging a single operation within an AIB would be difficult. An additional problem is the ability of VPs to fetch their own AIBs, meaning that a vector-fetch instruction could trigger a chain of thread-fetches that might differ from VP to VP, possibly causing thousands of VP instructions to be spawned by a single VTU command, which would severely strain a precise exception implementation.

The fact that Scale makes extensive use of decoupled execution only compounds the problem. The control processor will run ahead of the VTU, queuing up multiple vector-fetch and vector-memory commands. Since the VTU clusters are decoupled from one another, VP instructions from different AIBs can execute simultaneously. A single vector-fetched AIB can potentially write to every physical register in the VTU. A precise exception approach such as the reorder buffer scheme would need to buffer all of those results, essentially requiring a copy of the physical register file for every in-flight AIB. In the Scale processor prototype, which only contains architectural registers and no additional physical registers, 26% of the VTU area is occupied by register files [Kra07]. Although the chip uses standard cells for the register files, and custom layout would reduce the area, a precise exception implementation would still be very costly. Supporting a single in-flight AIB precisely would require a doubling of the register file area; decoupled execution results in many in-flight AIBs, incurring an impractical amount of area overhead. Even if an approach such as checkpointing is used so that a copy of the register file is not required for each AIB, multiple copies would still be needed and would be unreasonably expensive to implement.

Given the above problems with handling exceptions at the granularity of an AIB, an alternative approach would be to treat each VP as an independent processor in the system and to handle exceptions at the granularity of an individual VP instruction within an AIB. The exception handler would thus need the ability to restart a VP from an instruction inside an AIB instead of at an AIB boundary. However, there is no existing method in Scale to jump into the middle of an AIB; in fact, there is not even a notion of a program counter associated with VP execution, which would make it difficult to determine which instruction in an AIB caused an exception. Even if it was possible to pinpoint a faulting instruction within an AIB and then restart from it after handling an exception, this approach conflicts with the vector-thread programming model that an AIB should be executed atomically. Another complication is Scale's use of chain registers: since a VP instruction might read or write one or more ALU inputs, handling exceptions at the granularity of a VP instruction means that the chain register contents have to be preserved across an exception. Adding access paths to save and restore the chain registers would increase the complexity of the design.

The problem with preserving chain registers points to another issue with precise exceptions in Scale: the use of cross-VP communication. Since the cross-VP network is used to handle loop-carried dependences, the typical usage for cross-VP transfers is to have the control processor inject initial pre-loop values before entering the loop and retrieve the final post-loop values after exiting the loop. During the actual execution of the loop, the VPs will pass values to their neighbors using the cross-VP queues between the lanes, without any intervention from the control processor. However, a precise exception implementation would have to buffer the values as they are propagated between VPs, and in the event of an exception some method of restoring the cross-VP queues to a precise state would be needed. The fact that multiple AIBs may be active simultaneously—each of which might employ cross-VP communication—increases the overhead involved in tracking the contents of the network. Since cross-VP values are live across AIB boundaries, the need to buffer the values applies regardless of the granularity of exception handling—i.e. it does not matter whether exceptions are precise at the AIB level or VP instruction level.

The above issues illustrate the fact that precise exception support would be impractical in the current Scale design. However, by using software restart markers and treating the vector-thread unit as a black box, all of the troublesome architectural features can simply be ignored. The actual compiler implementation of this usage model is considered in the next section.

7.3 Scale Software Restart Marker Compiler Analysis for Vector-Style Code

Chapter 4 presented a high-level description of techniques that could be used to generate restart regions for vectorized code. In that chapter, the vector unit was treated as a black box. The same approach can be used for Scale's vector-thread unit when dealing with vector-style code. This section discusses the compiler analysis required to implement the techniques described in Sections 4.2–4.4. Those techniques were divided into three categories: basic restart region formation, livelock avoidance for software-managed TLBs, and performance optimizations. Before considering the compiler modifications needed for each of the categories, I first present a brief overview of the Scale compiler infrastructure,[1] as some of the analysis used to determine whether to parallelize code is re-used when inserting software restart markers. I then discuss the additional changes that are required for the three categories of techniques listed above. Although I have actually implemented many of these changes in the Scale compiler, there were some issues encountered when trying to implement certain techniques, such as copying overwritten input memory arrays, and I also discuss those problems in this section.

7.3.1 Parallelization in the Scale Compiler

A key part of this thesis work was the implementation of the Scale compiler infrastructure used to parallelize code. Figure 7-4 is a simplified version of the Scale compiler flow that omits many of the details irrelevant to software restart markers. The infrastructure ties together three existing compilers: SUIF [WFW$^+$94], Trimaran [CGH$^+$05], and GCC [gcc]. The SUIF front end accepts C source code and performs a memory dependence analysis that will be used during parallelization. The Trimaran back end maps loops to the vector-thread unit and also inserts software restart markers around the parallelized code. The fact that Trimaran uses region-based compilation [HHR95] facilitates the creation of restart regions. Finally, the GCC cross compiler tool chain produces the final executable.

The Scale compiler takes advantage of many of the features of the architecture, and is able to parallelize straightforward DOALL loops as well as loops that typically pose difficulties for traditional vectorizing compilers, such as loops with internal control flow,

[1]All references to the "Scale compiler" in this thesis refer to the compiler for the Scale architecture, not the Scale compiler [sca] for the TRIPS architecture.

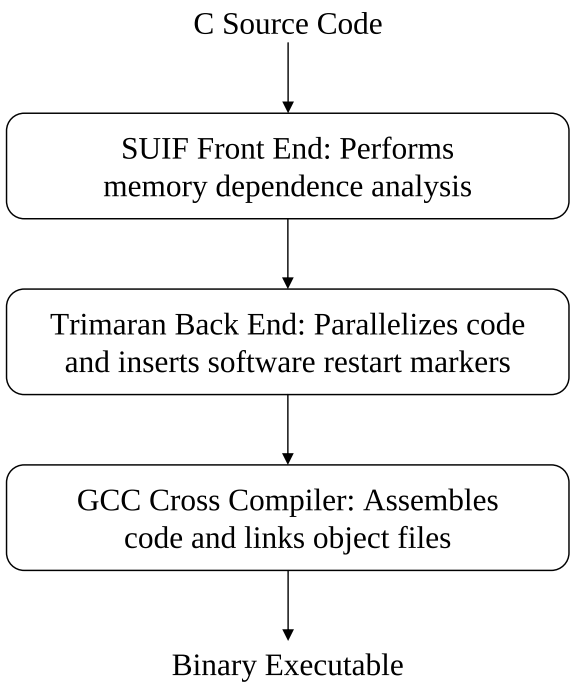

C Source Code
SUIF Front End: Performs memory dependence analysis
Trimaran Back End: Parallelizes code and inserts software restart markers
GCC Cross Compiler: Assembles code and links object files
Binary Executable

Figure 7-4: Simplified version of the Scale compiler flow.

loops with cross-iteration dependences, and outer loops. A more detailed description and evaluation of the Scale compiler infrastructure is presented in Appendix A, which shows how the compiler actually maps code to the vector-thread unit, and also demonstrates how the compiler exploits architectural features such as chain registers that are facilitated by the use of software restart markers. Besides showing that it is feasible to compile for Scale, Appendix A also supports the claim made in the introduction to this thesis that explicitly parallel architectures will likely become more widespread in the future. Although the Scale compiler is relatively early in its development, it still achieves significant speedups over scalar code running on a single-issue processor. In certain cases, the Scale compiler speedups exceed the speedups that would have been possible on an ideal multiprocessor with an infinite number of processing elements and support for thread-level speculation. While the topic of Scale compilation is an interesting subject on its own merits, for the purpose of restart region formation, the important aspect of the compiler is the analysis that is conducted when determining whether to parallelize code.

Since the Scale compiler currently only targets loops for parallelization, it has to determine if loop iterations can be executed in parallel. This requirement motivated the use of SUIF as the front end so that I could take advantage of its dependence library to conduct an accurate memory dependence analysis. Another reason for using SUIF was the fact that I had full access to the source code, which enabled me to add support for the `restrict` keyword, which indicates that the object pointed to by a particular pointer will not be accessed by any other pointer. The use of `restrict` was necessitated by the fact that all of the benchmarks evaluated in this thesis are written in C, and the use of pointers creates an aliasing problem—if the compiler cannot determine whether two pointers are aliased, it has to conservatively assume that they point to the same object, possibly preventing parallelization. Since alias analysis is a complicated topic beyond the scope of this thesis, I employed the `restrict` keyword to avoid this issue.

The results of the memory dependence analysis are merged with a separate register dependence analysis in the Trimaran back end during the parallelization phase. The compiler can handle DOALL loops as well as DOACROSS loops for which all cross-iteration dependences have a distance of 1; however, it does not currently handle loops with data-dependent exit conditions—i.e. "while" loops. To determine what type of loop is being processed, the compiler performs induction variable recognition. If the only dependences between loop iterations correspond to induction variables, then the compiler is dealing with a DOALL loop. In this case, induction variables are the only *register* variables that are live upon loop entry and also overwritten within the loop. (The same statement is not necessarily true for memory variables, as will be discussed shortly.) Similarly, when dealing with a DOACROSS loop, there are additional (non-induction) variables that introduce dependences between loop iterations, and the compiler has to be able to map these variables to Scale's cross-VP network in order to successfully parallelize the code. In the DOACROSS case, both the cross-VP register variables and the induction variables are live upon loop entry and also overwritten within the loop. Thus, although the analysis used for parallelization focuses on dependences between loop iterations, it also implicitly discovers the register variables that are live when entering the loop and overwritten during execution of the loop. For reasons described in the next section, compiler-generated restart regions in this thesis are limited to the granularity of a loop, meaning that the dependence analysis finds the overwritten register inputs to a restart region—an essential component of creating idempotent regions of code.

However, the results of the dependence analysis do not necessarily reflect whether memory variables that are live upon loop entry are overwritten within the loop. For example, the multiply_2 function in Figure 7-5 takes an input array, multiplies each element by 2, and stores the results back into the same array. Each loop iteration contains a load and store of the same memory element, and there is a dependence between each load/store pair. Since the dependence is contained within a single iteration, the loop is a DOALL loop, and the only values live between iterations are induction variables. However, the input array is live upon loop entry and also modified within the loop. The dependence graph does not reflect this fact since a different element of the array is accessed within each iteration; by contrast, a register is a single value, so if a register is modified in one loop iteration, the same register will be modified in the next iteration (assuming no internal control flow), forming a dependence. Thus, the dependence analysis used for parallelization provides information about which register inputs to the loop are overwritten, but not about which memory inputs are overwritten. This lack of information causes problems for the restart region analysis, as will be covered in Section 7.3.2.

```
void multiply_2(char *in, size_t n)
{
  int i;
  for (i = 0; i < n; i++)
    in[i] = in[i] * 2;
}
```

Figure 7-5: C `multiply_2` function.

Although some of the parallelization analysis can be re-used when inserting software restart markers in order to determine which register inputs to a loop are overwritten, the actual register analysis required for restart region formation by itself is less restrictive. When forming restart regions, the compiler only needs to know which loop inputs are written at least once within the loop. It does not need to know whether a variable is an induction variable or whether a loop-carried dependence can be mapped to the cross-VP network. This additional information is only required so that the compiler can determine if it is safe to parallelize the code. However, as the following sections demonstrate, there is some information required for software restart marker insertion that is not needed for parallelization.

7.3.2 Basic Restart Region Analysis

The fundamental approach to successfully generating restart regions is to ensure that the inputs to the region are preserved. There are two kinds of inputs to consider: register inputs and memory inputs.

Preserving Overwritten Register Inputs

As discussed in Chapter 4, functions with no overwritten input memory arrays can be made idempotent by simply ensuring that the input register arguments are not overwritten. The initial compiler implementation of software restart markers described in [HA06] for scalar code used a separate pass to create simple restart regions at the granularity of a function by preserving the input registers. After I added support to the compiler to parallelize code for Scale, I made two changes to this approach. First, since the Scale compiler only parallelizes loops, not straight-line code, the maximum granularity of a restart region is now a loop instead of a function. Limiting regions to the level of a loop helps to reduce potential re-execution overhead in complex functions, simplifies livelock analysis when dealing with software-managed TLBs, and allows the same compiler code to be used for both the basic restart region approach and the counted loop optimization. Second, rather than keeping software restart marker insertion as a separate pass, I was able to integrate the analysis into the compiler pass that actually parallelizes code, avoiding duplication of effort.

The basic approach to preserving overwritten register inputs is to detect the values that are live upon entering a region, determine whether they are modified within the region, and to make copies of any registers that are modified. As indicated in the previous section, for a DOALL loop all registers that are live upon loop entry and that are modified within the loop must contain induction variables. Thus, if the compiler parallelizes a particular DOALL loop, it then uses Trimaran's existing analysis to detect induction variables and copy those values to working registers in a special preloop block. The compiler also changes the parallelized loop to use the working registers for all instructions that read or write the induction variables. Figure 7-6 revisits the `memcpy` example from Chapter 4 and shows the output of the compiler when it generates restart regions in Scale code for this function. A special `restart_on` instruction replaces the previous *begin restart region* pseudocode that indicates the start of a new restart region. This instruction uses a PC-relative offset to determine the address to be placed in the restart PC—in this example, the address of the instruction immediately following `restart_on` is the restart address in the event of an exception. The `vcfgvl` VTU configuration command is part of the restart region so that if a context switch occurs, the VTU will be properly reconfigured when the process resumes execution. The `restart_off` instruction in the postloop block indicates that the processor should revert to conventional precise exception semantics. Since the control processor is decoupled from the VTU, the compiler has to ensure that the processor does not issue the `restart_off` instruction and then run ahead issuing additional instructions outside of the restart region. Thus, the compiler also inserts a `vsync` instruction before the `restart_off` instruction so that the control processor will stall until the VTU is idle.

Note that the registers `a0`, `a1`, and `a2` are no longer used in the execution of the loop in Figure 7-6. In fact, these registers are not used again in the rest of the function. Thus, from the standpoint of the compiler, the `a0`, `a1`, and `a2` registers are dead once the `move` instructions are issued to copy the induction variables. As a result, the register allocator by default will map each pair of original/backup virtual registers to the same physical register—e.g. instead of allocating the unique physical registers `t3` and `a0` to the corresponding virtual registers, the compiler will allocate `a0` to both virtual registers, leading to a `move, a0, a0` instruction. To prevent this problem, the compiler currently uses Trimaran's EL_OPER_LIVEOUT_DEST flag, which indicates that an operation writes a register which is live at the exit of the enclosing procedure. It searches through the procedure for instructions that write registers which hold the original input values for the restart region and marks them with the flag—in the example of Figure 7-6, the compiler will mark instructions which write `a0`, `a1`, and `a2`. While the assembly code segment in the figure does not show any writes to these registers, the Trimaran intermediate representation contains `MOVE` instructions that copy the values of the function parameters (treated as pseudo registers)

```
          restart_on preloop              # Begin the restart region and place
                                          # preloop address into restart PC
preloop:  li t0, 128                      # Set up maximum vector length
          move t3, a0                     # Copy the induction variables
          move t2, a1
          move t1, a2
          vcfgvl s0, t0, 1,0,1,0,0,0,0,0  # Configure the VTU: c0 and c1
                                          # each have 1 private register
loop:     setvl t0, t1                    # Set vector length
          vlb t2, c1/pr0                  # Load input vector
          vf memcpy_aib                   # Store data has to be in sd regs
          vsb t3, c0/sd0                  # Copy the data to output vector
          subu t1, t1, t0                 # Decrement the number of bytes remaining
          addu t2, t2, t0                 # Increment the input base register
          addu t3, t3, t0                 # Increment the output base register
          bnez t1, loop                   # Is loop done?
postloop: vsync                           # Stall until VTU is idle
          restart_off                     # End the restart region
done:     jr ra                           # Return from function

memcpy_aib:
          .aib begin
          c1 copy pr0      -> c0/sd0
          .aib end
```

Figure 7-6: Compiler-generated Scale code segment for the `memcpy` function, somewhat modified for clarity. The `restart_on`/`restart_off` instructions delimit a baseline restart region without any performance optimizations. The highlighted instructions are inserted to form the restart region.

into the corresponding argument registers. These `MOVE` instructions are marked with the EL_OPER_LIVEOUT_DEST flag.

When I initially implemented this approach, the baseline Trimaran compiler did not actually account for the EL_OPER_LIVEOUT_DEST flag during register allocation, so the allocator would still assign the same physical registers for the original induction variables and the copied values. I modified Trimaran's liveness analyzer so that it would keep track of every value that is marked by the flag and treat it as live throughout the entire procedure. These changes were made before I actually developed the Scale compiler infrastructure that could parallelize code. Now that the infrastructure is actually in place, an issue with using the EL_OPER_LIVEOUT_DEST flag has been made apparent: extending the lifetimes of registers throughout the entire procedure can significantly increase register pressure in functions that have multiple loops which are parallelized. An approach that only extends register lifetimes throughout the basic blocks of a restart region would be more appropriate to reduce the possibility of generating extra spill code.

The `memcpy` example is a case of a DOALL loop in which all overwritten register inputs to the loop are induction variables. A parallelized loop that overwrites register inputs which are not induction variables must be a DOACROSS loop that maps these loop-carried dependences to the cross-VP network. The Scale compiler keeps a separate structure that tracks these values; when creating restart regions, it uses this structure to make copies of the original values before entering the restart region, and to use those copies in the parallelized code, similar to the DOALL scenario above.

Handling Overwritten Input Memory Arrays

Chapter 4 considered different options for dealing with overwritten input memory arrays, and concluded that making a copy of the array elements would probably be the preferred approach. In the course of attempting to implement this technique, I encountered two complications which can be overcome, but are not actually addressed in the compiler implementation. Within the overall compiler development effort, dealing with overwritten input arrays was a lower priority than other techniques such as the counted loop optimization due to the fact that many of the benchmarks that have been mapped to Scale use disjoint input and output arrays; in fact, this characteristic is true of all of the benchmarks that the Scale compiler has been able to successfully handle for the evaluation.

The first complication when dealing with overwritten input arrays is the issue of determining when an array is overwritten. While it may seem simple to detect this scenario, the fact that software restart markers are inserted in the compiler back end makes the process more difficult. For example, consider again the multiply_2 function in Figure 7-5. Although the load and store operations in each iteration access the same memory element, when the compiler processes the C code for this function, it creates two different base registers—one for the load and one for the store—and assigns the same value to each register. During the actual formation of restart regions, the compiler examines the load and store instructions and sees unique base registers, so it assumes that the memory arrays being accessed are different. One way to deal with this problem is to search through the entire procedure for the initialization of each base register, and to determine whether different registers actually correspond to the same array. An issue with this approach is that for complex memory access patterns, it could be difficult to make this determination. A more appropriate solution would be to modify the front end to annotate load and store instructions with the corresponding array names from the source code. SUIF already contains a special array class that can be used to retrieve this information.

Note that detecting whether two instructions access the same array is also required for the technique of array renaming [AK01]. This technique is used to copy particular elements of an array to temporary storage in order to remove false dependences. Array renaming requires the compiler to determine whether two different accesses are to the same memory element. While this capability would be useful when inserting software restart markers, the compiler can also adopt a more conservative approach of always making a copy of the modified input array even if the modified elements are different than those that are read. The latter approach decreases the amount of analysis required at the possible expense of making unnecessary copies.

Aside from the issue of detecting accesses to the same array, a second complication is determining the actual size of the array to be copied. In Figure 7-5, `n` is the size of the `in` array, but the compiler does not have this information and has to analyze the memory accesses to determine the array bounds. For the `multiply_2` function, this analysis is straightforward: unit-stride accesses are used for `in`, so the compiler can determine that the size of the array is equal to the number of loop iterations. The same approach can be used for any innermost loop that contains strided memory accesses, and in fact, the Scale compiler currently implements these calculations when conducting livelock analysis, as described in the next section. However, outer loop parallelization can make the array size computation more complex when dealing with memory accesses within inner loops. Although determining array sizes in the compiler back end is still feasible for outer loops, it would probably be more appropriate to use the high-level information in the front end and attach the size information as annotations. Indexed memory accesses pose an additional difficulty since the compiler can no longer use a stride computation to determine the array bounds. One solution might be to have the compiler insert code that performs a run time check of the index vector values and keeps track of the minimum and maximum indices that will be accessed. Another solution is to use a programmer directive to specify the size of the array.

7.3.3 Livelock Analysis for Software-Managed TLBs

Chapter 4 discussed the issues with using a software-managed TLB in a vector processor, and the same issues exist for Scale. For these reasons, hardware-managed TLBs are a natural design choice for vector and vector-thread architectures. The Scale processor prototype does not have any TLB, but there is no architectural constraint that would prevent future vector-thread implementations from incorporating a hardware-managed TLB. However, in order to provide a sufficient test of the software restart markers model, the evaluation in Chapter 8 adds a software-managed TLB to the Scale simulation infrastructure, so I modified the compiler to generate code for this design. As stated previously, a software-managed TLB introduces the possibility of livelock, so the compiler has to prevent this problem from occurring when generating restart regions.

In order to avoid livelock, the compiler has to ensure that the total number of pages accessed within each restart region does not exceed the number of entries in the TLB that can be used by the program being compiled. Thus, the Scale compiler allows the user to specify both the page size and the number of available TLB entries, with the default values being the ones used in the evaluation in the next chapter. The user can also indicate if the compiler is targeting a system with a software-managed TLB. The processor might have separate TLBs for instruction and data, but currently the compiler assumes a unified structure, which is safe as long as each TLB in a split-TLB system has the number of available entries used in the compiler's analysis.

The compiler's livelock avoidance algorithm is outlined in Figure 7-7. Since the compiler needs to compute an upper bound on how many distinct pages are accessed in each loop iteration, the algorithm cannot be used for loops with indexed memory accesses. The algorithm is also currently restricted to innermost loops. The compiler first determines a safe upper bound for the total number of pages that can be accessed by memory instructions without incurring livelock. Since a unified TLB is assumed, the compiler estimates the number of entries needed to map the pages containing the loop instructions. The actual instructions in the intermediate representation will be converted to one or more assembly instructions. Additionally, Scale VP assembly instructions are converted into multiple primitive operations, or *primops*, which are used in the binary executable. Thus, the number of IR instructions is multiplied by an expansion factor—currently set at 10—to overestimate the final number of loop instructions in the binary. This result is then multiplied by the number of bytes in each instruction—an instruction is 4 bytes in Scale—and the total product is divided by the page size to obtain the amount of space (in pages) occupied by binary instructions. The division operation may have a remainder, so the quotient is incremented by 1. Depending on the alignment of the start of the block of instructions in memory, an extra page might be required—e.g. in a system with 4 KB pages, a contiguous 6 KB block (one-and-a-half pages worth of data) could be spread across three different pages. This possibility is accounted for by the alignment factor, which also accounts for the fact that the initial address of each data access in the loop might not be aligned to the start of a page.

In step 2 of the algorithm, the compiler makes a conservative estimate of the total number of pages corresponding to data accesses in a single loop iteration. Since the value of `vlmax` is not determined until the VTU register configuration is known later in the compilation process, the compiler assumes in this step that `vlmax` is set to 1. It also assumes that each memory instruction accesses a unique page, for the reason specified in Section 7.3.2, although additional analysis could lift this restriction. The number of pages that correspond to a single iteration is therefore the summation of each memory instruction's stride value in bytes divided by the page size. The stride value is used to represent the spacing between each memory element, which comes into play when the real value of `vlmax` is used.

The maximum number of iterations in a restart region when `vlmax` is 1 is the result of step 1 divided by the result of step 2—this computation is performed in step 3. In step 4, an instruction is inserted after the `vcfgvl` configuration command to divide the result of step 3 by the actual value of `vlmax`, obtaining the final result for the maximum number of iterations in a single restart region. Step 5 then modifies the loop to restrict the number of iterations in a region to be this value.

1. Calculate a safe upper bound on the total number of pages that can be mapped by the TLB for data accesses: $ub_{data} = num\_tlb\_entries - ub_{inst} - alignment\_factor$
   a. $ub_{inst} = [(num\_loop\_instructions) * (expansion\_factor) * (bytes\_per\_instruction) / page\_size] + 1$
   b. $alignment\_factor = num\_mem\_instructions + 1$
2. Estimate the total number of pages (possibly fractional) corresponding to data accesses in a single loop iteration, assuming that `vlmax` is 1:
   $num\_pages\_per\_iteration = (\sum mem\_instruction\_strides_{bytes}) / page\_size$
3. Obtain the value for the maximum number of iterations when `vlmax` is 1:
   $max\_iterations\_single\_vp = ub_{data} / num\_pages\_per\_iteration$
4. Insert an instruction in the compiler-generated code to compute the maximum number of iterations for the actual `vlmax` that is set by the `vcfgvl` instruction:
   $max\_iterations = max\_iterations\_single\_vp / vcfgvl\_dest\_reg$
5. Modify the loop so that only *max_iterations* loop iterations will be executed in a restart region

Figure 7-7: The algorithm used to avoid livelock when compiling for a processor with a software-managed TLB.

Figure 7-8 shows the changes that are made to the code in Figure 7-6 to avoid livelock. The code is generated for the evaluation setup used in Chapter 8: 4 KB pages and 248 TLB entries available for user-level memory accesses. The upper bound on the number of pages for instructions is 1, and the alignment factor is 3, so the result of step 1 is 244. The vector-load and vector-store instructions in the loop operate on byte elements and are unit-strided, so the sum of their equivalent byte strides is 2, and the result of step 2 is 2/4096. The maximum number of iterations when `vlmax` is 1 is therefore 499,712. Note that the modified restart region now contains three branches. The first branch at the end of the loop body is used to determine whether the loop has completed. The second branch is used to determine whether the restart region has completed, while the third (unconditional) branch is used to begin a new restart region. An optimization could be employed to remove one of the two additional branches if the iteration counter for the restart region is restricted to always be less than or equal to the total number of iterations remaining in the loop.

```
restart:  restart_on preloop              # Begin the restart region and place preloop address into restart PC
preloop:  li t0, 128                      # Set up maximum vector length
          move t3, a0                     # Copy the induction variables
          move t2, a1
          move t1, a2
          vcfgvl s0, t0, 1,0,1,0,0,0,0,0  # Configure the VTU: c0 and c1 each have 1 private register
          li t4, 499712                   # Get maximum number of iterations for vlmax=1
          div t4, t4, s0                  # Get maximum number of iterations for actual vlmax
loop:     setvl t0, t1                    # Set vector length
          vlb t2, c1/pr0                  # Load input vector
          vf memcpy_aib                   # Store data has to be in sd regs
          vsb t3, c0/sd0                  # Copy the data to output vector
          subu t1, t1, t0                 # Decrement the number of bytes remaining
          addu t2, t2, t0                 # Increment the input base register
          addu t3, t3, t0                 # Increment the output base register
          beqz t1, postloop               # Is loop done?
livelock: subu t4, t4, 1                  # Decrement iteration counter for restart region
          bnez t4, loop                   # Have maximum number of iterations been processed?
          vsync                           # Stall until VTU is idle
          restart_off                     # End the restart region
          move a0, t3                     # Update induction variables
          move a1, t2
          move a2, t1
          j restart
postloop: vsync                           # Stall until VTU is idle
          restart_off                     # End the restart region
done:     jr ra                           # Return from function

memcpy_aib:
          .aib begin
          c1 copy pr0      -> c0/sd0
          .aib end
```

Figure 7-8: Scale code for the `memcpy` function when creating restart regions with a software-managed TLB. The modifications necessary to avoid livelock are highlighted.

7.3.4 Analysis for Performance Optimizations

Section 4.4 discussed two optimizations to reduce the performance overhead of software restart markers: prefetching entries for a software-managed TLB; and the counted loop optimization. This section presents a brief description of the compiler analysis necessary to implement these optimizations, which builds on the compiler modifications that have already been described.

Implementing the Prefetching Optimization

The analysis used to avoid livelock can also be used to implement the prefetching optimization for software-managed TLBs. The compiler calculates the upper bound on the total number of pages that will be accessed in the restart region by each memory instruction within the loop. It then sets up a prefetching loop for each memory instruction to obtain all of the page mappings in advance before entering the restart region. Each prefetching loop uses a special TLB prefetch instruction that has the same format as load instructions. This prefetch instruction triggers a TLB access for the specified memory address, but does not actually make a memory request. Alternatively, a standard load instruction could be used to prefetch a mapping, but that approach could introduce additional stall cycles if the load triggers a cache miss.

A problem with the current prefetching implementation is that the size of each array is unknown, so a prefetch instruction could access a protected memory page past the bounds of the arrays. To solve this problem, the prefetch instruction is marked as speculative so that it will not cause an exception if the process accesses a page outside of its protection domain.

Implementing the Counted Loop Optimization

The counted loop optimization from Section 4.4.2 is used to restart after an exception from the earliest uncompleted loop iteration. The optimization can be used if the only variables live upon loop entry and modified within the loop are induction variables; as mentioned in Section 7.3.2, these conditions must already be satisfied to parallelize DOALL loops in the Scale compiler, simplifying the analysis required. The restart region formation for this optimization is similar to the basic software restart markers model in that the initial induction variable values are copied to working registers, and the working registers are used throughout the loop. However, for the counted loop optimization, the copies are performed before the restart region rather than at the beginning of the region. Additionally, the loop iteration counter—which is implemented as a special control register in Scale—is initialized to 0 before entering the region. The `restart_on` instruction specifies the address of the beginning of the fixup code block, which retrieves the value of the loop iteration counter, applies the loop induction operations a corresponding number of times to the original induction variable values, and jumps back to the beginning of the restart region. Finally, the `commitbar_loop` instruction is added at the end of the loop body.

7.4 Software Restart Markers in Scale Threaded Code

Vector-thread architectures have the ability to use a *free-running threads* execution model, which is useful to exploit TLP in the absence of structured loop-level parallelism. Each VP thread executes independently, using atomic memory operations to obtain tasks from a shared work queue. Since the Scale compiler does not currently generate threaded code, software restart markers have not been implemented for this execution model. However, this section presents a brief proposal for a future implementation.

The unstructured nature of the free-running threads approach makes a black box usage model inappropriate: the control processor no longer manages the execution of the VPs as a group, and individual VPs will normally perform many non-idempotent updates. Chapter 6 showed how software restart markers could be adapted to multithreaded architectures. A similar design can be used for Scale threaded code by having each VP employ the complete usage model of software restart markers. Since AIBs are atomic blocks of instructions, the smallest granularity of a restart region should be an AIB. Ideally, restart regions could be larger than AIBs, although this may not actually provide much benefit as an AIB can contain on the order of 100 VP instructions, and a non-idempotent update will likely occur before the end of an AIB is reached.

Figure 7-9 shows a thread-fetched AIB that is part of code executed under the free-running threads model. A VP loads a value, increments it, and stores the result to the same memory element. It also increments a second value and fetches the next AIB to be executed. In order to place this code inside a restart region, a necessary requirement is to store the restart address somewhere. VPs have no existing notion of a program counter, but if the address of the first AIB in a restart region is kept in a register, a VP can begin fetching from this address after an exception. One approach is to reserve one of the VP's general registers to serve as the restart PC. In Figure 7-10, `c1/pr0` acts as the restart PC that will be updated at the beginning of a new restart region. In the example, each `fetch` instruction that writes `c1/pr0` will end the current restart region and begin a new region. Thus, a region is formed around the `aib1` AIB. This model also permits a region to contain multiple AIBs: only the last AIB will update `c1/pr0`. However, the region in Figure 7-10 is not yet idempotent, as the store instruction overwrites an input value to the AIB, and the second `addu` instruction on cluster 2 also performs a non-idempotent update.

```
aib1:
       .aib begin
       c0 lw 0(pr0)      -> c3/cr0          # Load a value
       c3 addu cr0, 1    -> c0/sd0          # Increment the value
       c0 sw sd0, 0(pr0)                    # Store the updated result
       c2 addu pr0, 1    -> c2/pr0          # Increment another value
       c1 la aib2        -> c1/cr0          # Get new AIB address
       c1 fetch cr0                         # Fetch new AIB
       .aib end
```

Figure 7-9: Thread-fetched AIB under the free-running threads model.

```
aib0:
       .
       .
       c1 la aib1        -> c1/cr0           # Get new AIB address
       c1 fetch cr0      -> c1/pr0           # Fetch new AIB, update restart PC
       .aib end

aib1:  # New restart region
       .aib begin
       c0 lw 0(pr0)      -> c3/cr0           # Load a value
       c3 addu cr0, 1    -> c0/sd0           # Increment the value
       c0 sw sd0, 0(pr0)                     # Store the updated result
       c2 addu pr0, 1    -> c2/pr0           # Increment another value
       c1 la aib2        -> c1/cr0           # Get new AIB address
       c1 fetch cr0      -> c1/pr0           # Fetch new AIB, update restart PC
       .aib end
```

Figure 7-10: Free-running threads code that uses `c1/pr0` as the restart PC.

Figure 7-11 shows modified code in which the restart regions are now idempotent. The original non-idempotent updates are now placed in a new AIB. Note that the self-dependent `addu` from Figure 7-10 has been split into two different instructions: an add that targets a different register, and a copy of the value into the original register. This modified example is idempotent as written assuming that instructions execute in program order. However, Scale's decoupled execution permits instructions from different AIBs to be executing simultaneously. For example, if cluster 1 is running ahead of the other clusters, the `fetch` instruction in the `aib1` AIB might execute and update the restart PC before the `lw` instruction has a chance to execute. If the `lw` instruction then causes a page fault, `c1/pr0` will incorrectly have `aib1a` as the restart address instead of `aib1`. To address the decoupling issue, the compiler can simply have the last instruction for each cluster in a restart region send its result to cluster 1, as that forces the `fetch` instruction to wait until all clusters have finished executing their instructions. Figure 7-12 shows the final code with restart regions. In the `aib1` AIB, the instructions on clusters 0, 2, and 3 now write the `c1/cr0` register. These writes must occur before the `la` instruction on cluster 1 can target `cr0`, so the fetch of the next AIB will only execute once all of the AIB instructions have completed. Since the store instruction in the `aib1a` AIB does not write a value, a dummy `li` instruction is inserted to synchronize the execution of cluster 0 with cluster 1.

In the event of a restartable exception such as a software-managed TLB miss, the control processor can handle the exception and restart the faulting VP from the address contained in `c1/pr0`. Alternatively, if the overhead of determining which VP caused the exception is

```
aib0:
       .
       .
       c1 la aib1        -> c1/cr0            # Get new AIB address
       c1 fetch cr0      -> c1/pr0            # Fetch new AIB, update restart PC
       .aib end

aib1:  # New restart region
       .aib begin
       c0 lw 0(pr0)      -> c3/cr0            # Load a value
       c3 addu cr0, 1    -> c0/sd0            # Increment the value
       c2 addu pr0, 1    -> c2/pr1            # Increment another value
       c1 la aib1a       -> c1/cr0            # Get new AIB address
       c1 fetch cr0      -> c1/pr0            # Fetch new AIB, update restart PC
       .aib end

aib1a: # New restart region
       .aib begin
       c0 sw sd0, 0(pr0)                      # Store the updated result
       c2 copy pr1       -> c2/pr0            # Copy the incremented value
       c1 la aib2        -> c1/cr0            # Get new AIB address
       c1 fetch cr0      -> c1/pr0            # Fetch new AIB, update restart PC
       .aib end
```

Figure 7-11: Free-running threads code that sets up idempotent restart regions.

```
aib0:
       .
       .
       c1 la aib1        -> c1/cr0            # Get new AIB address
       c1 fetch cr0      -> c1/pr0            # Fetch new AIB, update restart PC
       .aib end

aib1:  # New restart region
       .aib begin
       c0 lw 0(pr0)      -> c3/cr0, c1/cr0    # Load a value, sync with c1
       c3 addu cr0, 1    -> c0/sd0, c1/cr0    # Increment the value, sync with c1
       c2 addu pr0, 1    -> c2/pr1, c1/cr0    # Increment another value, sync with c1
       c1 la aib1a       -> c1/cr0            # Get new AIB address
       c1 fetch cr0      -> c1/pr0            # Fetch new AIB, update restart PC
       .aib end

aib1a: # New restart region
       .aib begin
       c0 sw sd0, 0(pr0)                      # Store the updated result
       c0 li 0           -> c1/cr0            # Dummy synchronization instruction
       c2 copy pr1       -> c2/pr0, c1/cr0    # Copy the incremented value, sync with c1
       c1 la aib2        -> c1/cr0            # Get new AIB address
       c1 fetch cr0      -> c1/pr0            # Fetch new AIB, update restart PC
       .aib end
```

Figure 7-12: Free-running threads code that sets up idempotent restart regions and accounts for decoupled execution.

too great, the control processor can kill all of the VPs and then restart them after handing the exception. If a swappable exception such as a page fault occurs, a context switch will be performed, so each VP's register state needs to be saved. When the process resumes, the VPs can be restarted from the addresses held in their restart PCs. The use of chain registers in each cluster can reduce the amount of per-VP architectural register state that has to be preserved across an exception. Not only can this alleviate context switch overhead, similarly to the VLIW design of Chapter 5, but it can also increase the number of free-running threads in the system, similarly to the multithreaded design of Chapter 6.

7.5 Summary

This chapter described the Scale vector-thread architecture, which simultaneously exploits multiple forms of parallelism. It would be very difficult to implement precise exceptions in Scale due to its architectural features such as the cross-VP network, but the black box usage model of software restart markers can be used in vector-style code to avoid these issues. I presented a Scale compiler implementation of software restart markers that preserves overwritten register inputs, avoids livelock for software-managed TLBs, and implements the prefetching and counted loop optimizations. Additionally, I discussed how software restart markers could be implemented in threaded code for Scale.

Chapter 8

Performance Overhead Evaluation

In this chapter, I evaluate the performance overhead of using software restart markers to implement virtual memory in Scale. I first describe the evaluation methodology and then present results which show that the performance reduction due to software restart markers is small.

8.1 Methodology

I evaluate the use of software restart markers in benchmarks from the EEMBC embedded benchmark suite [eem]. Table 8.1 contains descriptions of the benchmarks and the speedups of the handwritten parallelized code—without any use of software restart markers or virtual memory support—over the baseline scalar code. The table shows that Scale provides substantial performance improvements over an idealized single-issue processor. These speedups are included to show that the overhead of software restart markers is measured relative to a high-performance design. Also, the speedups demonstrate that even if using software restart markers causes a substantial performance degradation, the resulting slower parallelized code will still typically be an improvement over scalar code—e.g. a 90% performance reduction in parallelized `viterbi` will still be faster than the scalar version. I use handwritten code for the evaluation rather than compiler-generated code because the Scale compiler is relatively new—I spent about six months developing the infrastructure necessary to map code to the vector-thread unit—and thus it can only handle a limited number of benchmarks. Additionally, the immaturity of the compiler infrastructure also means that there is a substantial gap in performance between compiler-generated code and handwritten code, as discussed in Appendix A. Since a higher level of performance produces a larger overhead when using software restart markers, I attempt to create a worst-case scenario by manually inserting software restart markers into handwritten code, using the same techniques described in Chapter 7.

The code is evaluated on the Scale microarchitectural simulator, which includes detailed models of the VTU, but uses a single-instruction-per-cycle latency for the control processor.

Benchmark	Data Set	Description	Parallelized Code Speedup
autocor	data3	Fixed-point autocorrelation	40.0
conven	data3	Convolutional encoder	941.4
fbital	pent	Bit allocation for DSL modems	48.2
fft	data_3	Fast Fourier transform	17.7
fir	-	Finite impulse response filter	114.1
hpg	-	High pass grey-scale filter	50.2
rgbcmy	-	RGB to CMYK color conversion	23.7
rgbyiq	-	RGB to YIQ color conversion	39.9
rotate	medium	Binary image 90 degree rotation	11.0
viterbi	data_4	Viterbi decoder	10.3

Table 8.1: Benchmark descriptions and speedups of handwritten parallelized code over the baseline scalar code.

The default simulation setup uses a VTU with four lanes and four clusters in each lane. Each cluster contains 32 physical registers. The 32 KB L1 unified cache is 32-way set-associative with 32-byte lines, and L1 cache hits have a two-cycle latency. The L1 cache is backed by a magic main memory with a minimum latency of 35 cycles.

When an exception is handled, a `vkill` instruction is executed that kills all previously issued VTU commands and resets the VTU to an idle state after responses have been received for all outstanding load requests. The only exceptions that are generated in this evaluation are software-managed TLB misses. I do not model page faults, as the purpose of this work is to measure the overhead involved in allowing a machine to handle page faults rather than in the speed impact of the page faults themselves. While an actual Scale implementation would typically use a hardware page table walker, a software-managed TLB provides a worst-case scenario and allows the software restart markers model to be sufficiently tested. To provide a basis for comparison, I also approximate the performance overhead when using a system with a hardware-managed TLB. Each page has a fixed size of 4 KB. It would be more practical to permit larger page sizes in a vector-thread processor, but small pages increase the number of TLB misses, providing a more pessimistic evaluation of software restart markers. I use the standard MIPS page table structure, and model the 2 MB virtually-addressed linear user page table as well as the 2 KB physically-addressed root-level page table.

Since the maximum vector length is 128 for the default Scale configuration, the TLB for VTU memory accesses has to contain at least 128 entries to avoid livelock in a system that handles TLB refills in software. Additionally, a sufficient extra number of TLB entries are required for AIB cache refills, control processor instruction accesses, and control processor data accesses. Thus, the total number of TLB entries in the system has to be greater than 128. However, simply using a large unified TLB by itself—e.g. with 256 entries—is impractical because up to 10 cache requests can be generated in each cycle. The Tarantula extension to the Alpha architecture [EAE$^+$02] deals with a large number of simultaneous requests by using a parallel array of TLBs, one for each lane. While this simplifies the hardware, it can lead to duplicate mappings for vector memory accesses with small strides. Additionally, the exception handler for Tarantula is complicated by the fact that it has to refill the correct TLB within the array in the event of a TLB miss. Austin and Sohi [AS96] explore the use of an interleaved TLB, and also introduce the concepts of piggyback ports—which allow simultaneous requests for the same virtual page number to be satisfied by a single access—and pretranslation—which avoids TLB accesses for pointer references that access the same page. However, these techniques do not help when dealing with vector memory accesses in which each element is located on a different page. The approach employed by the evaluation in this thesis is to use a micro-TLB [CBJ92], which contains a small number of entries and is refilled in hardware by the larger unified TLB. If a TLB miss occurs in the unified TLB, then the missing entry can be refilled by software or by hardware. Other designs that have used the micro-TLB approach include the MIPS R4000 processor [MWV92] and the VIRAM-1 processor [Koz02]. For Scale, the micro-TLB contains 10 entries and can accept up to 4 translation requests in each cycle. Blocked requesters are forced to retry in the next cycle. The micro-TLB is backed by a single-ported unified TLB that is shared between the control processor and the VTU. The unified TLB contains 256 entries, 8 of which are "wired entries" reserved for root-level page table entries and kernel mappings. Both the micro-TLB and unified TLB are written in FIFO order to avoid livelock. Note that the use of a FIFO scheme means that the TLBs are not fully-associative; instead, the TLBs are structured like reorder buffers, simplifying hardware complexity.

8.2 Results

This section presents the performance reduction due to using software restart markers to support virtual memory. I first consider the worst-case scenario of using software to refill the 256-entry unified TLB. I then evaluate the more practical scenario of hardware TLB refill. Finally, I briefly discuss the results within a larger context and provide a comparison to the performance overhead that might be observed with a precise exception implementation.

8.2.1 Software-Managed TLB Evaluation

Figure 8-1 shows the reduction in performance for the parallelized code when using software restart markers in a system that refills the unified TLB in software, and when all benchmarks are run for 100 iterations. Benchmarks are weighted equally to compute an average. The performance overhead includes three components: the cost of handling TLB misses; the overhead introduced by the extra instructions used to create the restart regions; and the cost of re-executing instructions if an exception occurs in the middle of a restart region. Although I do not have detailed breakdowns of each overhead component, I present some intuition about dominant components later in this section. Results are presented for three different schemes: the original software restart marker design, with an average performance reduction of 8.8%; the same design with prefetching of TLB entries enabled, with an average performance reduction of 3.9%; and the counted loop optimization, with an average performance reduction of 4.1%. Prefetching loops were only added to benchmarks that are likely to access multiple pages of data. These are the image-processing programs in this workload: `hpg`, `rgbcmy`, `rgbyiq`, and `rotate`. For the remaining benchmarks, the baseline restart marker scheme was used even when prefetching was enabled. This technique of selective prefetching could be implemented in the compiler by using profiling. The counted loop optimization was implemented for every benchmark in which it could be supported. The only exceptions were `fbital`, `fir`, and `viterbi`. Both `fbital` and `fir` contain data-dependent updates of variables that are live upon loop entry, preventing the use of the counted loop optimization, while the `viterbi` benchmark switches the roles of the input and output buffers in each loop iteration. For these benchmarks, I reverted to the baseline restart marker scheme when obtaining the "Count" data in Figure 8-1.

I initially presented results for software restart marker performance overhead in [HA06], using many of the same benchmarks. (I incorporate two new benchmarks—`fbital` and `fir`—in this thesis evaluation, but do not use the `dither` benchmark, as there was an error when executing my manually created version of the counted loop optimization that I was not able to debug in time for the publication of this thesis.) In my previous work, I targeted an older version of the Scale architecture and also executed benchmarks using a simulator with a less realistic TLB model (which did not account for the potentially large number of simultaneous memory accesses) as well as a single-cycle magic memory model. While the use of the most recent version of the Scale architecture and a more accurate TLB model have some impact on the results in this thesis, it is mainly the more accurate modeling of memory access latency that leads to several significant differences with the data in [HA06]. For example, Figure 8-1 shows a large performance reduction for the `fft` benchmark. This degradation is due to the fact that two of the restart regions for that program are contained within a loop, and a command has to be inserted at the end of each region to synchronize the control processor and the VTU. The fact that memory accesses no longer complete in a single cycle substantially increases the amount of synchronization time. Similarly, in [HA06], the use of a `commitbar_loop` instruction (referred to in that work as an `excepbar_loop` instruction) negatively affected performance, as each instance of the command occupied an issue slot in the VTU that could be used by another instruction. While this observation still holds true for some of the benchmarks in this thesis evaluation—such as `rotate`—for certain benchmarks with smaller datasets—notably `autocor` and `conven`—the insertion of the `commitbar_loop` instruction in each loop iteration actually improves performance, apart from any consideration of virtual memory. When using a real cache model, the `commitbar_loop` changes the timing of memory accesses, and for these particular benchmarks, the different timing provides a greater benefit than the negative impact caused by an additional VTU command. Thus, for `autocor` and `conven`, the performance degradation for the counted loop optimization is smaller than the other restart marker schemes.

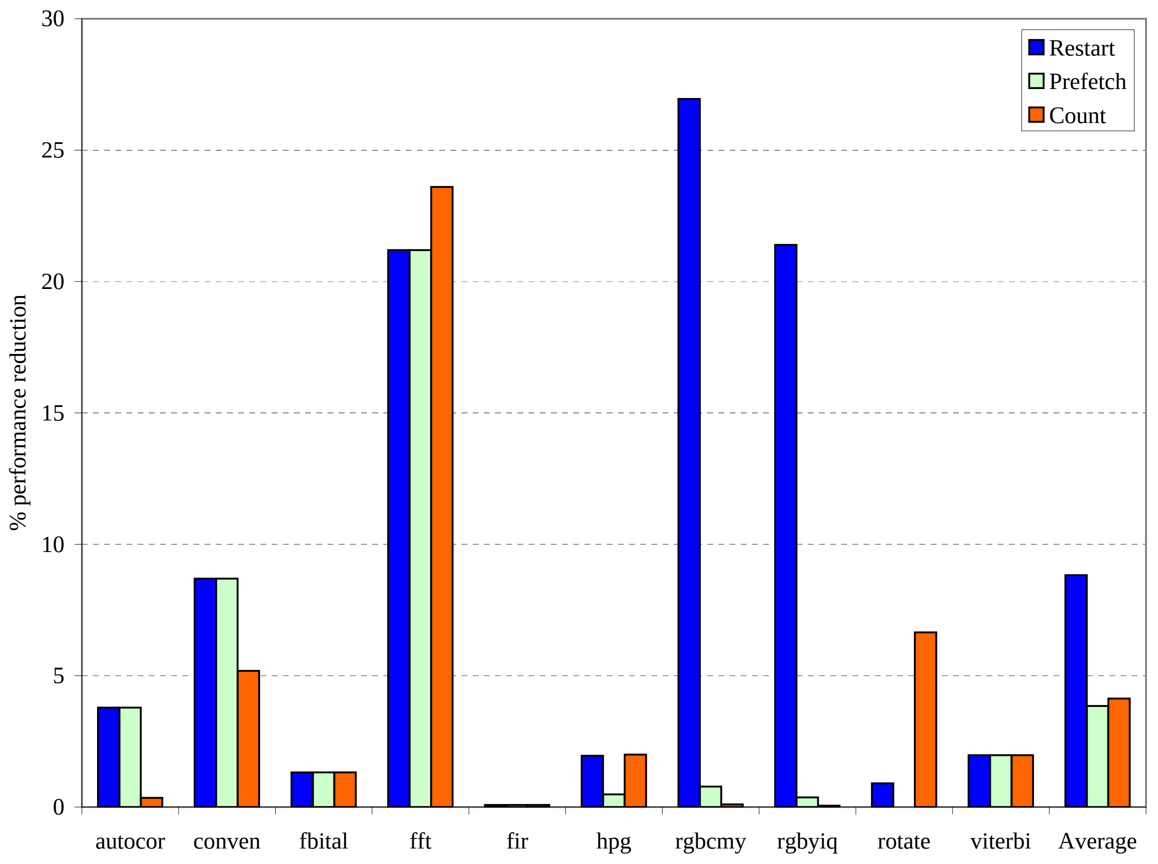


Figure 8-1: Total performance reduction in parallelized code due to using software restart markers to implement virtual memory in Scale when using a software-managed TLB. Three schemes are shown—the original restart marker design, restart markers with prefetching enabled (which only changes the image-processing benchmarks `hpg`, `rgbcmy`, `rgbyiq`, and `rotate`), and restart markers using the counted loop optimization. For the `fbital`, `fir`, and `viterbi` benchmarks, the original restart marker design is used in place of the counted loop optimization. All benchmarks are run for 100 iterations.

All of the datasets in this evaluation can be completely mapped by the TLB—the largest input dataset is about 200 KB. Thus, after the first iteration of a benchmark is executed, there will be no further TLB misses, and executing subsequent iterations will amortize any performance overhead seen in the first iteration from restarting at the beginning of a region. This effect is similar to warming up a cache that is large enough to hold the working set of a program. However, it masks the relative merit of each of the performance optimizations, and the results shown in Figure 8-1 mainly reflect the overhead component introduced by the extra instructions used to create restart regions. To address this issue, Figure 8-2 presents the performance overhead for the three different schemes when each benchmark is only run for one iteration. For the image-processing benchmarks, particularly `rgbcmy` and `rgbyiq`, the default datasets require a large number of TLB entries in order to be fully mapped. Thus, there is a significant performance overhead when using the original software restart marker design, as the multiple exceptions within each region cause the re-execution of a great deal of work. The average overhead for the original scheme is 34.4%. Using prefetching in conjunction with the original restart marker design produced the best results, with an average overhead of 3.3%. (The overhead is smaller than when running the benchmarks for 100 iterations because executing additional iterations allows the cache to warm up, improving baseline performance, while the overhead of the prefetch instructions remains relatively constant in each iteration.) The counted loop optimization did not perform as well as expected—with an average overhead of 11.5%—in part due to the fact that there is a significant latency in the Scale system between the TLB check for a memory operation and the time that the operation is retired. Thus, if multiple iterations of a loop are active at the same time, and a TLB miss occurs for the latest iteration, the exception is immediately handled, and all of the earlier work is discarded. However, for the image-processing benchmarks, which have larger datasets, the counted loop optimization typically provides a significant advantage over the original restart marker design. The one exception is `rgbhpg`, which processes an array a column at a time. As a result, there can be accesses to several different pages in a given loop iteration, or spread across a few iterations, leading to a great deal of re-executed work.

It should be noted that the prefetching scheme produces code that is tied to a particular hardware implementation—if the actual TLB size is less than that used by the compiler in its analysis, then the prefetch code will livelock. Thus, the counted loop optimization is more suitable for producing code that is compatible with a variety of implementations. Additionally, prefetching does not account for the possibility of other types of exceptions occurring within a restart region. I placed each prefetching loop before the entry to the main loop, outside of any restart regions. As a result, if a context switch occurs within a restart region and the TLB entries are flushed, when the process resumes, the performance will be similar to that of the original design. It is possible to place a prefetching loop inside of a restart region, but this can introduce additional overhead if the entire region is contained within a loop and is executed multiple times, or if an exception occurs that does not cause a context switch, as the prefetching loop will be executed unnecessarily upon each restart.

8.2.2 Hardware-Managed TLB Evaluation

As stated previously, using hardware to refill the TLB is an appropriate design choice for vector-thread architectures, as a TLB miss will not cause an exception, but will instead just introduce an extra delay for the corresponding memory access. I did not have time to implement an accurate hardware page table walker in the Scale simulator, and so I

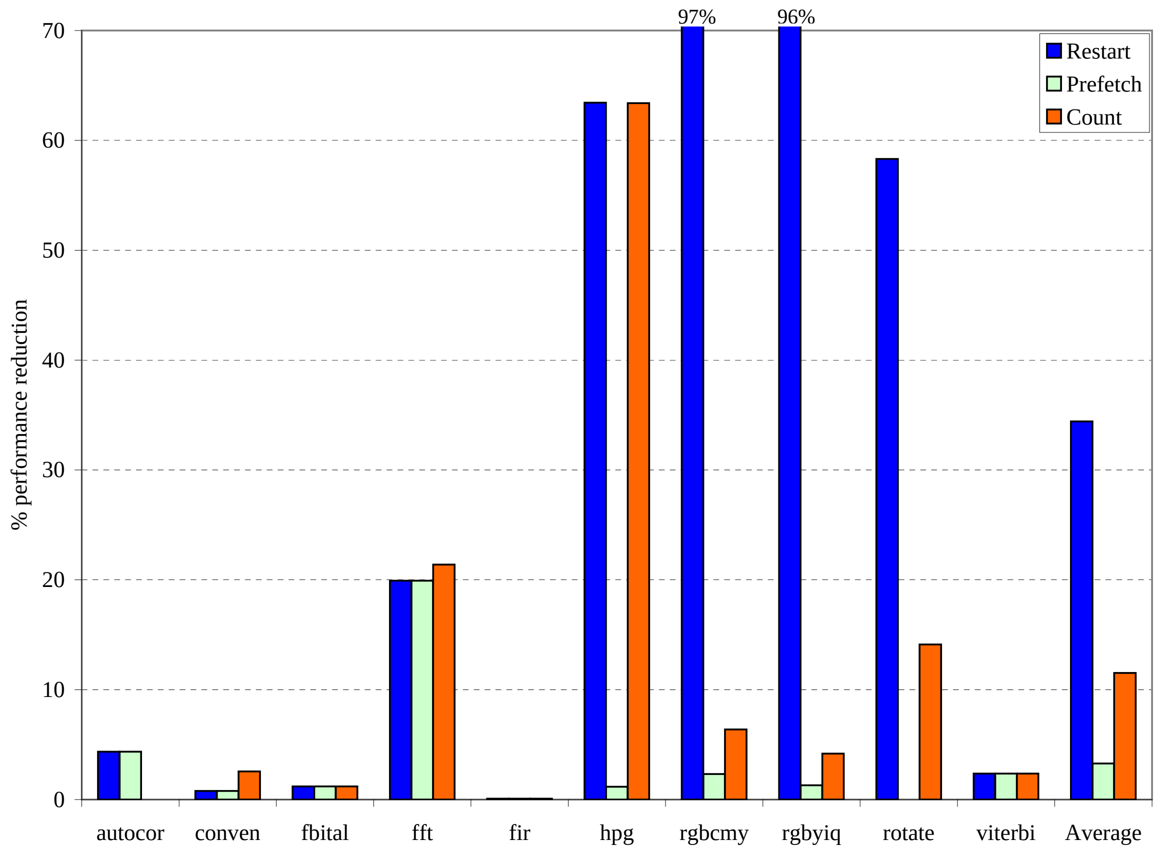


Figure 8-2: Total performance reduction in parallelized code due to using software restart markers to implement virtual memory in Scale when using a software-managed TLB and all benchmarks are run for 1 iteration.

approximate the impact of using a hardware-managed unified TLB by adding a fixed latency to the total cycle count in the event of a TLB miss. The software handler for TLB misses used in the previous section can refill the TLB in fewer than 10 cycles if there is no cache miss or TLB miss for the corresponding virtually-addressed page table entry. If a cache miss occurs but there is no TLB miss for the page table entry, the handler will take nearly 50 cycles to refill the TLB. If the page table access triggers a TLB miss, then the root-level page table has to be accessed, and the total TLB refill latency can exceed 100 cycles. However, this last scenario rarely occurs. Thus, in order to set up a reasonable worst-case scenario for using a hardware-managed TLB, I set the hardware refill latency to a fixed value of 50 cycles.

Figure 8-3 shows the performance overhead that is incurred when hardware is used to refill the unified TLB and each benchmark is run for a single iteration. Since prefetching is only useful for software-managed TLBs, that scheme is not shown in the figure. However, I do present results for the counted loop optimization, as that approach can reduce the amount of repeated work in the event of a context switch. The original restart marker design incurs an average overhead of 3.1%, which is largely attributable to the synchronization commands in the `fft` benchmark. The counted loop optimization incurs a greater average overhead of 4.0%, as the extra instructions used in implementing the counted loop optimization for the `rotate` benchmark causes a significant degradation.

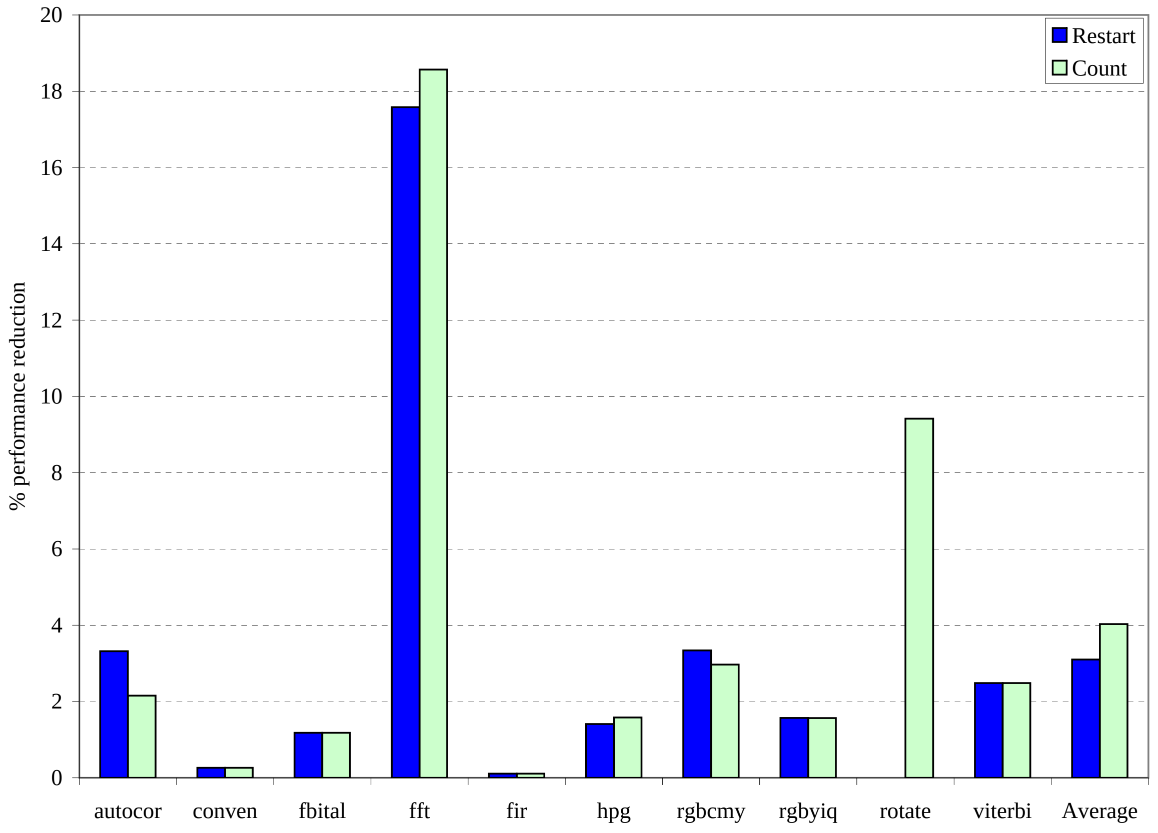


Figure 8-3: Total performance reduction in parallelized code due to using software restart markers to implement virtual memory in Scale when using a hardware-managed TLB and all benchmarks are run for 1 iteration.

8.2.3 Discussion

It is interesting to note that the average performance overhead shown in Figure 8-2 when using the prefetching optimization with a software-managed TLB is only slightly larger than the overhead shown in Figure 8-3 when using the original restart marker design with a hardware-managed TLB. Prefetching can be very effective in avoiding performance degradation. However, it does have the issues mentioned earlier in this chapter; also, software-managed TLBs introduce the general concern with livelock that has been discussed in previous chapters.

Although there are several cases in which there is a large performance reduction due to using software restart markers to implement virtual memory, recall that the substantial speedups achieved by the baseline parallelized code mean that even the parallelized benchmarks with restart regions will still typically be much faster than the corresponding scalar code. However, the issue with the synchronization overhead in the `fft` benchmark does indicate that research into alternative methods of terminating restart regions could be beneficial.

As a basis for comparison, the CODE microarchitecture [KP03] supports virtual memory in a vector machine, but by using a clustered vector register file with register renaming. The performance of CODE is evaluated across a workload containing many of the same benchmarks used in this evaluation. There is an average performance reduction of 5% when supporting precise exceptions on the default CODE configuration with 8 vector registers per cluster. This overhead is strictly due to the register pressure from implementing register renaming—the cost of handling TLB misses is not considered. By contrast, the software restart markers model requires no additional registers or store buffering, making it more scalable to future processor generations that will have larger numbers of simultaneous in-flight operations.

8.3 Summary

This chapter evaluated the use of software restart markers in the Scale vector-thread architecture, using worst-case scenarios to obtain the results. Overall, the performance overhead of software restart markers is acceptably small in light of the benefits that they provide. Even when using a software-managed TLB—which would be unlikely in an actual Scale system—software restart markers only degrade performance of highly optimized code by an average of about 4% when using either prefetching or the counted loop optimization and running benchmarks for 100 iterations. Prefetching is also able to tolerate TLB misses when executing benchmarks for a single iteration, with an average performance reduction of less than 4%. Finally, when using a hardware-managed TLB and running benchmarks for a single iteration, the average performance reduction with the original restart marker design is about 3%, while the average reduction with the counted loop optimization is about 4%.

Chapter 9

Conclusions and Future Work

An exception handling mechanism is an essential part of any computer system. Exceptions are used to indicate errors, to permit user interaction with the program, and to implement features such as virtual memory. There have been relatively few exception handling techniques that have gained mainstream acceptance, and these have been primarily used for sequential architectures in general-purpose systems. However, with the recent shift toward parallel computing and the blurring of the boundaries between processor domains, a fresh look at existing methods of dealing with exceptions is warranted.

Exception handling approaches are typically judged according to whether they adhere to the precise exception model. While precise exceptions provide many benefits—including a simple interface to the exception handler and the ability to support accurate debugging—they also require instructions to be committed to architectural state in program order. This restriction can lead to substantial exception management overhead with respect to processor area, energy consumption, and performance. Additionally, the precise exception model does not adapt well to explicitly parallel architectures, in which the overhead of handling context switches can be substantial. These sources of overhead provided the motivation for this thesis work.

9.1 Summary of Contributions

This thesis investigated an alternative to the precise exception model in the form of *software restart markers*, which use compile-time information to reduce the overhead of exception management. To lay the foundation for developing a new exception model, I introduced several axes of classification for exception handling mechanisms and showed that—contrary to what is often stated—certain important features such as virtual memory do not actually require precise exception support. I then showed how the software restart markers model could be used to enable these features by having the compiler delimit regions of idempotent code. In the event of an exception, the process will restart from the head of the region. These restart regions provide two main advantages: they permit out-of-order commit within a region, eliminating the need to buffer values; and they introduce the notion of temporary state, which does not have to be preserved across an exception.

I demonstrated the generality of software restart markers by presenting designs for vector, VLIW, and multithreaded architectures. In vector processors, the use of restart regions avoids the need to buffer large numbers of in-flight instructions and also permits the entire vector register file to be mapped to temporary state. VLIW architectures that use the EQ scheduling model can take advantage of software restart markers to enable exceptions within software-pipelined loops, and also to reduce the number of architectural registers needed in the system. Multithreaded architectures can use the concept of temporary state to reduce register pressure, potentially enabling a greater number of threads.

To evaluate the feasibility of using software restart markers as well as the performance overhead that they incur, I implemented this model in the Scale vector-thread architecture. I used the Trimaran compiler infrastructure to generate the restart regions and to implement optimizations to reduce the number of re-executed instructions. Using software restart markers for vector-style code in Scale resulted in a performance degradation of less than 4%. I also showed how software restart markers could be used for threaded code in Scale.

Since a primary motivation for this thesis was the belief that explicitly parallel architectures would become more prevalent in the future, I demonstrated the performance-enhancing potential of using parallel designs in mainstream systems by developing a compiler infrastructure for Scale. Vector-thread architectures are designed to exploit multiple forms of parallelism simultaneously; this thesis demonstrated the ability of a compiler to actually take advantage of those architectural features. The Scale compiler is able to parallelize simple DOALL loops, as well as code that causes problem for established vectorizing compilers, such as loops with complex internal control flow or cross-iteration dependences. In certain instances, the speedups of compiler-generated code running on Scale exceeded the speedups that would have been possible on an ideal multiprocessor that supported thread-level speculation. Given the fact that the compiler development is still in its initial stages and there are obvious areas in which it can be improved, these results attest to the potential of using explicitly parallel architectures.

9.2 Future Work

While this thesis touched on several different areas, there are many opportunities for future work.

Improved Handling of Memory Dependences

The current Scale compiler implementation of software restart markers is unable to generate restart regions for cases where input memory values are overwritten and the stride for the corresponding memory operations cannot be properly determined, as it then has no way of knowing what size to use when creating a backup memory array. Future work could incorporate additional front end analysis so that the total amount of memory accessed by a loop can be calculated.

Additional Applications

The evaluation of the performance overhead of software restart markers included several benchmarks, but was restricted by the fact that the Scale compiler can currently only handle a small number of EEMBC codes. Although there is a larger set of available hand-coded benchmarks, these are primarily smaller kernels within the embedded application domain. Generating restart regions in additional applications would permit a more comprehensive evaluation.

Varying Restart Region Sizes

Related to the previous point, the fact that the evaluation used small embedded kernels meant that the maximum restart region size was somewhat limited compared to what might be observed in larger applications. The implementation of restart regions in the baseline model for both compiler-generated and handwritten code made regions as large as possible, subject to the constraints of livelock. In a system with a hardware-managed TLB, restart regions could encompass millions of instructions, drastically increasing re-execution overhead in the event of an exception. If the counted loop optimization is not possible, then compiler analyses should be implemented to strip mine restart regions, aiming for a compromise between permitting parallel execution by increasing region size and avoiding re-execution overhead by decreasing region size.

Hybrid Hardware-Software Approach

Software restart markers avoid the hardware overhead associated with precise exceptions, but this advantage comes at the cost of flexibility: only instructions from one restart region at a time are executed, so in segments of code with small regions, performance could be degraded. An alternative approach would be to take an existing hardware-based exception handling mechanism and incorporate compile-time information to reduce exception management overhead. For example, software restart markers are somewhat like hardware checkpoints in a checkpoint repair scheme, but with a key difference being that only one checkpoint is active at a time in the former approach. One possibility would be to target a superscalar processor with a hardware checkpointing scheme and use compiler annotations so that the processor would only checkpoint live architectural registers. Alternatively, the architecture could be modified so that it has a handful of checkpoint registers—e.g. out of 32 architectural registers, 8 are checkpointed—and all other registers are treated as temporary state that can only be live between checkpoints. This technique would reduce the overhead of each checkpoint while still enabling a more flexible exception handling approach than software restart markers.

Evaluating Software Restart Markers in Multiprocessors

The focus of this thesis was on exception handling for a single core. In a multiprocessor system, the use of software restart markers can change the memory consistency model. Although I suggested a possible solution—add barriers around memory operations that communicate between threads—a possible area for future work would be to actually implement this technique and evaluate its impact.

Scale Compiler Development

The primary topic of this thesis was exception handling using software restart markers, but a key aspect of this work was the development of the Scale compiler infrastructure. As I pointed out previously, Scale compilation is a worthy topic on its own, and there are many areas in which the current compiler can be improved.

9.3 Concluding Remarks

This thesis has presented software restart markers as an alternative to the precise exception model for explicitly parallel architectures. Software restart markers are able to reduce the exception management overhead associated with precise exception implementations. I have shown that using software restart markers can provide multiple benefits while incurring little performance degradation. Hopefully this work will serve to spark additional research into the frequently neglected area of exception handling.

Appendix A

Compiling for Vector-Thread Architectures

This appendix provides an overview of the compiler infrastructure for Scale that was developed as part of this thesis work. A substantial portion of this appendix is also presented elsewhere [HA08]. As discussed in Section 7.3, the compiler analysis used to create restart regions builds on the analysis used when parallelizing Scale code. While this appendix provides additional details about the parallelization process, it also serves another purpose. In the introduction to this thesis, I listed the shift to explicitly parallel computing as a key motivating factor for investigating alternatives to the precise exception model. The bulk of this thesis has been devoted to showing how software restart markers can be used to cheaply handle exceptions in explicitly parallel architectures. However, there are other important requirements that a new processor design should satisfy to facilitate its widespread adoption: it should enable a high level of performance; it should be energy-efficient; and it should be a good compilation target. If a design does not have these characteristics, it will probably find limited success, and the particular approach it uses to handle exceptions is not likely to have a significant effect on its longevity. Chapter 8 presented data indicating that the performance degradation caused by software restart markers in the Scale architecture is small. Krashinsky [Kra07] shows that Scale is also a high-performance, energy-efficient design for the embedded processor domain. This appendix addresses the final requirement listed above: whether Scale is a suitable compilation target. It presents results showing that the compiler can exploit parallelism in a variety of programs, even in code that would not typically be parallelizable by compilers for traditional architectures such as vector or VLIW.

A.1 Scale Compiler Overview

This section presents an overview of the Scale compiler infrastructure. Figure A-1 shows the compiler flow used for this work. The SUIF front end parses a C source code file and converts it into the SUIF intermediate representation (IR). The compiler then performs memory dependence analysis, annotating the IR with dependence information that will be later used by Trimaran. The SUIF IR is then fed into a SUIF-to-Trimaran converter [LRA05], which outputs the program in Trimaran's intermediate representation. Trimaran performs classical optimizations including common subexpression elimination, copy propagation, loop-invariant code motion, and dead code elimination. The output of the optimization phase is then sent to the scalar-to-VP code transformation phase, which attempts to map code to the vector-thread unit. If VP instructions are generated, then software restart markers will also be inserted, as discussed in Chapter 7. After the transformation phase, cluster assignment is performed, and then the first (prepass) instruction scheduling phase occurs. Prepass instruction scheduling is followed by register allocation and a second (postpass) instruction scheduling phase, after which AIBs are formed for the VP code. Once AIB formation occurs, the compiler then searches for opportunities to replace occurrences of general registers with chain registers. Finally, Trimaran generates assembly code, and this is processed by the GCC cross-compiler to create the final binary executable.

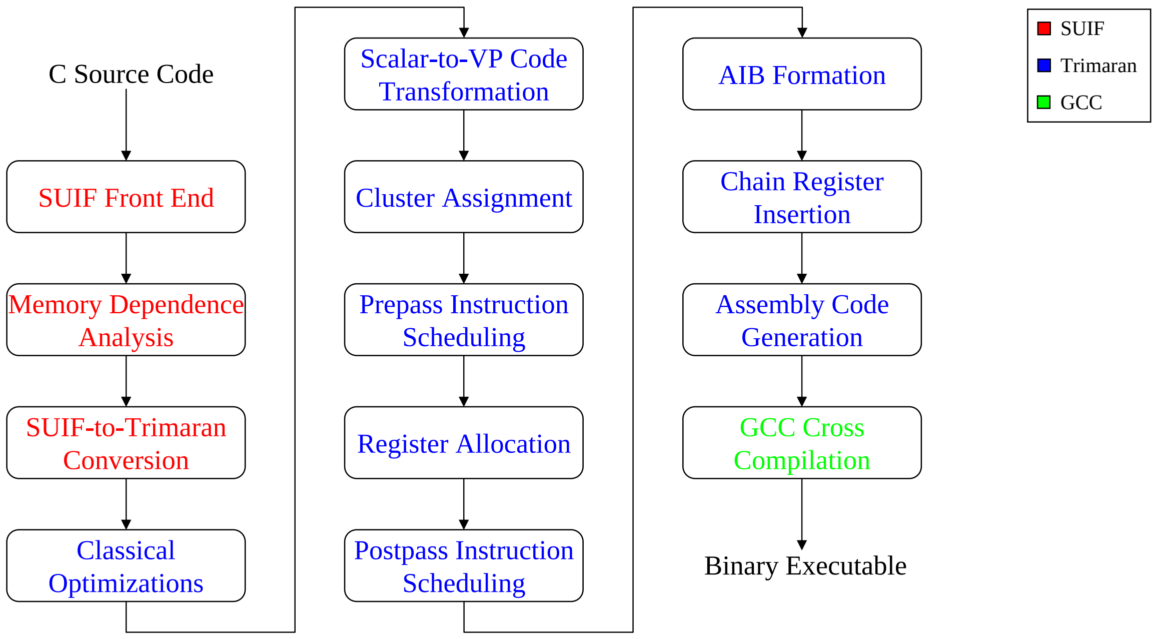


Figure A-1: Scale compiler flow. The legend indicates which compiler infrastructure is used for a particular phase.

The issue of compiling for Scale is a worthwhile topic in its own right, apart from any consideration of software restart markers. However, to keep this appendix within the overall scope of the thesis and prevent it from becoming too broad, I avoid delving too deeply into the details of the Scale compiler. Instead, I discuss certain key compiler phases, with particular emphases on how the compiler maps code to the vector-thread unit and on how it targets certain architectural features that would pose difficulties for a precise exception implementation, including the cross-VP network, chain registers, and decoupled execution. It should be noted that during the development of the Scale compiler infrastructure, I prioritized the addition of support for various Scale features such as cross-VP communication over the attempt to improve performance by modifying certain phases such as cluster assignment and instruction scheduling. As a result, I ended up employing simple heuristics in those phases, leaving substantial room for improvement in future work. The primary purpose of this section is not to introduce novel analyses or performance-enhancing techniques, but rather to show the feasibility of compiling for the Scale architecture and to describe the current snapshot of the compilation infrastructure.

A.1.1 Memory Dependence Analysis

A key motivating factor for using SUIF as the front end is its dependence library, as an accurate dependence graph is important in order to determine what parallelism can be exploited in the program. I use the library to annotate memory operations with direction vectors indicating the existence of dependences. A special direction type is used for cross-iteration dependences with a distance of 1, as these can be mapped to transfers on the cross-VP network.

The dependence analysis is potentially complicated by the fact that Scale was designed for embedded systems, which typically run programs written in C. The use of pointers in C programs creates an aliasing problem, in which the compiler cannot determine whether two different pointers will access the same memory location, and hence must make an extremely conservative assumption of dependences, resulting in little or no exploitable parallelism. To help with this problem, I extended the SUIF front end to support the `restrict` keyword, which indicates that the object pointed to by a particular pointer will not be accessed by any other pointer. While this solution requires the programmer to manually add the `restrict` keyword to the function prototype, it is no more onerous than the use of programmer-inserted OpenMP directives when parallelizing code in other systems. Alias analysis is a complex topic that falls outside of the scope of this work. However, it is also an orthogonal problem to the actual generation of code for Scale, and thus existing alias analysis techniques can be incorporated into the Scale compiler in the future.

A.1.2 Scalar-to-VP Code Transformation

In this phase, the compiler attempts to map parallel sections of code to the vector-thread unit. If it successfully generates VP code, then it can also insert software restart markers as covered in Chapter 7. However, throughout this appendix, all code is compiled without software restart markers being inserted in order to emphasize Scale-specific compilation issues apart from exception handling. Since VT architectures excel at exploiting loop-level parallelism, the compiler currently only focuses on transforming loops, which frequently dominate execution time in the types of applications considered in this work. The compiler uses Trimaran's existing infrastructure to detect all of the loop nests in the function under consideration. It then processes each loop nest in an attempt to find a loop that can be transformed. If a loop is successfully transformed, then its loop nest will not be modified any further—e.g. if the innermost loop in a nest is parallelized, none of the outer loops will be processed by this phase.

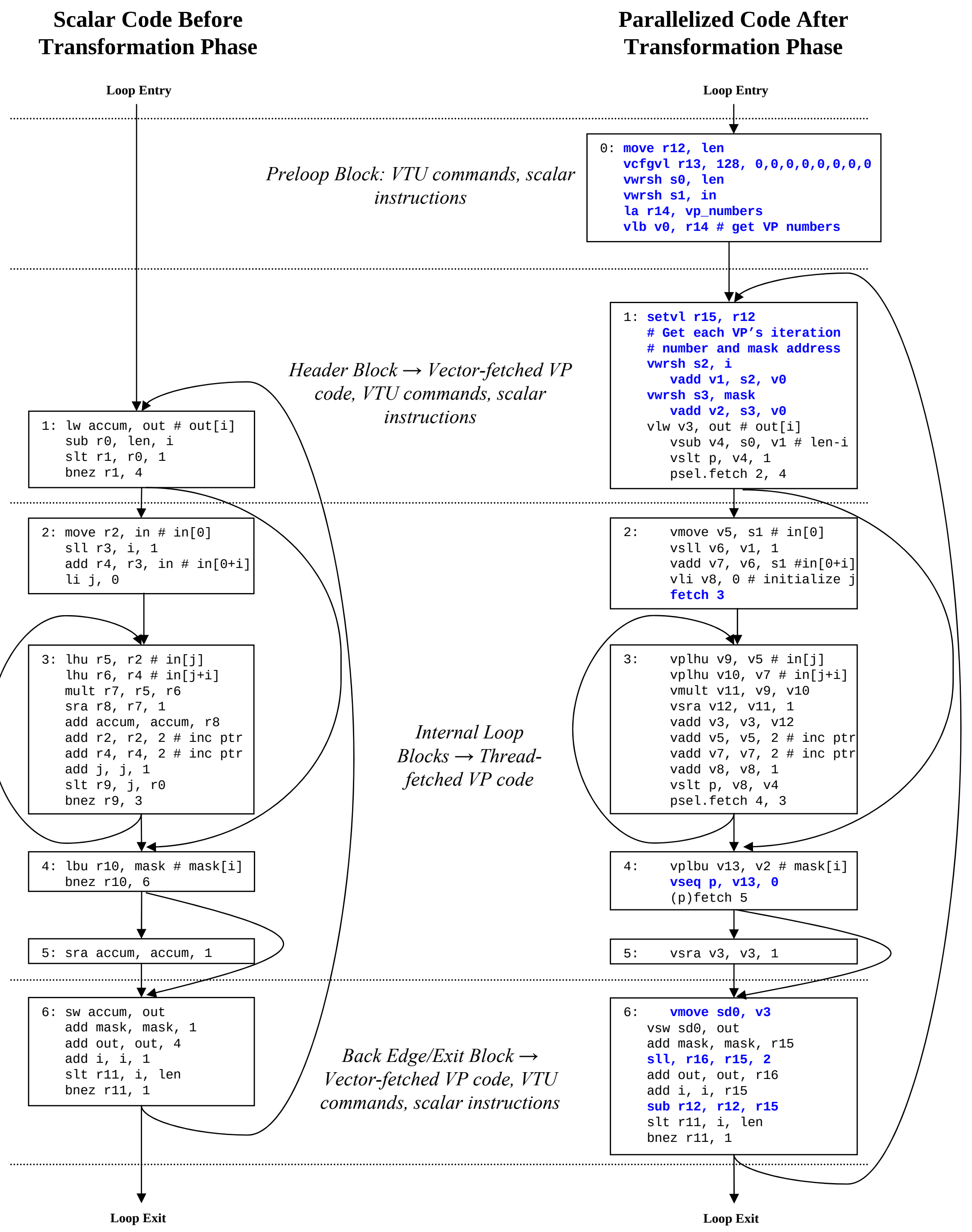
To illustrate the details of this compiler phase, I will show how the function in Figure A-2 is transformed. This example contains a loop nest that also has internal control flow. The inner loop has a loop-carried dependence due to the accumulation. Figure A-3 shows the scalar code that is the input to the transformation phase on the left. The parallelized output code of the transformation phase is shown on the right. The following sections describe how the transformation actually occurs.

```
void example(unsigned int len, unsigned short * restrict in,
             unsigned char * restrict mask, unsigned int * restrict out) {
  unsigned int i, j, accum;
  for (i = 0; i < len; i++) {
    accum = out[i];
    for (j = 0; j < len-i; j++)
      accum += (in[j]*in[j+i]) >> 1;
    if (mask[i] == 0)
      accum >>= 1;
    out[i] = accum;
  }
}
```

Figure A-2: C code example used to illustrate how the compiler works.

Code Transformation Overview

As stated in Section 7.1.3, the typical approach to parallelize loops for VT architectures is to strip mine the loop so that the control processor launches a group of VPs simultaneously, with each VP executing a single loop iteration. In this transformation phase, the basic blocks in the loop will be mapped to a combination of VTU commands (including any vector memory instructions), vector-fetched VP code, thread-fetched VP code, and scalar instructions—e.g. to process induction variables. The mapping used for a particular basic block depends on its type. Any given loop can be decomposed into basic blocks that can be grouped into four different categories, with the groups potentially overlapping so that a single block may be placed in multiple categories. First, there is a header block at the


Scalar Code Before Transformation Phase
Parallelized Code After Transformation Phase
Loop Entry
Loop Entry
Preloop Block: VTU commands, scalar instructions
Header Block → Vector-fetched VP code, VTU commands, scalar instructions
Internal Loop Blocks → Thread-fetched VP code
Back Edge/Exit Block → Vector-fetched VP code, VTU commands, scalar instructions
Loop Exit
Loop Exit

```
1: lw accum, out # out[i]
   sub r0, len, i
   slt r1, r0, 1
   bnez r1, 4

2: move r2, in # in[0]
   sll r3, i, 1
   add r4, r3, in # in[0+i]
   li j, 0

3: lhu r5, r2 # in[j]
   lhu r6, r4 # in[j+i]
   mult r7, r5, r6
   sra r8, r7, 1
   add accum, accum, r8
   add r2, r2, 2 # inc ptr
   add r4, r4, 2 # inc ptr
   add j, j, 1
   slt r9, j, r0
   bnez r9, 3

4: lbu r10, mask # mask[i]
   bnez r10, 6

5: sra accum, accum, 1

6: sw accum, out
   add mask, mask, 1
   add out, out, 4
   add i, i, 1
   slt r11, i, len
   bnez r11, 1
```

```
0: move r12, len
   vcfgvl r13, 128, 0,0,0,0,0,0,0,0
   vwrsh s0, len
   vwrsh s1, in
   la r14, vp_numbers
   vlb v0, r14 # get VP numbers

1: setvl r15, r12
   # Get each VP's iteration
   # number and mask address
   vwrsh s2, i
      vadd v1, s2, v0
   vwrsh s3, mask
      vadd v2, s3, v0
   vlw v3, out # out[i]
      vsub v4, s0, v1 # len-i
      vslt p, v4, 1
      psel.fetch 2, 4

2:    vmove v5, s1 # in[0]
      vsll v6, v1, 1
      vadd v7, v6, s1 #in[0+i]
      vli v8, 0 # initialize j
      fetch 3

3:    vplhu v9, v5 # in[j]
      vplhu v10, v7 # in[j+i]
      vmult v11, v9, v10
      vsra v12, v11, 1
      vadd v3, v3, v12
      vadd v5, v5, 2 # inc ptr
      vadd v7, v7, 2 # inc ptr
      vadd v8, v8, 1
      vslt p, v8, v4
      psel.fetch 4, 3

4:    vplbu v13, v2 # mask[i]
      vseq p, v13, 0
      (p)fetch 5

5:    vsra v3, v3, 1

6:    vmove sd0, v3
   vsw sd0, out
   add mask, mask, r15
   sll, r16, r15, 2
   add out, out, r16
   add i, i, r15
   sub r12, r12, r15
   slt r11, i, len
   bnez r11, 1
```

Figure A-3: The control flow graph on the left contains the scalar code that is the input to the transformation compiler phase. On the right is the parallelized output of the phase. All register numbers are virtual. The type of register depends on the prefix in front of the number: "r"=scalar; "v"=private VP; "s"=shared VP; "sd"=store-data VP. Certain scalar registers are referred to by their variable names for clarity. Highlighted instructions in the parallelized code are inserted by the transformation phase, as opposed to simply being converted from a scalar instruction. VP instructions that will be grouped into AIBs are indented to differentiate them from scalar instructions and VTU commands executed on the control processor.

beginning of the loop. For the outer loop in Figure A-3, the header block is block 1. There may also be a group of blocks that have a back edge to the header block. In Figure A-3, block 6 is the single back edge block for the outer loop. Additionally, there may be a group of blocks that have a control-flow edge exiting the loop—block 6 is also an exit block for the outer loop. The final group consists of any remaining blocks in the loop that do not fall into the first three categories. This group consists of blocks 2, 3, 4, and 5 for the outer loop in Figure A-3. An innermost loop with no conditionals has a single basic block, which is the header block, a back edge block, and an exit block—this is block 3 for the inner loop in Figure A-3. Although Trimaran can handle more complex loop structures, I impose the restriction that a loop can only have a single back edge block and a single exit block, and that those blocks must be the same. The reason for this requirement is so that when VPs direct their own control flow by executing fetch instructions, they will always transfer control back to the control processor at the same point in the program—in the example, this point is block 6. Otherwise, it would not be possible for the control processor to manage the VPs as a group. For example, suppose there was an additional back edge block 7 in Figure A-3, and block 5 could branch to either block 6 or block 7. If block 7 performed different induction variable updates than block 6, then in order for the strip-mined loop to work correctly, the control processor would have to know for each VP whether control was transferred back to block 6 or block 7 so that the induction variables could be correctly updated. This information would be difficult to obtain, so the simple solution is to restrict the number of back edge and exit blocks. In practice, this restriction did not cause a problem, as the codes targeted in this work were automatically mapped by the compiler to have a single back edge/exit block.

Another restriction on the compiler is that loops with data-dependent exit conditions—i.e. "while" loops—are not currently handled. One approach to deal with these loops is to speculatively execute iterations and to later nullify the effects of any VPs that should not have been launched. Future work will explore compilation of this type of code.

Figure A-3 contains annotations between the scalar code and parallelized code describing the type of mapping that occurs. In the Scale compilation model, the header block and back edge block are transformed to a combination of vector-fetched VP code for non-memory operations, VTU commands including vector memory instructions, and scalar instructions to handle any induction operations as well as the spawning of new VPs. Since the control flow for the remaining loop blocks may differ from iteration to iteration, it is dependent on data that will be unique to a particular VP. Thus, those blocks are transformed to thread-fetched VP code, and each memory instruction is transformed to a VP memory operation within the appropriate block. A VP will continue to direct its own control flow until it executes a block that does not issue a thread-fetch instruction—in Figure A-3, block 5 is a thread-fetched block that does not issue another thread-fetch instruction. At this point, the VP will halt and return control to the control processor at the back edge/exit block.

As seen in Figure A-1, the actual formation of AIBs takes place in a later phase of the compiler, when the directives to delimit the beginning and end of each AIB are inserted. This ordering means that the compiler actually interacts with VP code at the level of a basic block. Only later, when the AIBs are actually formed, is a basic block's VP code in the intermediate representation divided into one or more AIBs, as described in Section A.1.5. The motivation for this phase ordering is twofold. First, since there is a size limit for AIBs (32 instructions per cluster for Scale), if the AIB boundaries were created before register allocation, there would be a potential for spill code insertion to cause the limit to be violated. Second, delimiting AIBs before instruction scheduling would impose restrictions on the scheduler, as all VP instructions in an AIB would have to be moved as a group relative to any instructions outside the AIB. These restrictions could hinder performance—for example, as discussed in Section A.1.4, when processing several vector loads, it can be desirable to have each load immediately followed by any dependent VP computation. To avoid complicating the scheduling of VP instructions relative to non-VP instructions such as vector loads, the intermediate representation supports arbitrary intermingling of both types of instructions within a basic block. The assembly code generator handles the actual separation of VP instructions into a distinct section of the file. For each AIB that is relocated into its own section from the header block or back edge/exit block, the assembly code generator also inserts a vector-fetch instruction in its original location.

Loop Selection

In traditional vectorizing compilers, selecting which loop in a nest should be vectorized typically depends on factors such as the structure of memory accesses or the loop trip count—which requires profiling to estimate dynamically determined trip counts. This process can be quite complicated [AK01], and is not covered within the scope of this thesis. Rather than attempting to account for various loop-specific factors, the compiler currently uses a simple heuristic of first working from the innermost to the outermost loop in the nest, searching for a DOALL loop that can be transformed. It attempts to transform innermost loops first because the control processor is more efficient than virtual processors at handling branches. If no DOALL loops in a nest can be mapped to the vector-thread unit, then the compiler processes the loop nest again, attempting to map loop-carried dependences to the cross-VP network. Cross-VP communication is primarily designed to be used within a vector-fetched AIB, so the compiler first determines if the innermost loop has no internal control flow and if it can be transformed using cross-VP transfers. If that is not possible, the compiler restricts the vector length to equal the number of *independent VPs*, and again works from the innermost loop in the nest to the outermost loop. The number of independent VPs is the maximum number of VPs that can execute in parallel without having to be time-multiplexed. In the Scale implementation, this value is the number of lanes. Since using independent VPs removes the possibility a VP will be time multiplexed on a lane, it also removes the possibility of deadlock when using arbitrary cross-VP communication.

It should be noted that reduction operations can be handled on Scale by using the standard parallelization technique of computing partial results on each lane (in a shared register) and then merging the results during a final execution phase. However, I have not yet added compiler support for this approach, so currently the compiler treats all loop-carried dependences in the same manner when selecting which loop to transform. In Figure A-3, the inner loop contains a loop-carried dependence on `accum`, so the compiler will first consider the outer loop to see if it can be parallelized.

Loop Transformation

To determine if a loop can be transformed, the compiler employs a modified version of the approach used for vector architectures by Allen and Kennedy [AK01]. The compiler first conducts a dependence analysis. Since a loop may contain internal control flow, the compiler processes each possible control flow path through the loop and updates the dependence graph accordingly. The SUIF-generated direction vectors for memory dependences are used to provide additional information such as potential cross-VP transfers. Once the dependence graph is constructed, the compiler uses Tarjan's algorithm to identify strongly connected components [Tar72]. If there are any dependence cycles that occur in the loop currently being considered, it will not be transformed unless the cycle involves register dependences contained within an inner loop (as those will be local to each VP), or unless cross-VP communication is possible. Another requirement is that all memory operations that will potentially be transformed to vector memory instructions—i.e. those in the loop's header block or back edge block—must have a statically determinable stride value.

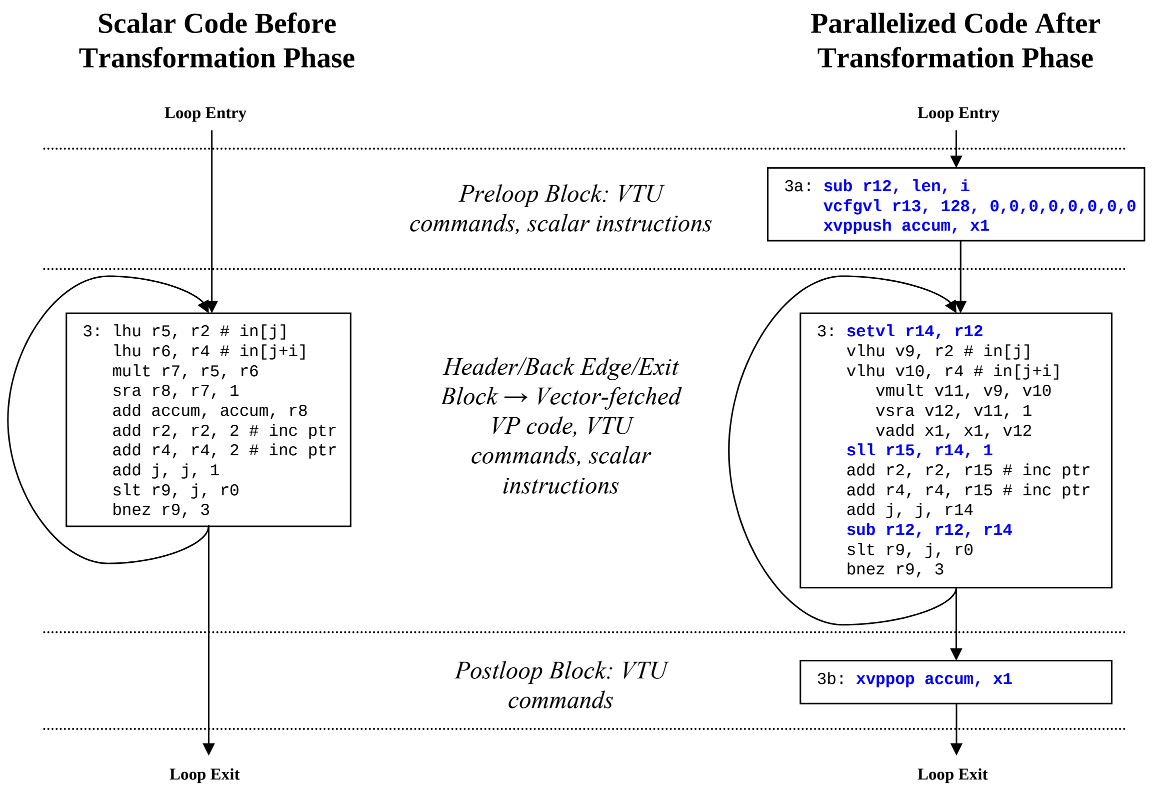
If the compiler determines that a loop meets the requirements for transformation, it sets up the code necessary for the scalar processor to control the execution of the loop. It creates a special preloop basic block that contains the configuration command for the VTU. This is block 0 in Figure A-3's parallelized code. The command is updated after the chain register insertion phase with the actual configuration identifying the number of registers required in each cluster. The preloop block also sets up loop-invariant values, including constants, registers that are never written, and AIB addresses needed by VP fetch instructions. All of these values are written into shared registers using the `vwrsh` command. (For clarity, Figure A-3 shows fetch instructions using the block labels rather than actual registers.) Besides the performance benefit of avoiding the computation of these values in each iteration, mapping the values to shared registers that can be used by multiple VPs also reduces the per-VP physical register requirements.

Another function of the preloop block is to load each VP's number—found within a special data block containing the numbers 0 through 127—into a private register if necessary. This loading occurs if an induction variable will be used in an internal loop block, and each VP has to compute the value of that variable. For example, the value of `mask[i]` is loaded in block 4 of Figure A-3. Since block 4 is an internal loop block, a VP load instruction will be used, and the address register `v2` has to be set up by the control processor. The initialization of `v2` is performed in block 1 of each strip-mined loop iteration by having each VP take the baseline value of the `mask` address for that iteration—i.e. the value needed by VP 0—and adding its own VP number.

Block 1 also contains the `setvl` command to set the vector length in each strip-mined iteration. The induction operations for the loop (found in block 6) are updated so that the induction value is multiplied by the vector length. Note that the loop counter increments from 0 to the trip count `len`. However, decrementing from `len` to 0 would be more appropriate when using the `setvl` instruction, which takes the number of iterations remaining as its source operand. In the example, the compiler adds another variable `r12` which decrements to 0, while keeping the separate counter `i` which increments to `len`. While this approach is simple, optimized code would merge the two variables.

There is generally a one-to-one mapping between scalar instructions and VP instructions, so transforming the code is mostly straightforward. Note that most of the opcode names in the parallelized code are the same as their scalar counterparts with a "`v`" added as a prefix to indicate a vector instruction. VP memory instructions have a "`vp`" prefix to distinguish them from vector memory commands. Branches are somewhat different for VPs than for a typical scalar processor. When a VP finishes executing a block of code, it will terminate unless it fetches another block. There is no notion of "falling through" to the next block of code. Thus, in block 2, an explicit `fetch` instruction is added to fetch block 3. In block 5, since there is no fetch instruction, the thread will terminate and control returns to the control processor. To conditionally fetch a block, the predicate register has to be used. Each VP has a single private predicate register that can be used for control flow and for if-conversion [AKPW83]. Block 1 shows how a scalar conditional branch instruction is converted to a `psel.fetch` instruction which selects between two different blocks to fetch. In block 4, if the predicate is false, no fetch will occur and the VP will terminate.

Register values that are live within a single iteration are unique to each VP, so they are mapped to private registers. Cross-VP values are mapped to a "dummy" register file, and are later converted to `prevVP`/`nextVP` names in the assembly code that correspond to the queues between each lane. The reason for using registers for these values is to ensure that the compiler can easily track the dependences between receive/send operation pairs. Figure A-4 illustrates the use of cross-VP transfers by parallelizing the inner loop of Figure A-2 instead of the outer loop. The `x1` cross-VP "register" is used to hold the value of `accum` as it is passed from VP to VP. When generating code with cross-VP transfers, the compiler also sets up `xvppush` VTU commands in the preloop block for the control processor to push initial cross-VP values into the queue, and creates a postloop block containing `xvppop` commands that pop final cross-VP values from the queue after the loop ends.


Scalar Code Before Transformation Phase
Parallelized Code After Transformation Phase
Loop Entry
Loop Entry
Preloop Block: VTU commands, scalar instructions
Header/Back Edge/Exit Block → Vector-fetched VP code, VTU commands, scalar instructions
Postloop Block: VTU commands
Loop Exit
Loop Exit

```
3: lhu r5, r2 # in[j]
   lhu r6, r4 # in[j+i]
   mult r7, r5, r6
   sra r8, r7, 1
   add accum, accum, r8
   add r2, r2, 2 # inc ptr
   add r4, r4, 2 # inc ptr
   add j, j, 1
   slt r9, j, r0
   bnez r9, 3
```

```
3a: sub r12, len, i
    vcfgvl r13, 128, 0,0,0,0,0,0,0,0
    xvppush accum, x1
```

```
3: setvl r14, r12
   vlhu v9, r2 # in[j]
   vlhu v10, r4 # in[j+i]
      vmult v11, v9, v10
      vsra v12, v11, 1
      vadd x1, x1, v12
   sll r15, r14, 1
   add r2, r2, r15 # inc ptr
   add r4, r4, r15 # inc ptr
   add j, j, r14
   sub r12, r12, r14
   slt r9, j, r0
   bnez r9, 3
```

```
3b: xvppop accum, x1
```

Figure A-4: The inner loop of Figure A-2 is parallelized in this example. Since the loop contains a single basic block, vector-fetched VP code and vector memory instructions are used throughout the parallelized loop. The “x” register prefix in the parallelized code indicates a value that will be mapped to the cross-VP network. The `vadd` instruction in block 3 obtains a value from the previous VP and writes its result to the next VP; the use of `x1` will be later replaced by `prevVP`/`nextVP` names.

A.1.3 Cluster Assignment

The compiler is responsible for assigning each VP instruction to a particular cluster. It uses a modified version of the clustering approach described in [CFM03], which targets a typical VLIW architecture and tries to balance work among the clusters while still avoiding inter-cluster moves along the critical path. For a VLIW compiler, the focus is on improving ILP for a single thread. By contrast, Scale also exploits DLP and TLP, and although reducing per-thread latency is important in the compiler, it is typically more important to focus on the other forms of parallelism to improve processor throughput—thus reducing the overall latency of the program. Since the vector length depends on the cluster with the most severe register requirements, the Scale compiler prioritizes achieving a more balanced partitioning of instructions between clusters over minimizing the per-thread latency. Even when inter-cluster communication is required, the fact that Scale VP instructions are converted into implementation-specific micro-operations, or *micro-ops*, helps to alleviate any communication overhead. Micro-ops allow the compiler to target multiple clusters with a single VP instruction, avoiding the need to insert an explicit copy operation for each inter-cluster transfer. As part of the cluster assignment phase, the compiler modifies the VP instructions to write multiple destinations when necessary.

Additionally, to enhance decoupling—which also serves to hide latency—the compiler attempts to generate acyclic dataflow between the clusters, as a specific cluster's operations within an AIB have to be executed as an atomic group for each VP. The compiler also tries to place long-latency operations on clusters separate from the instructions which depend on the results. This is intended to take advantage of the decoupling between clusters. For example, if a multiply feeds an add within a single AIB, placing the multiply and add on separate clusters will allow the multiply latency to be hidden by the interleaved execution of independent VPs.

Balancing all of these constraints effectively is a difficult challenge, so to avoid spending a great deal of time focusing on optimizing the cluster assignment phase, I use simple heuristics during clustering. For example, to reduce the possibility of acyclic dataflow, the compiler biases the initial cluster assignment as well as the desirability metrics in an attempt to have cluster dataflow move in a single direction. Similarly, if a block contains a long-latency instruction such as a multiply, which executes on cluster 3, the compiler tries to avoid mapping any non-multiplication instructions to cluster 3 so that the cluster's execution resources can be fully devoted to processing long-latency instructions for each VP. While the approach of using simple heuristics can be beneficial for various codes, it also leaves significant room for improvement.

Figure A-5 compares the baseline cluster assignment method—i.e. the approach discussed in [CFM03]—with the modified clustering that accounts for the Scale-specific constraints described above. The first five instructions in each block are on the critical path. As shown in Figure A-5(a), the default cluster assignment keeps the dependence chain for the `vmult` instruction grouped together on cluster 3 to avoid inter-cluster moves which could lengthen the critical path. Additionally, the pointer increments are also mapped to cluster 3, leading to an imbalanced assignment. Figure A-5(b) shows how Scale-specific clustering affects the assignment. The dependence chain for the `vmult` instruction is now broken apart so that the following `vsra` and `vadd` instructions are mapped to cluster 2; this assignment is designed to permit cluster 3 to be solely devoted to processing multiply instructions. However, all of the remaining `vadd` instructions and the `vslt` instruction are also mapped to cluster 2. There are two primary reasons for this imbalanced assignment. First, cluster 2 is the only cluster that does not execute any long-latency instructions (cluster 0 executes memory instructions, cluster 1 executes thread-fetch instructions which take multiple cycles to complete, and cluster 3 executes multiply/divide instructions); as a result, the current set of heuristics tends to be biased toward placing single-cycle latency instructions on cluster 2. Second, the compiler's heuristic that attempts to avoid acyclic dataflow is somewhat simplistic. As it tests different cluster assignments, it tries to anticipate the possibility of a later iteration of the clustering algorithm reassigning an instruction and introducing a cycle, so it tends to pack instructions on one side of the cluster structure (i.e. toward cluster 3), and then forces any reassignments to move instructions in a single direction (i.e. toward cluster 0). While this approach can be effective at times, it sometimes fails to redistribute the instructions in a balanced manner. Improving the cluster assignment phase of the Scale compiler is an important component of future work.

```
3:    c0 vplhu (v5) -> c3/v9 # in[j]
      c0 vplhu (v7) -> c3/v10 # in[j+i]
      c3 vmult v9, v10 -> c3/v11
      c3 vsra v11, 1 -> c3/v12
      c3 vadd v3, v12 -> c3/v3
      c3 vadd v14, 2 -> c3/v14, c0/v5 # inc ptr
      c3 vadd v15, 2 -> c3/v15, c0/v7 # inc ptr
      c1 vadd v8, 1 -> c1/v8 # inc j
      c1 vslt v8, v4 -> c1/p # j < len-i?
      c1 psel.fetch 4, 3
```

(a)

```
3:    c0 vplhu (v5) -> c3/v9 # in[j]
      c0 vplhu (v7) -> c3/v10 # in[j+i]
      c3 vmult v9, v10 -> c2/v11
      c2 vsra v11, 1 -> c2/v12
      c2 vadd v3, v12 -> c2/v3
      c2 vadd v14, 2 -> c2/v14, c0/v5 # inc ptr
      c2 vadd v15, 2 -> c2/v15, c0/v7 # inc ptr
      c2 vadd v8, 1 -> c2/v8 # inc j
      c2 vslt v8, v4 -> c1/p # j < len-i?
      c1 psel.fetch 4, 3
```

(b)

Figure A-5: (a) Baseline cluster assignment for block 3 of the parallelized code in Figure A-3. No Scale-specific optimizations are performed. The code format has been changed to resemble the final Scale assembly code, as this makes it easier to represent the targeting of multiple destinations. For example, cluster 3 is used to increment the pointer value contained in `c0/v5`; it does this by keeping its own copy of the register in `c3/v14` and incrementing both registers. (b) Cluster assignment for block 3 with Scale-specific optimizations enabled.

A.1.4 Instruction Scheduling

Typical instruction scheduling approaches try to reduce the critical path length of the section of code being analyzed. This frequently leads to long-latency instructions being scheduled early, while short-latency dependent instructions are scheduled later. As a result, register lifetimes are often lengthened by the scheduler, leading to increased register usage. By contrast, the primary goal of Scale's instruction scheduler is to minimize register usage by scheduling dependence chains together. This is done for two reasons. First, by reducing the number of registers used by each VP, the maximum vector length may increase, enabling greater processor throughput. Second, scheduling dependence chains together makes it more likely that the compiler will be able to explicitly target chain registers—the inputs to the ALU—potentially further reducing the register usage for each VP and also enabling more energy-efficient execution. As mentioned earlier, long-latency operations are usually placed on a different cluster than any dependent instructions, so Scale's use of decoupling helps to hide the latency from targeting dependence chains in this phase.

I modified the traditional list scheduler used in Trimaran to attempt to group dependence chains together, and if no chaining opportunities exist, to revert to its standard approach. I also attempted to account for additional factors, such as the possibility that a VP instruction may target multiple clusters, as then there may be multiple opportunities to use chain registers. However, similarly to the cluster assignment phase, the compiler's scheduling heuristics sometimes fail to achieve their goals. An additional complication is caused by the fact that my modifications are primarily restricted to the priority queue class used to select the next instruction to schedule. The problem with this approach is that an instruction can be selected from the priority queue later than another instruction and yet have an earlier scheduling time, as Trimaran's machine description encodes parallel execution resources in the processor. Thus, the desired dependence chains might be broken up, depending on the actual scheduling times of particular instructions.

Figure A-6 presents a comparison of the standard Trimaran scheduling approach and the Scale-specific attempt to schedule dependence chains together. In Figure A-6(a), which shows the result of the default approach, the reason for Trimaran's late scheduling of the second load instruction is unclear: in the first prepass scheduling phase, the load is scheduled earlier, but in the postpass phase the scheduling time is changed. Regardless, no particular attempt is made to schedule dependence chains together. However, two chains are grouped by default: the `vadd`/`vslt` pair near the top of the block; and the `vmult`/`vsra`/`vadd` trio near the bottom of the block. Figure A-6(b) shows the result when the Scale-specific optimizations are employed. As can be seen in the figure, only the `vsra`/`vadd` pair is grouped together (although it is possible for instructions in a dependence chain to use chain registers if they are separated by instructions on different clusters). When the instructions are actually being scheduled, dependence chains are processed together—e.g. the two loads, the multiply, the shift, and the dependent add are the first five instructions scheduled in the prepass phase. However, once the scheduling times are actually assigned to all the instructions, the ordering changes—e.g. the first `vadd` is independent of the first `vplhu`, and is assigned the same scheduling time, so it is inserted between the loads. Addressing this issue is deferred for future work.

```
3:    c0 vplhu (v5) -> c3/v9 # in[j]
      c2 vadd v8, 1 -> c2/v8 # inc j
      c2 vslt v8, v4 -> c1/p # j < len-i?
      c2 vadd v14, 2 -> c2/v14, c0/v5 # inc ptr
      c0 vplhu (v7) -> c3/v10 # in[j+i]
      c2 vadd v15, 2 -> c2/v15, c0/v7 # inc ptr
      c3 vmult v9, v10 -> c2/v11
      c2 vsra v11, 1 -> c2/v12
      c2 vadd v3, v12 -> c2/v3
      c1 psel.fetch 4, 3
```

(a)

```
3:    c0 vplhu (v5) -> c3/v9 # in[j]
      c2 vadd v8, 1 -> c2/v8 # inc j
      c0 vplhu (v7) -> c3/v10 # in[j+i]
      c2 vslt v8, v4 -> c1/p # j < len-i?
      c3 vmult v9, v10 -> c2/v11
      c2 vadd v14, 2 -> c2/v14, c0/v5 # inc ptr
      c2 vsra v11, 1 -> c2/v12
      c2 vadd v3, v12 -> c2/v3
      c2 vadd v15, 2 -> c2/v15, c0/v7 # inc ptr
      c1 psel.fetch 4, 3
```

(b)

Figure A-6: (a) Default output of Trimaran's postpass scheduling phase for the code of Figure A-5(b). No attempt is made to schedule dependence chains together. To simplify the comparison between the two blocks of code which have different physical register allocations, virtual register numbers are still used for instruction operands. (b) Output of the postpass scheduling phase when the scheduler attempts to group dependence chains. The fact that the scheduler modifications are restricted to Trimaran's priority queue class can result in some chains being broken up.

A.1.5 AIB Formation

In this phase, the compiler processes each basic block and inserts directives around VP instructions that delimit the boundaries of AIBs. As mentioned previously, the actual separation of AIBs into a separate section of the file takes place during assembly code generation. There are several situations which will cause an AIB to be ended. First, for each VP instruction the compiler generates the corresponding micro-ops to determine whether the AIB size limit will be exceeded. If the size limit is reached, the AIB will be ended. Another reason to end an AIB is if a control processor instruction is encountered that has a dependence on one of the VP instructions in the current AIB. Finally, the end of a basic block will also terminate an AIB.

Although there are advantages to forming AIBs later in the compilation process, one problem caused by the current phase ordering is the possibility of creating cross-VP transfers that are later determined to span more than one AIB. Cross-VP communication is designed to be used primarily within a single vector-fetched AIB to avoid the possibility of deadlock. When AIB formation occurs, if a vector-fetched AIB containing cross-VP transfers is discovered to be too large, it is possible that a prevVP receive will be placed in a different AIB than its corresponding nextVP send, possibly causing deadlock due to the fact that VP execution is time-multiplexed across the lanes. Currently, this problem is not an issue, as an AIB with reasonably balanced cluster assignment can hold on the order of 100 instructions, and since the compiler performs no loop unrolling or if-conversion, basic blocks are typically not very large. However, if more aggressive compilation techniques are employed, this phase ordering may have to be revisited.

A.1.6 Chain Register Insertion

Once AIBs are formed, the compiler can map temporary values to chain registers, which are only valid within an AIB. The use of chain registers can reduce the physical register file resources required by each VP and thus potentially increase the vector length. To determine which values can be mapped to chain registers, the compiler constructs the live ranges for each register value, keeping a list of potential candidates. A value with a live range that crosses an AIB boundary cannot be mapped to a chain register. Also, every operation that executes on a particular cluster overwrites the values in that cluster's chain registers. If a potential chain register value would not be overwritten by another operation during its lifetime, it will be mapped to a chain register. Figure A-7 illustrates the use of chain registers.

Note that it is possible for VP register spill code to be generated when first running register allocation. Performing chain register insertion during the initial allocation phase could reduce register pressure and lessen the possibility of registers being spilled. However, chain register insertion has to take place after AIB formation, and forming AIBs before running the register allocator for the first time could create a problem if spill code causes the AIB to overflow its size limit. An alternative strategy employed by the TRIPS compiler is to form blocks before register allocation, but to use iterative block splitting if spill code insertion causes a block size violation [SBG$^+$06].

```
3:    AIB_BEGIN
      c0 vplhu (pr0) -> c3/pr2 # in[j]
      c2 vadd pr0, 1 -> c2/pr0 # inc j
      c0 vplhu (pr1) -> c3/pr0 # in[j+i]
      c2 vslt pr0, pr5 -> c1/p # j < len-i?
      c3 vmult pr2, pr0 -> c2/pr4
      c2 vadd pr1, 2 -> c2/pr1, c0/pr0 # inc ptr
      c2 vsra pr4, 1 -> c2/pr4
      c2 vadd pr3, pr4 -> c2/pr3
      c2 vadd pr2, 2 -> c2/pr2, c0/pr1 # inc ptr
      c1 psel.fetch 4, 3
      AIB_END
```

$\longrightarrow$

```
3:    AIB_BEGIN
      c0 vplhu (pr0) -> c3/cr0 # in[j]
      c2 vadd pr0, 1 -> c2/pr0 # inc j
      c0 vplhu (pr1) -> c3/cr1 # in[j+i]
      c2 vslt pr0, pr5 -> c1/p # j < len-i?
      c3 vmult cr0, cr1 -> c2/pr4
      c2 vadd pr1, 2 -> c2/pr1, c0/pr0 # inc ptr
      c2 vsra pr4, 1 -> c2/cr1
      c2 vadd pr3, cr1 -> c2/pr3
      c2 vadd pr2, 2 -> c2/pr2, c0/pr1 # inc ptr
      c1 psel.fetch 4, 3
      AIB_END
```

Figure A-7: On the left is the code of Figure A-6(b) after AIB formation, and using physical register specifiers. On the right is the code after chain register insertion.

A.2 Compiler Evaluation

I evaluate the Scale compiler implementation by using benchmarks from the EEMBC benchmark suite. Table A.1 lists the benchmarks that the compiler is currently able to handle as well as the types of loops that were parallelized during compilation; the actual benchmark descriptions were previously provided in Table 8.1. The autocorrelation benchmark contains a loop nest in which the inner loop has a cross-iteration dependence. By default, the compiler parallelizes it using outer-loop vectorization (listed as `autocor_olv` in the table). However, for the sake of comparison, I also configured the compiler to parallelize the inner loop by using Scale's cross-VP network (`autocor_xvp`). Aside from inserting the `restrict` keyword to indicate that pointers do not alias, the only other modifications to the source code involved changing the loops for `rgbcmy` and `rgbyiq` to use array accesses instead of pointer accesses. This is due to a temporary restriction that I imposed early in the compiler development that an induction variable could only be updated once within a loop iteration, as this simplified the computations of strides for vector memory accesses. Since `rgbcmy` and `rgbyiq` contain pointer variables that are incremented multiple times within the loop, that inhibits parallelization within the current compiler setup. Lifting that restriction in future work will enable the compilation of code without having to make any pointer-to-array conversions.

Since the compiler infrastructure is still early in its development, the total number of EEMBC benchmarks that it can handle is somewhat small. The primary focus in this work was on mapping the compiler's intermediate representation to the vector-thread architectural features, not on obtaining a vectorizable intermediate form from the original source code. Incorporating more established front-end loop transformations would enable a broader coverage of benchmarks. Also, the compiler currently has certain artificial restrictions in place that were inserted to ease the development and debugging process; lifting those restrictions will enable the compiler to handle more programs. Despite the limited number of benchmarks, a key point is that they represent a wide variety of loop types, including loops that would be non-vectorizable without significant transformations. It should be noted that I only include benchmarks that can be automatically parallelized, with the exceptions of the modifications discussed above. Previous work [KVN$^+$06, IBE$^+$07] has shown the difficulties of automatically parallelizing EEMBC benchmarks, using Intel's compiler in an evaluation. The Intel compiler is unable to vectorize many of the innermost loops in EEMBC benchmarks for various reasons (although the use of the `restrict` keyword would likely help in certain cases). It can alternatively target outer loops, but this requires user assistance, such as OpenMP directives. Although this is only a single example, it serves to illustrate the point that the problems encountered in automatically parallelizing the embedded benchmarks targeted in this work are not unique to the Scale compiler.

The compiler-generated code is evaluated on the most recent version of the Scale microarchitectural simulator, using the default four-lane configuration. Table A.1 contains the speedups of the compiler-generated parallelized code that uses the VTU when compared to the scalar code that only uses the control processor. The scalar code is generated using the same compilation infrastructure described in Section A.1, but without performing any scalar-to-VP code transformations. The fact that control processor pipeline hazards are not modeled when simulating baseline scalar code provides a pessimistic evaluation of the parallelized code speedups.

The smallest speedup is about 3× for `fir`. This benchmark contains a loop nest with an inner loop that only consists of two accumulations. Since the compiler parallelizes the

				Per-Cycle Statistics		
Benchmark	Loop Type	Avg Vector Length	Kernel Speedup	Compute Ops	Load Elms	Store Elms
autocor_olv	DI	16.6	7.3	6.5	1.3	0.0
autocor_xvp	XI	60.6	10.1	2.7	1.8	0.0
fir	XI	35.0	3.3	0.4	0.4	0.0
hpg	DP	63.6	33.0	4.0	2.0	0.2
rgbcmy	DC	20.0	7.2	2.9	0.5	0.7
rgbyiq	DP	28.0	26.0	6.0	0.9	0.9

Table A.1: List of benchmarks and types of loops that were parallelized, average vector length, speedups over scalar code, and the average numbers of VTU compute ops, load elements, and store elements per cycle. The different loop types are taken from [KBH+04a]: [DP] data-parallel loop with no control flow, [DC] data-parallel loop with conditional thread-fetches, [XI] loop with cross-iteration dependences, [DI] data-parallel loop with inner loop. For `autocor`, the `data3` dataset was used.

inner loop using cross-VP transfers, there is not much computation to perform in parallel. Additionally, two of the four VP instructions in the parallelized benchmark are long-latency multiplies, further limiting the speedup. At the other extreme, `hpg` has a large speedup of 33×. However, there is one caveat with this result. In general, I observed that when generating scalar code, the compiler infrastructure used in this appendix (SUIF-to-Trimaran-to-GCC) is competitive with the GCC-only approach used for the hand-coded benchmarks described in Chapter 7, and in some instances is superior. However, for `hpg`, the Scale compiler generates code that contains a significant number of register spills, and this code is slower by about a factor of 3 than using GCC alone. For the sake of consistency, I used the same baseline compiler infrastructure for all benchmarks, but even when compared against the faster GCC-generated scalar code, the compiler still achieves about an 11× speedup for `hpg`. `rgbyiq` has a speedup of about 26× regardless of the baseline scalar code used. This is a simple DOALL loop that the compiler can handle in a straightforward manner. For `autocor`, the use of the cross-VP network produces a superior speedup to outer-loop vectorization. This is not inherent to the program—for the hand-coded assembly, the outer-loop vectorized version is superior—but is due to the fact that the compiler infrastructure is still very inefficient when generating code with thread fetches. By contrast, the inner-loop vectorized version of `autocor` that uses the cross-VP network only has a single basic block to parallelize, which is simpler for the compiler to handle. As the compiler is further developed and other optimizations such as if-conversion are integrated, the efficiency of outer-loop vectorization should increase.

An interesting comparison can be made between the results for Scale and the limit study on thread-level speculation conducted by Islam et al. [IBE+07]. TLS designs are somewhat more flexible than Scale in the types of codes they can handle, as they typically have hardware support to squash speculative threads that have violated dependences. The Scale compiler focuses on non-speculative execution and is therefore conservative in its approach to handling dependences between threads. In [IBE+07], the speedup over a single-issue scalar processor is computed for a variety of multicore TLS configurations. The benchmarks consist of the EEMBC consumer and telecom suites, which includes `autocor`, `hpg`, `rgbcmy`, and `rgbyiq`. For an ideal TLS machine with an infinite number of cores and zero thread-management overhead, the speedups are approximately 5× for `autocor`, 19× for `hpg`, 14× for `rgbcmy`, and 17× for `rgbyiq`. Note that for `autocor` and `rgbyiq`, the actual speedups achieved by the Scale compiler using a realistic model for the vector-thread unit exceed the upper bound on the TLS speedups. (I do not include `hpg` for the reason mentioned earlier.) For a 16-core machine with no thread-management overhead (which matches the issue capability of Scale's VTU but is idealized with respect to latency), the speedups are approximately 5× for `autocor`, 9× for `hpg`, 8× for `rgbcmy`, and 9× for `rgbyiq`. Obviously the setup used for the Scale compiler evaluation is somewhat different; still, this comparison attests to the performance improvements made possible by various Scale features, such as vector memory instructions and decoupled execution. The last three columns of Table A.1 also indicate the effectiveness of Scale's features, as they show that the total number of VTU compute operations and memory elements processed per cycle is substantially less than the number of operations per cycle that would be executed in an idealized 16-core machine. The Scale chip prototype [Kra07] provides evidence that a 16-issue VT architecture has much lower area and power consumption than a 16-core multiprocessor.

Figure A-8 shows how the optimizations performed in various compiler phases affect the speedups. "No Opt" refers to simply performing the scalar-to-VP code transformation: the clustering used is the approach described in [CFM03] with none of the Scale-specific modifications described in Section A.1.3. The standard Trimaran list scheduler is used with no attempt to schedule dependence chains together, and no chain registers are used. I then enable the various optimizations to determine their impact. In certain cases, turning on optimizations does not affect the average vector length, as shown in Figure A-9, and thus performance is relatively unchanged. For some benchmarks, even when the vector length is increased, it does not help performance. For example, although scheduling dependence chains together doubles the vector length for `fir`, it does not increase resource utilization, as the baseline code already spawns a sufficient number of VPs to maximize performance for a particular cluster assignment.

Performing Scale-specific clustering provides significant benefits for some benchmarks, and is inconsequential in others. As discussed in Section A.1.3, the set of heuristics used in this phase leave significant room for improvement; as a result, the compiler sometimes packed the majority of operations within a single cluster, thus inhibiting decoupling. Dependence chain scheduling provides small benefits for `rgbcmy` and `rgbyiq`, but has a negative impact on `hpg` in spite of the fact that it increases the average vector length. (I have not yet pinpointed the reason for this decrease, although `hpg` has a more complex memory access pattern than the other benchmarks, so that could be a factor.) However, for the remaining benchmarks, dependence chain scheduling had little effect, either due to no effect on the vector length, or due to resource utilization already being maximized as mentioned above. Chain register insertion also did not typically affect the vector length for the benchmarks which were evaluated, resulting in little or no impact on performance. This lack of effectiveness was due to the fact that for the codes under consideration, the use of chain registers did not reduce the number of general registers needed for the "critical" cluster that determined the vector length. However, as shown in Table A.2 there were multiple cases in which a large percentage of VP register accesses were mapped to chain registers. Although chain register insertion did not improve performance in this evaluation, it is still useful to save energy. One interesting observation from Table A.2 is that chain register usage decreases for `rgbyiq` when trying to schedule dependence chains together. Each iteration of the `rgbyiq` loop loads the three elements of an input pixel, performs a set of conversions, and stores three elements to the output array. Since each output element depends on all three input elements, the dependence chains are interconnected. However, when the compiler attempts to group dependence chains, each vector load is immediately followed by a snippet of dependent computation that is placed in its own AIB. By contrast, the default scheduling approach first issues all three vector-loads, and then issues a single AIB that contains all of the dependent computation. Since chain registers cannot be live across AIB boundaries, having a single large AIB turns out to be more effective for `rgbyiq` in terms of increasing chain register usage. This observation points to the need to be more selective in the instruction scheduling phase when targeting dependence chains. Another interesting statistic is the decrease in chain register usage for `hpg` when Scale-specific clustering is turned on (the "Full" set of optimizations) versus just enabling dependence chain scheduling and chain register insertion. Overall, not only do cluster assignment and instruction scheduling impact each other's effectiveness, but they also impact the effectiveness of chain register insertion. Future refinements to these phases should improve chain register usage, hopefully leading to longer vector lengths and higher performance.

Figure A-10 provides a comparison between the speedups of the compiler-generated code and the speedups of the handwritten assembly code. This comparison shows that there is still significant room for improvement in the compiler infrastructure. The hand-coded benchmarks have been highly optimized and take advantage of algorithmic transformations as well as Scale features that the compiler does not yet support such as segment-strided memory accesses. Further development of the compiler should narrow the performance gap. Additionally, the algorithm changes used in the hand-coded benchmarks could also be applied to the compiler's input source code. However, for this particular evaluation I wanted to minimize programmer intervention as much as possible, so I left the overall structure of the source code unchanged.

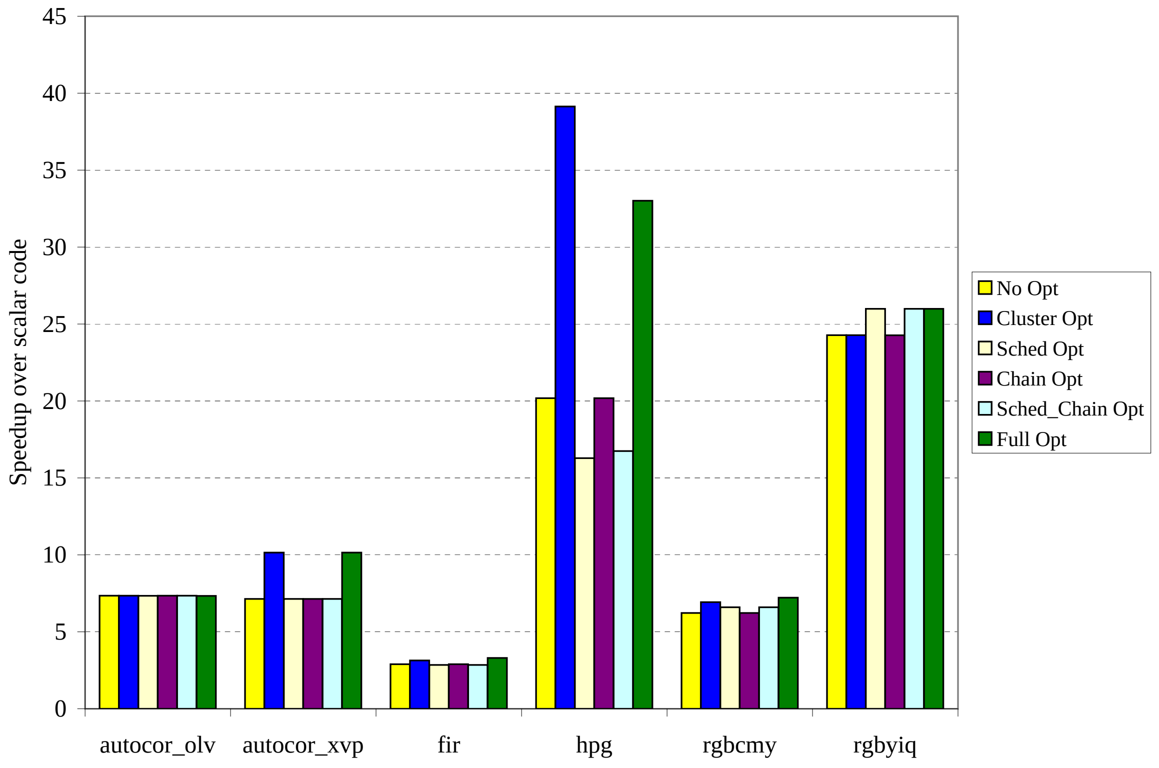


Figure A-8: Comparison of the various speedups of the compiler-generated code when different optimizations are turned on. No Opt = No optimizations; Cluster Opt = Scale-specific clustering; Sched Opt = Schedule dependence chains together; Chain Opt = Target chain registers; Sched_Chain Opt = Schedule dependence chains together and target chain registers; Full Opt = Turn on all optimizations.

	% Register Reads			% Register Writes		
Benchmark	**Chain**	**Sched_Chain**	**Full**	**Chain**	**Sched_Chain**	**Full**
autocor_olv	10.5	10.5	15.8	16.6	16.6	24.9
autocor_crossvp	33.4	33.4	33.3	49.9	49.9	49.9
fir	0.7	25.5	25.3	0.9	34.0	33.6
hpg	5.4	13.5	8.1	7.4	18.5	11.1
rgbcmy	0.0	0.0	0.0	0.0	0.0	0.0
rgbyiq	37.8	28.9	28.9	70.8	54.2	54.2

Table A.2: Percentage of VP register reads/writes that are mapped to chain registers when enabling different compiler optimizations: "Chain" = only chain register insertion is enabled with no Scale-specific clustering or scheduling; "Sched_Chain" = both dependence chain scheduling and chain register insertion are enabled; "Full" = all Scale-specific optimizations are enabled.

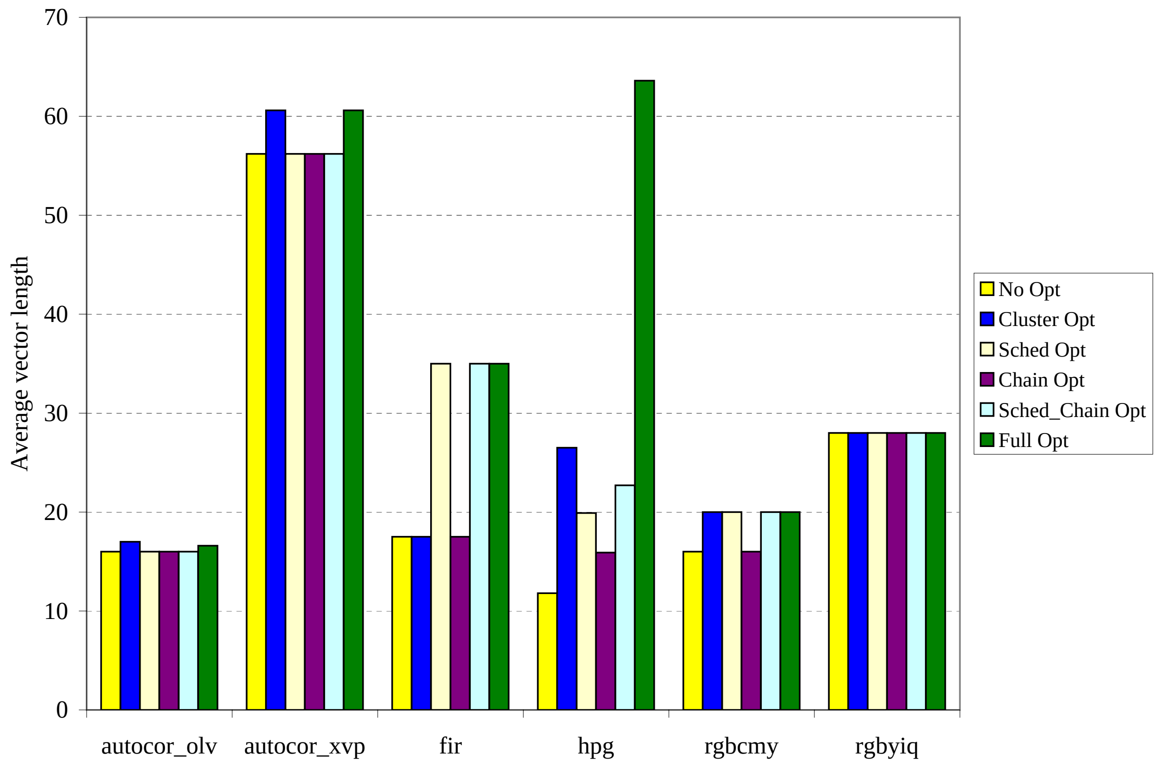


Figure A-9: Comparison of the average vector lengths when different optimizations are turned on.

A.3 Related Work

The TRIPS architecture [SNL+03] can also exploit multiple forms of parallelism. However, unlike Scale, which exploits DLP, TLP, and ILP simultaneously, TRIPS only targets one form of parallelism at any given time and explicitly "morphs" between modes of execution. Speculation is used to find parallelism. The TRIPS compiler focuses on forming blocks full of useful instructions [SBG+06] and mapping instructions to ALUs [NKB+04, CCK+06]. The Scale compiler has somewhat different priorities, as it deals with issues more typically encountered in vectorization, and it also focuses on non-speculative execution.

Stream processing [KDK+01] targets DLP by mapping computation kernels—typically the body of a loop—to a Kernel Exection Unit (KEU), which is a co-processor that contains clusters of ALUs. ILP can also be exploited within each kernel. While this approach shares some similarities with Scale, there are some significant differences between the Scale compiler and the stream processing compiler [DDM06]. The KEU can only communicate with memory through a Stream Register File (SRF). A major job of the compiler is to manage the utilization of the SRF. By contrast, this is much less of a concern for the Scale compiler due to Scale's cached shared memory model and decoupled cache refills [BKGA04]. Additionally, the stream processing compiler performs a binary search to determine the best strip-size when strip mining, while this is not an issue for Scale. Finally, the stream

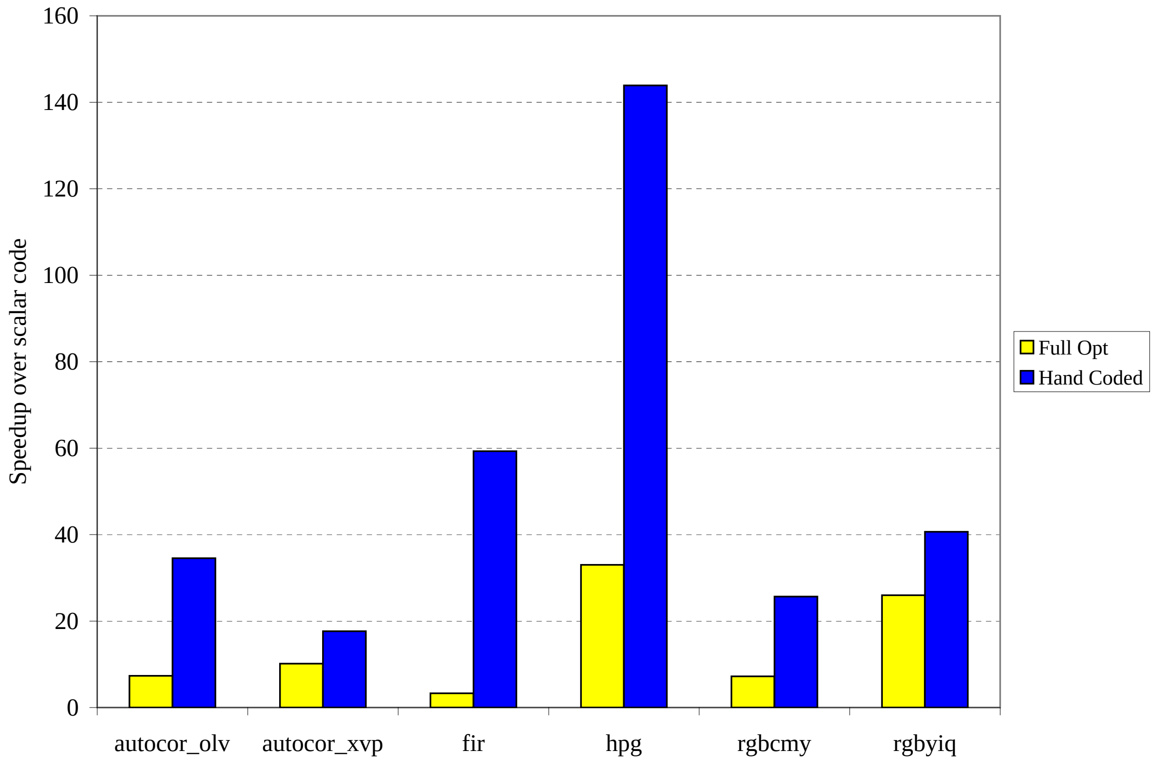


Figure A-10: Comparison of the speedups of the compiler-generated code with all optimizations turned on and the hand-coded assembly.

processing compiler uses a standard list scheduling approach to reduce latency of a kernel, while the Scale compiler focuses more on improving throughput as the Scale hardware provides decoupling to tolerate latency.

Eichenberger et al. [EOO$^+$05] describe the compiler for the Cell processor, which exploits multiple forms of parallelism. They provide multiple programming models for Cell depending on how involved the programmer wants to be with the low-level details of the architecture. The paper focuses on many issues which are not present in Scale: Cell uses SIMD functional units for data-level parallelism, so alignment is a significant concern; the Synergistic Processor Elements (SPEs) have local non-coherent memories, so the compiler has to handle transfers between the system memory and the local memories using DMA commands; the SPEs have no branch prediction and assume all branches to be not-taken, so the compiler has to insert branch hints and schedule them appropriately. The Scale compiler has different priorities due to the significantly different architecture.

The XIMD architecture [WS91] extends a VLIW architecture by providing an instruction sequencer for each functional unit. This allows the XIMD to behave like a multithreaded machine, while still retaining the ability to function like a VLIW. The compiler schedules loops for the XIMD by using a technique called *iteration mapping* [NHS93], which attempts to balance the exploitation of fine- and medium-grained parallelism in order to fully utilize processor resources. Since Scale hardware time-multiplexes VP threads onto the physical resources and makes extensive use of decoupling, there is less of a burden on the compiler to fully utilize resources, although the cluster assignment phase can have a significant impact.

Microthreading [Jes01] shares some similarities with VT. A microthread executes a single loop iteration and its execution can be interleaved with that of other microthreads sharing the same physical resources. The compilation infrastructure required for microthreading is discussed in [BBdG^{+}06], although no actual compiler implementation is presented. It is the compiler's responsibility to insert *swch* commands to force a context switch after issuing loads which could miss in the cache. This is done to prevent potential stalls. The Scale compiler aims for a similar goal by attempting to place VP operations which depend on loads on a different cluster than cluster 0, so that decoupling will be possible.

Multi-threaded vectorization [Chi91] tries to span the gap between vector and VLIW machines in order to target code that would not be traditionally vectorizable. The compiler starts with a schedule that has been software-pipelined and builds on that to determine the issue time and issue period for each vectorized instruction. However, the compiler has to know the functional unit latencies, and the work is restricted to a single vector lane.

Tian et al. [TBGG03] exploit parallelism at both the thread-level and instruction-level on an Intel platform that supports Hyper-Threading Technology. The Intel compiler can perform parallelization that is guided by OpenMP directives or pragmas. It also supports automatic loop multithreading to exploit medium-grained parallelism. For automatic parallelization, the compiler builds a loop hierarchy structure, performs data dependence analysis to find loops without loop-carried dependences, and determines whether it will be profitable to generate multithreaded code for the loop. The compiler can also perform "intra-register vectorization" by generating SIMD instructions. This technique can be combined with the above multithreading approaches. Scale's advantages include the ability to handle loop-carried dependences, and more efficient parallelization of loops, especially with respect to vector memory operations and the spawning of threads.

Superword level parallelism (SLP) [LA00] can be used to improve performance in machines that support SIMD operations. Shin et al. [SHC05] discuss how SLP can be applied even in the presence of control flow by using if-conversion. The disadvantage of this approach is that all possible control paths have to be executed. This is addressed with the concept of generating *branches-on-superword-condition-codes* (BOSCCs) [SHC04, Shi07]. BOSCCs allow a vector instruction or group of vector instructions to be bypassed if the guarding vector predicate contains all false values. Scale is more flexible in its handling of control flow because each VP thread—which corresponds to a single superword element in the BOSCC approach—can direct its own control flow without being tied to the other threads.

The Daecomp compiler for decoupled access/execute processors [RF00] uses a five-step process to partition instructions between the access and execute instruction streams. It attempts to avoid dependences that will reduce *slip* between the Access Processor and Execute Processor—i.e. the more the Access Processor runs ahead, the more slip there is. For example, it uses techniques like common sub-expression replication to avoid copies from the Execute Processor to the Access Processor. This is a similar goal to the Scale compiler's clustering approach, which tries to avoid cyclic dataflow.

A.4 Summary

This appendix described the overall flow of the Scale compiler, showing how it mapped loops to the architecture and how it attempted to take advantage of various Scale features such as decoupled execution. The compiler can parallelize several types of loops, and is able to achieve significant speedups over a single-issue scalar processor. Although there are many improvements that can be made to the compiler infrastructure, this appendix illustrated the feasibility of compiling for Scale and the performance benefits that can be achieved by automatically parallelizing codes for vector-thread architectures. This evaluation serves to demonstrate the potential of using explicitly parallel architectures more widely in mainstream computing.

Bibliography

[AB04] S. Alli and C. Bailey. A mechanism for implementing precise exceptions in pipelined processors. In *Proceedings of the 2004 EUROMICRO Symposium on Digital System Design*, pages 598–602, August–September 2004.

[ABC+95] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[ABC+06] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, pages 1–6, June 1990.

[ADM95] D. I. August, B. L. Deitrich, and S. A. Mahlke. Sentinel scheduling with recovery blocks. Technical Report CHRC-95-05, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, January 1995.

[AEL88] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.

[AHKW02] K. Asanović, M. Hampton, R. Krashinsky, and E. Witchel. Energy-exposed instruction sets. In *Power Aware Computing*, chapter 5. Kluwer Academic/Plenum Publishers, June 2002.

[AJLA95] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[AK01] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers, 2001.

[AKPW83] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[AL91] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, April 1991.

[ALBL91] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[ARS03a] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: an efficient, scalable alternative to reorder buffers. *IEEE Micro*, 23(6):11–19, November/December 2003.

[ARS03b] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: towards scalable large instruction window processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–434, December 2003.

[AS96] T. M. Austin and G. S. Sohi. High-bandwidth address translation for multiple-issue processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 158–167, May 1996.

[Asa98] K. Asanović. *Vector microprocessors*. PhD thesis, University of California at Berkeley, 1998.

[AST67] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 Model 91: machine philosophy and instruction handling. *IBM Journal of Research and Development*, 11(1):8–24, January 1967.

[BAH$^+$94] B. Burgess, M. Alexander, Y.-W. Ho, S. P. Litch, S. Mallick, D. Ogden, S.-H. Park, and J. Slaton. The PowerPC 603 microprocessor: a high performance, low power, superscalar RISC microprocessor. In *Proceedings of COMPCON 1994*, pages 300–306, February–March 1994.

[BBdG$^+$06] T. Bernard, K. Bousias, B. de Geus, M. Lankamp, L. Zhang, A. Pimentel, P. M. W. Knijnenburg, and C. R. Jesshope. A microthreaded architecture and its compiler. In *Proceedings of the 12th International Workshop on Compilers for Parallel Computers*, pages 326–340, January 2006.

[BEKK00] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–898, November 2000.

[BK92] G. Blanck and S. Krueger. The SuperSPARC microprocessor. In *Proceedings of COMPCON 1992*, pages 136–141, February 1992.

[BKGA04] C. Batten, R. Krashinsky, S. Gerding, and K. Asanović. Cache refill/access decoupling for vector machines. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 331–342, December 2004.

[BKW94] K. Bala, M. F. Kaashoek, and W. E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, pages 243–253, November 1994.

[BL04] G. B. Bell and M. H. Lipasti. Deconstructing commit. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 68–77, March 2004.

[Bro02] D. H. Brown Associates, Inc. Cray launches X1 for extreme supercomputing, November 2002.

[BSH91] D. P. Bhandarkar, R. Supnik, and S. Hobbs. Exception reporting mechanism for a vector processor. U.S. Patent 5,043,867, August 1991.

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

[BYA93] G. R. Beck, D. W. L. Yen, and T. L. Anderson. The Cydra 5 minisupercomputer: architecture and implementation. *The Journal of Supercomputing*, 7(1–2):143–180, May 1993.

[CBJ92] J. B. Chen, A. Borg, and N. P. Jouppi. A simulation based study of TLB performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 114–123, May 1992.

[CCK$^+$06] K. E. Coons, X. Chen, S. K. Kushwaha, D. Burger, and K. S. McKinley. A spatial path scheduling algorithm for EDGE architectures. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, October 2006.

[CCYT05] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, May/June 2005.

[CDSW05] C. Caşcaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 339–349, September 2005.

[CFM03] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 300–311, June 2003.

[CGH$^+$05] L. N. Chakrapani, J. Gyllenhaal, W. W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran: an infrastructure for research in instruction-level parallelism. *Lecture Notes in Computer Science*, 3602:32–41, August 2005.

[Cha81] A. E. Charlesworth. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family. *Computer*, 14(9):18–27, September 1981.

[Chi91] T. Chiueh. Multi-threaded vectorization. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 352–361, May 1991.

[CHJ+90] R. P. Colwell, W. E. Hall, C. S. Joshi, D. B. Papworth, P. K. Rodman, and J. E. Tornes. Architecture and implementation of a VLIW supercomputer. In *Proceedings of the 1990 Conference on Supercomputing*, pages 910–919, November 1990.

[Chr96] D. Christie. Developing the AMD-K5 architecture. *IEEE Micro*, 16(2):16–27, April 1996.

[CNO+87] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, October 1987.

[COLV04] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 48–59, February 2004.

[Cor98] Compaq Computer Corporation. Alpha architecture handbook, October 1998.

[CSB92] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[CSC+05] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero. Kilo-instruction processors: overcoming the memory wall. *IEEE Micro*, 25(3):48–57, May/June 2005.

[CSVM04] A. Cristal, O. J. Santana, M. Valero, and J. F. Martinez. Toward kilo-instruction processors. *ACM Transactions on Architecture and Code Optimization*, 1(4):389–417, December 2004.

[DDHS00] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, March/April 2000.

[DDM06] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 33–42, September 2006.

[Den70] P. J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[DHB89] J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, April 1989.

[Die98] K. Diefendorff. K7 challenges Intel. *Microprocessor Report*, 12(14):1, 6–11, October 1998.

[DM97] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing*, pages 68–75, July 1997.

[DOOB95] P. K. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton. Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading. Technical Report RC19928, IBM Research Division, T.J. Watson Research Center, February 1995.

[DT92] H. Dwyer and H. C. Torng. An out-of-order superscalar processor with speculative execution and fast, precise interrupts. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 272–281, December 1992.

[EAE$^+$02] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a vector extension to the Alpha architecture. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 281–292, May 2002.

[eem] EEMBC. http://www.eembc.org/.

[EOO$^+$05] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the CELL processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, September 2005.

[ERPR95] J. H. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan. Superscalar instruction execution in the 21164 Alpha microprocessor. *IEEE Micro*, 15(2):33–43, April 1995.

[ESV$^+$99] J. T. J. Van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E. J. D. Pol, M. J. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. E. Vranken. TriMedia CPU64 architecture. In *Proceedings of the 1999 IEEE International Conference on Computer Design*, pages 586–592, October 1999.

[EV96] R. Espasa and M. Valero. Decoupled vector architectures. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pages 281–290, February 1996.

[EVS97] R. Espasa, M. Valero, and J. E. Smith. Out-of-order vector architectures. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 160–170, December 1997.

[FBF$^+$00] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 203–213, June 2000.

[FFY05] J. A Fisher, P. Faraboschi, and C. Young. *Embedded computing: a VLIW approach to architecture, compilers, and tools.* Elsevier Inc., 2005.

[FG01] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 230–239, June–July 2001.

[Fis83] J. A. Fisher. Very Long Instruction Word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, June 1983.

[gcc] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[GH96] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[Gsc07] M. Gschwind. The Cell Broadband Engine: exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, June 2007.

[GT96] G. Goldman and P. Tirumalai. UltraSPARC-II: the advancement of ultracomputing. In *Proceedings of COMPCON 1996*, pages 417–423, February 1996.

[Gwe95] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, February 1995.

[HA06] M. Hampton and K. Asanović. Implementing virtual memory in a vector processor with software restart markers. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 135–144, June–July 2006.

[HA08] M. Hampton and K. Asanović. Compiling for vector-thread architectures. In *To appear, Proceedings of the 2008 International Symposium on Code Generation and Optimization*, April 2008.

[Hal96] T. Halfhill. AMD K6 takes on Intel P6. *Byte*, 21(1):67–72, January 1996.

[Ham01] M. Hampton. Exposing datapath elements to reduce microprocessor energy consumption. Master's thesis, Massachusetts Institute of Technology, June 2001.

[Hen94] D. S. Henry. Adding fast interrupts to superscalar processors. Computation Structures Group Memo 366, December 1994.

[HH93] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 39–50, May 1993.

[HHR95] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: an introduction and motivation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 158–168, November–December 1995.

[HK94] G. C. Hwang and C. M. Kyung. New hardware scheme supporting precise exception handling for out-of-order execution. *Electronics Letters*, 30(1):16–17, January 1994.

[HL99] T. Horel and G. Lauterbach. UltraSPARC-III: designing third-generation 64-bit performance. *IEEE Micro*, 19(3):73–85, May/June 1999.

[HM93] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[Hof05] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, February 2005.

[HP87] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 18–26, June 1987.

[HP07] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, Inc., fourth edition, 2007.

[HS99] S. Hily and A. Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 64–67, January 1999.

[Hsu94] P. Y.-T. Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[HSU$^+$01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 5(1), February 2001.

[Hun95] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *Proceedings of COMPCON 1995*, pages 123–128, March 1995.

[IBE$^+$07] M. M. Islam, A. Busck, M. Engbom, S. Lee, M. Dubois, and P. Stenström. Limits on thread-level speculative parallelism in embedded applications. In *Proceedings of the 11th Annual Workshop on the Interaction Between Compilers and Computer Architecture*, pages 40–49, February 2007.

[IBM07] IBM. Cell Broadband Engine programming handbook, version 1.1, April 2007.

[ISK07] J. In, I. Shin, and H. Kim. SWL: a search-while-load demand paging scheme with NAND flash memory. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 217–226, June 2007.

[Jes01] C. Jesshope. Implementing an efficient vector instruction set in a chip multiprocessor using micro-threaded pipelines. In *Proceedings of the 6th Australasian Conference on Computer Systems Architecture*, pages 80–88, January 2001.

[JGS99] C. Jaramillo, R. Gupta, and M. L. Soffa. Comparison checking: an approach to avoid debugging of optimized code. In *Proceedings of the 7th European*

Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 268–284, September 1999.

[JJ06] A. Jaleel and B. Jacob. In-line interrupt handling and lock-up free translation lookaside buffers (TLBs). *IEEE Transactions on Computers*, 55(5):559–574, May 2006.

[JM98a] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, July/August 1998.

[JM98b] B. Jacob and T. Mudge. Virtual memory: issues of implementation. *Computer*, 31(6):33–43, June 1998.

[JM98c] B. L. Jacob and T. N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 295–306, October 1998.

[Joh91] M. Johnson. *Superscalar microprocessor design.* P T R Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[KAF$^+$97] A. R. Kennedy, M. Alexander, E. Fiene, J. Lyon, B. Kuttanna, R. Patel, M. Pham, M. Putrino, C. Croxton, S. Litch, and B. Burgess. A G3 PowerPC superscalar low-power microprocessor. In *Proceedings of COMPCON 1997*, pages 315–324, February 1997.

[KAO05] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[KBH$^+$04a] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 52–63, June 2004.

[KBH$^+$04b] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The vector-thread architecture. *IEEE Micro*, 24(6):84–90, November/December 2004.

[KCL$^+$99] S. W. Keckler, A. Chang, W. S. Lee, S. Chatterjee, and W. J. Dally. Concurrent event handling through multithreading. *IEEE Transactions on Computers*, 48(9):903–916, September 1999.

[KDK$^+$01] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: media processing with streams. *IEEE Micro*, 21(2):35–46, March/April 2001.

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

[Kla00] A. Klaiber. The technology behind Crusoe processors. White paper, Transmeta Corporation, January 2000.

[KMAC03] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, March/April 2003.

[KOE$^+$01] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Reference idempotency analysis: a framework for optimizing speculative execution. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 2–11, June 2001.

[Koz02] C. Kozyrakis. *Scalable vector media-processors for embedded systems.* PhD thesis, University of California at Berkeley, May 2002.

[KP03] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 399–409, June 2003.

[KPEG04] G. Kucuk, D. V. Ponomarev, O. Ergin, and K. Ghose. Complexity-effective reorder buffer designs for superscalar processors. *IEEE Transactions on Computers*, 53(6):653–665, June 2004.

[Kra07] R. M. Krashinsky. *Vector-thread architecture and implementation.* PhD thesis, Massachusetts Institute of Technology, June 2007.

[KS02a] G. B. Kandiraju and A. Sivasubramaniam. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 129–139, June 2002.

[KS02b] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for TLB prefetching: an application-driven study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 195–206, May 2002.

[KS02c] H. Kim and J. E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 71–81, May 2002.

[KTHK03] K. Kitagawa, S. Tagaya, Y. Hagihara, and Y. Kanoh. A hardware overview of SX-6 and SX-7 supercomputer. *NEC Research & Development Journal*, 44(1), January 2003.

[KVN$^+$06] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, M. Girkarmark, X. Tian, and H. Saito. Challenges in exploitation of loop parallelism in embedded applications. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 173–180, October 2006.

[LA00] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 145–156, June 2000.

[LH91] B. D. Lightner and G. Hill. The Metaflow Lightning chipset. In *Proceedings of COMPCON 1991*, pages 13–18, February–March 1991.

[Lov77] D. B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[LRA05] S. Larsen, R. Rabbah, and S. Amarasinghe. Exploiting vector parallelism in software pipelined loops. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 119–129, November 2005.

[LS98] C. G. Lee and M. G. Stoodley. Simple vector microprocessors for multimedia applications. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 25–36, November–December 1998.

[LTT95] D. Levitan, T. Thomas, and P. Tu. The PowerPC 620 microprocessor: a high performance superscalar RISC microprocessor. In *Proceedings of COMPCON 1995*, pages 285–291, March 1995.

[MBH^{+}02] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), February 2002.

[MCH^{+}92] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247, October 1992.

[McL93] E. McLellan. The Alpha AXP architecture and 21064 processor. *IEEE Micro*, 13(3):36–47, June 1993.

[MH01] M. C. Merten and W. W. Hwu. Modulo schedule buffers. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 138–149, December 2001.

[MNM01] R. Manohar, M. Nyström, and A. J. Martin. Precise exceptions in asynchronous processors. In *Proceedings of the 2001 Conference on Advanced Research in VLSI*, pages 16–28, March 2001.

[MP89] S. W. Melvin and Y. N. Patt. Performance benefits of large execution atomic units in dynamically scheduled machines. In *Proceedings of the 3rd International Conference on Supercomputing*, pages 427–432, June 1989.

[MPV93] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 202–213, December 1993.

[MRHP02] J. F. Martinez, J. Renau, M. C. Huang, and M. Prvulovic. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 3–14, November 2002.

[MS03] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, March 2003.

[MSWP03] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 129–140, February 2003.

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

[MWV92] S. Mirapuri, M. Woodacre, and N. Vasseghi. The MIPS R4000 processor. *IEEE Micro*, 12(2):10–22, April 1992.

[ND95] P. R. Nuth and W. J. Dally. The named-state register file: implementation and performance. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, pages 4–13, January 1995.

[NHS93] C. J. Newburn, A. S. Huang, and J. P. Shen. Balancing fine- and medium-grained parallelism in scheduling loops for the XIMD architecture. In *Proceedings of the IFIP WG10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 39–52, January 1993.

[NKB+04] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. W. Keckler. Static placement, dynamic issue (SPDI) scheduling for EDGE architectures. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 74–84, September–October 2004.

[OG90] R. R. Oehler and R. D. Groves. IBM RISC System/6000 processor architecture. *IBM Journal of Research and Development*, 34(1):23–36, January 1990.

[OSM+98] E. Özer, S. W. Sathaye, K. N. Menezes, S. Banerjia, M. D. Jennings, and T. M. Conte. A fast interrupt handling scheme for VLIW processors. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 136–141, October 1998.

[OW00] F. P. O'Connell and S. W. White. POWER3: the next generation of PowerPC processors. *IBM Journal of Research and Development*, 44(6):873–884, November 2000.

[PA95] J. S. Park and G. S. Ahn. A software-controlled prefetching mechanism for software-managed TLBs. *Microprocessing and Microprogramming*, 41(2):121–136, May 1995.

[PGH+87] A. R. Pleszkun, J. R. Goodman, W. C. Hsu, R. T. Joersz, G. Bier, P. Woest, and P. B. Schechter. WISQ: a restartable architecture using queues. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 290–299, June 1987.

[PKL+95] N. Patkar, A. Katsuno, S. Li, T. Maruyama, S. Savkar, M. Simone, G. Shen, R. Swami, and D. Tovey. Microarchitecture of HaL's CPU. In *Proceedings of COMPCON 1995*, pages 259–266, March 1995.

[PLK$^+$04] C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min. Compiler-assisted demand paging for embedded systems with flash memory. In *Proceedings of the 4th ACM International Conference on Embedded Software*, pages 114–124, September 2004.

[PLW$^+$06] J. Peng, G.-Y. Lueh, G. Wu, X. Gou, and R. Rakvic. A comprehensive study of hardware/software approaches to improve TLB performance for Java applications on embedded systems. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 102–111, October 2006.

[PM93] J. K. Pickett and D. G. Meyer. Enhanced superscalar hardware: the schedule table. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 636–644, November 1993.

[PSS$^+$91] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman. The Metaflow architecture. *IEEE Micro*, 11(3):10–13, 63–73, June 1991.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

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

[QCEV99] F. Quintana, J. Corbal, R. Espasa, and M. Valero. Adding a vector unit to a superscalar processor. In *Proceedings of the 13th International Conference on Supercomputing*, pages 1–10, June 1999.

[RB95] W. F. Richardson and E. Brunvand. Precise exception handling for a self-timed processor. In *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors*, pages 32–37, October 1995.

[RBH$^+$95] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 285–298, December 1995.

[REL03] J. Redstone, S. Eggers, and H. Levy. Mini-threads: increasing TLP on small-scale SMT processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 19–30, February 2003.

[RF00] K. D. Rich and M. K. Farrens. Code partitioning in decoupled compilers. *Lecture Notes in Computer Science*, 1900:1008–1017, 2000.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, December 1981.

[RMKU04] A. Rodrigues, R. Murphy, P. Kogge, and K. Underwood. Characterizing a new class of threads in scientific applications for high end supercomputers. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 164–174, June–July 2004.

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

[Rud97] K. W. Rudd. Efficient exception handling techniques for high-performance processor architectures. Technical Report CSL-TR-97-732, Departments of Electrical Engineering and Computer Science, Stanford University, August 1997.

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

[RYYT89] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towie. The Cydra 5 departmental supercomputer: design philosophies, decisions, and trade-offs. *Computer*, 22(1):12–26, 28–30, 32–35, January 1989.

[SA00] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, September 2000.

[SBG$^+$06] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *Proceedings of the 4th International Symposium on Code Generation and Optimization*, pages 185–195, March 2006.

[SBV95] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[sca] Scale Home Page. http://www-ali.cs.umass.edu/scale/.

[SDC94] S. P. Song, M. Denman, and J. Chang. The PowerPC 604 RISC microprocessor. *IEEE Micro*, 14(5):8–17, October 1994.

[SDS00] A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-based TLB preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 117–127, June 2000.

[Ses98] N. Seshan. High VelociTI processing. *IEEE Signal Processing Magazine*, 15(2):86–101, 117, March 1998.

[SG01] M. H. Samadzadeh and L. E. Garalnabi. Hardware/software cost analysis of interrupt processing strategies. *IEEE Micro*, 21(3):69–76, May/June 2001.

[SHC04] J. Shin, M. Hall, and J. Chame. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. In *Proceedings of the 6th Workshop on Media and Streaming Processors*, December 2004.

[SHC05] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of the 2005 International Symposium on Code Generation and Optimization*, pages 165–175, March 2005.

[Shi07] J. Shin. Introducing control flow into vectorized code. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, September 2007.

[Sit93] R. L. Sites. Alpha AXP architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

[SKT+05] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, July/September 2005.

[SL99] E. Stotzer and E. Leiss. Modulo scheduling for the TMS320C6x VLIW DSP architecture. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 28–34, May 1999.

[Smi81] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. *Society of Photooptical Instrumentation Engineers Real-Time Signal Processing IV*, 298:241–248, 1981.

[Smi98] J. E. Smith. Retrospective: implementing precise interrupts in pipelined processors. In *25 Years of the International Symposia on Computer Architecture (selected papers)*, page 42, June–July 1998.

[SNL+03] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[Soh90] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[SP85] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 36–44, June 1985.

[SP88] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[spe] Spec homepage. http:://www.spec.org.

[SR01] A. Saulsbury and D. S. Rice. Microprocessor with reduced context switching overhead and corresponding method. U.S. Patent 6,314,510, November 2001.

[SRA+04] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, October 2004.

[SS01] E. Stotzer and R. H. Scales. Interruptable multiple execution unit processing during operations using multiple assignment of registers. U.S. Patent 6,178,499, January 2001.

[SSS+02] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon. Low-power data forwarding for VLIW embedded architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(5):614–622, October 2002.

[SWB95] J. S. Synder, D. B. Whalley, and T. P. Baker. Fast context switches: compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, 19(1):35–42, February 1995.

[SWL05] P. G. Sassone, D. S. Wills, and G. H. Loh. Static strands: safely collapsing dependence chains for increasing embedded power efficiency. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 127–136, July 2005.

[Tar72] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[TBGG03] X. Tian, A. Bik, M. Girkar, and P. Grey. Exploiting thread-level and instruction-level parallelism for Hyper-Threading Technology. *Intel Developer Update Journal*, January 2003.

[TD93] H. C. Torng and M. Day. Interrupt handling for out-of-order execution processors. *IEEE Transactions on Computers*, 42(1):122–127, January 1993.

[TDJSF$^+$02] J. M. Tendler, J. S. Dodson, Jr. J. S. Fields, , H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[TEL95] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[THA$^+$99] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, September 1999.

[TL94] C. A. Thekkath and H. M. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the 6th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, October 1994.

[TO96] M. Tremblay and J. M. O'Connor. UltraSparc I: a four-issue processor supporting multimedia. *IEEE Micro*, 16(2):42–50, April 1996.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[TS88] M. R. Thistle and B. J. Smith. A processor architecture for Horizon. In *Proceeding of the 1988 ACM/IEEE Conference on Supercomputing*, pages 35–41, November 1988.

[UH93] N. Ullah and M. Holle. The MC88110 implementation of precise exceptions in a superscalar architecture. *ACM SIGARCH Computer Architecture News*, 21(1):15–25, March 1993.

[UIT94] T. Utsumi, M. Ikeda, and M. Takamura. Architecture of the VPP500 parallel supercomputer. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pages 478–487, November 1994.

[UNS+94] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest, and R. Brown. Design tradeoffs for software-managed TLBs. *ACM Transactions on Computer Systems*, 12(3):175–205, August 1994.

[URv03] T. Ungerer, B. Robič, and J. Šilc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, March 2003.

[WC95] W. Walker and H. G. Cragon. Interrupt processing in concurrent processors. *IEEE Computer*, 28(6):36–46, June 1995.

[WE93] C.-J. Wang and F. Emnett. Implementing precise interruptions in pipelined RISC processors. *IEEE Micro*, 13(4):36–43, July/August 1993.

[WFW+94] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[WK92] P. R. Wilson and S. V. Kakkad. Pointer swizzling at page fault time: efficiently and compatibly supporting huge address spaces on standard hardware. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 364–377, September 1992.

[Wol04] A. Wolfe. Intel clears up post-Tejas confusion. *VARBusiness*, May 17 2004.

[WS85] C. Whitby-Strevens. The transputer. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 292–300, June 1985.

[WS91] A. Wolfe and J. P. Shen. A variable instruction stream extension to the VLIW architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, April 1991.

[WW93] C. A. Waldspurger and W. E. Weihl. Register relocation: flexible contexts for multithreading. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 120–130, May 1993.

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

[ZES99] C. B. Zilles, J. S. Emer, and G. S. Sohi. The use of multithreading for exception handling. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 219–229, November 1999.

[ZP06] X. Zhou and P. Petrov. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *Proceedings of the 43rd Annual Conference on Design Automation*, pages 352–357, July 2006.