SUPPORT FOR SPECULATIVE EXECUTION IN HIGH-PERFORMANCE PROCESSORS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Michael David Smith

November 1992

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Mark A. Horowitz (Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Monica S. Lam

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Teresa H. Meng

Approved for the University Committee on Graduate Studies:

Abstract

Superscalar and superpipelining techniques increase the overlap between the instructions in a pipelined processor, and thus these techniques have the potential to improve processor performance by decreasing the average number of cycles between the execution of adjacent instructions. Yet, to obtain this potential performance benefit, an instruction scheduler for this high-performance processor must find the independent instructions within the instruction stream of an application to execute in parallel. For non-numerical applications, there is an insufficient number of independent instructions within a basic block, and consequently the instruction scheduler must search across the basic block boundaries for the extra instruction-level parallelism required by the superscalar and superpipelining techniques. To exploit instruction-level parallelism across a conditional branch, the instruction scheduler must support the movement of instructions above a conditional branch, and the processor must support the speculative execution of these instructions.

We define boosting, an architectural mechanism for speculative execution, that allows us to uncover the instruction-level parallelism across conditional branches without adversely affecting the instruction count of the application or the cycle time of the processor. Under boosting, the compiler is responsible for analyzing and scheduling instructions, while the hardware is responsible for ensuring that the effects of a speculatively-executed instruction do not corrupt the program state when the compiler is incorrect in its speculation. To experiment with boosting, we built a global instruction scheduler, which is specifically tailored for the non-numerical environment, and a simulator, which determines the cycle-count performance of our globally-scheduled programs. We also analyzed the hardware requirements for boosting in a typical load/store architecture. Through the cycle-count simulations and an understanding of the cycle-time impact of the hardware support for boosting, we found that only a small amount of hardware support for speculative execution is necessary to achieve good performance in a small-issue, superscalar processor.

This dissertation is dedicated to

the loving memory of my brother

Andrew Fairman Smith.

Acknowledgments

Six and a half years at Stanford. Nine thousand hours in front of a workstation. Eleven million keystrokes and mouse clicks. Is this toughening of the fingertips the essence of a graduate career? Fortunately not. I can honestly say that I enjoyed my graduate career because of the people I met between those keystrokes and mouse clicks.

Certainly, the one person who has the biggest influence on any graduate career is the principle thesis advisor. I consider myself lucky to have had Mark Horowitz as my advisor for he is a truly unique individual. As a principle thesis advisor, I guess that Mark is obligated to listen to the crazy ideas of his students, but he always listened to the craziest of my ideas with genuine interest and unfaltering patience. Of course, he never listened for long because he has this uncanny ability to understand your entire idea, the ramifications of your idea, and the problems with your idea from the first two sentences out of your mouth. I thank him for all that he has taught me and for the time that he has spent with me.

Actually, I am one of those fortunate individuals with more than one interested advisor. Monica Lam graciously acted as my alternate advisor, answering whatever compiler questions I had. I am not sure that Monica realized just how little I knew of compiler technology when she first agreed to support my research, but in a short period of time, she helped me learn more about compilers than I would have ever imagined possible.

I would also like to particularly acknowledge the support and guidance of three other Stanford professors. The first of these professors is John Hennessy. John helped get me started at Stanford, he sat on my orals committee, and he basically kept me sharp throughout my graduate career. John continually referred his external visitors to my cubicle, and he often stopped by to suggest that I volunteer for yet another talk. Though I first viewed these activities as an unwelcome distraction, I later realized that they were opportunities which had an immeasurable effect on my research and on my development. I also wish to thank Professor Anoop Gupta for treating me as a colleague from my very first hour at Stanford. I hope that our discussions were as helpful to him as they were to me. Finally, I want to thank Professor Teresa Meng who chaired my orals committee and acted as a reader for this dissertation.

Besides my professors, I wish to acknowledge the support of the staff of the Center for Integrated Systems and the help and friendship of my fellow students in the DASH, SUIF, and TORCH research groups. Without their aid, none of the research in this thesis would have been possible. I should especially thank the original members of the TORCH group (Tom Chanak, Phil Lacroute, John Maneatis, Don Ramsey, and Drew Wingard) for believing in my work long enough to make it a reality. Also, I need to particularly thank Wolf Weber and Kourosh Gharachorloo for so honestly reviewing my papers and talks.

Like many projects at Stanford, my research was also supported by many generous individuals outside the university. Of all of these individuals, four desire special recognition. I want to thank Peter Davies, Mike Johnson, and Earl Killian who each in some way contributed to the simulation environment used in this research. I also want to thank Neil Wilhelm for his understanding and guidance during those difficult years when I was searching for a research topic.

My final thanks must go to my family. My family has grown enormously since my first days at Stanford, and I cherish the understanding and compassion that they all showed me throughout the years. Of course, my deepest thanks must go to my wife Chris, who more than anyone else has supported me both financially and emotionally. Chris never once questioned me as to when I would be done, and she did a wonderful job of filling in those few hours that I was away from my workstation.

This work was supported by the Defense Advanced Projects Research Agency (DARPA) under contract N00039-91-C-0138. The author’s support by Digital Equipment Corporation through the CIS Fellow-Mentor-Advisor program is also gratefully acknowledged.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reducing the run time of a program is the foremost goal of processor design, and in this thesis, we are particularly interested in reducing the run time of non-numerical applications. The run time of a program consists of three factors: the cycle time of the processor, the average number of cycles per instruction (CPI) for a program, and the total number of instructions executed by that program. The product of these three factors not only equals the run time of the program, but it also represents the performance of the processor. This thesis investigates an architectural technique for high-performance processor design which decreases the average CPI of a program while minimally affecting the instruction count of that program and the cycle time of the processor.

During the RISC revolution of the past decade, one way computer architects improved processor performance was through the use of pipelining and instruction scheduling. Pipelining decomposes the steps needed to execute an instruction into independent stages and then overlaps the processing of these stages. Pipelining increases performance by reducing the average time between the execution of adjacent instructions (i.e. by reducing the average CPI). Pipelining works best when the processing of an instruction in one stage is independent of the instruction processing in the other stages. To optimize the advantage of pipelining in RISC processors, an instruction scheduler attempts to arrange the instruction stream to guarantee that the execution of an instruction is independent of the previous instructions still in execution. This independence in the instruction stream is called *instruction-level parallelism* (ILP), and by exploiting ILP, existing instruction schedulers for RISC machines nearly achieve an average CPI of one.

To further decrease the average CPI, computer architects are designing pipelines with an even greater overlap in the execution of adjacent instructions. A *superscalar* or *superpipelined* processor is an example of processor with this type of enhanced pipeline [Jouppi and Wall 1989]. To take advantage of superscalar and superpipelining techniques, an instruction scheduler must extract and exploit a larger amount of the ILP within an application. Recent studies on the amount of exploitable ILP within applications have shown that *speculative execution* is required to appreciably increase the amount of ILP in non-numerical applications [Wall 1991, Lam and Wilson 1992]. As a result, this thesis investigates a mechanism for speculative execution that increases the amount of exploitable ILP within non-numerical applications. Yet, increasing the amount of exploitable ILP is not enough to guarantee faster run times, this thesis also shows that it is possible for an implementation of the speculative mechanism and for an implementation of the instruction scheduler that uses this mechanism to minimally impact the cycle time of the processor and the instruction count of the application. The rest of this chapter further motivates the need for speculative execution, and it introduces our approach to handling speculative execution and instruction scheduling within the domain of non-numerical applications.

1.1 Constraints on ILP

The dependence constraints within an application define the amount of *exploitable* ILP within that application. By removing or reducing the severity of some of these dependence constraints, one can increase the amount of exploitable ILP within the application. This section overviews the dependence constraints on ILP, and it describes a number of important techniques which reduce or remove some of these dependence constraints. In other words, this section describes the techniques required for an instruction scheduler to extract a large amount of the ILP within an application.

Data dependence and control dependence are the two types of program constraints that enforce an ordering on the instructions in an application and therefore limit the amount of exploitable ILP within the application. Data dependence is manifested in three forms: true (Read-After-Write) dependences, anti- (Write-After-Read) dependences, and output (Write-After-Write) dependences. Figure 1-1 lists a small code segment which contains an example of each of these types of data dependence. There exists a true data dependence between instruction `i1` and instruction `i2` of Figure 1-1 since `i1` is producing a result which `i2` needs. True dependences represent the flow of data through the program, and thus, they are difficult to remove without rewriting the algorithm.

Anti- and output dependences, on the other hand, are simply due to the reuse of storage resources in the machine, and because of this fact, we refer to these two types of data dependence as *storage conflicts*. Figure 1-1 contains storage conflicts due to the reuse of

```
i1: r1 = r2 + r3
i2: r3 = r1 + 1
i3: r1 = r4 + 2
```

Figure 1-1: Examples of data dependence.

register resources. For example, there exists an anti-dependence between instruction `i1` and instruction `i2` since `i2` needs to write its result to register `r3`, and it cannot perform this write until `i1` reads the previous value from `r3`. There exists an output dependence between `i1` and `i3` since both instructions write their results into register `r1`, and `i1` must write its result first.

Like the earlier studies on ILP, recent studies on RISC object files show that storage conflicts due to registers can severely limit the amount of exploitable ILP within an application [MDSmith et al. 1989, Wall 1991]. There are techniques, such as *register renaming*, that an instruction scheduler can use to remove these anti- and output dependences. For instance, if the destination register of `i3` is changed from `r1` to `r5`, the first and third instructions are now independent and can execute in parallel. Thus, register renaming increases the amount of exploitable ILP by removing register conflicts. Yet, register renaming does not remove all of the storage conflicts within an application because data dependences also occur between memory locations.

Obviously, a data dependence can flow through a memory location in the same manner that it flowed through a register. For instance, a true data dependence exists between a store instruction and a load instruction if the store writes a value to a memory location that the load later reads. Similarly, an anti-dependence can exist between a load and a store, and an output dependence can exist between two stores. The determination of a data dependence between two memory operands (called *alias analysis*) is more difficult than the determination of a data dependence between two register operands because, unlike a register operand, a single memory operand can address many different memory locations during the execution of the application. In other words, two memory operations may be data dependent during one instance of their execution and not data dependent during the next instance. Wall [1991] shows that alias analysis can increase the amount of exploitable ILP within an application.

Control dependence, like data dependence, also limits the amount of exploitable ILP within an application. Control dependence is due to the existence of conditional branches in the program, and these branches impose a dependence upon the instructions that follow them. Figure 1-2 contains a simple code segment in which instructions `i2` and `i4` are control dependent upon the conditional branch instruction `i1`. That is, a processor must first evaluate the branch condition before it can determine whether register `r1` should be incremented or decremented.

```
i1: if (cond) then
i2:   r3 = r1 + 1
i3: else
i4:   r3 = r1 - 1
```

Figure 1-2: Example of control dependence.

Again like the earlier studies on ILP, the recent studies on RISC object files show that the amount of exploitable ILP within a basic block is quite small [MDSmith et al. 1989, Wall 1991, Lam and Wilson 1992]. Yet, these same studies show that the removal of the control dependence constraints (i.e. the a priori knowledge of the direction of every branch in an application) greatly increases the amount of exploitable ILP. Unfortunately, it is impossible to have a priori knowledge about every branch in an application because some conditional branches depend upon information available only at run time. Even so, one can minimize the effects of control dependence by speculating on the direction of a branch.

Speculative execution is the execution of an instruction before it is known whether the instruction execution is necessary or correct. *Branch speculation* involves the prediction of a conditional branch direction and then the speculative execution of the instructions dependent upon that branch. Branch speculation increases the amount of exploitable ILP by removing control dependence constraints. For example, if the condition in Figure 1-2 usually evaluates so that instruction `i2` is executed, branch speculation provides a mechanism for scheduling and executing instruction `i2` before `i1`. The resulting speculative execution increases the exploitable ILP of the example code segment if the instruction scheduler can schedule the speculative execution of instruction `i2` in parallel with the instructions that occur in the block before `i1`.

Finally, numerous studies show that the severity of the data and control dependence constraints varies with the type of application [Nicolau and Fisher 1984, Wall 1991, Lam and Wilson 1992]. Thus, the relative importance of the techniques which increase the amount of exploitable ILP within an application depends upon the domain of the application. Register renaming is important in all applications, but even with register renaming, numerical applications, especially vectorizable applications, contain significantly more exploitable ILP than non-numerical applications, applications such as compilers and text editors. In fact, non-numerical object codes exhibit very little exploitable ILP because of their high frequency of conditional branches which depend upon run-time information. Thus, an instruction scheduler for non-numerical applications must take advantage of speculative execution in order to increase the amount of exploitable ILP within these applications.

1.2 Background

This thesis follows an approach to high-speed processor design where we distribute the functionality for instruction scheduling with speculative execution between the compiler and the hardware. To understand why this type of an approach is necessary, Section 1.2.1 overviews the strengths and shortcomings of a pure hardware and a pure software instruction scheduler. Section 1.2.2 then describes how each pure approach supports the potential of speculative execution.

1.2.1 Current approaches to instruction scheduling

In general, an instruction scheduler rearranges the instruction stream to maximize processor performance. More specifically, the instruction scheduler first analyzes a set of instructions to determine the dependences between the instructions and the processor resources required by the instructions. The analysis of resource needs is important because there are only a finite number of resources in any given processor. The instruction scheduler then defines an ordering on the instructions that satisfies the instruction dependences and the resource constraints. ILP allows the instruction scheduler to avoid pipeline stalls which result from the instruction dependences and resource constraints. These pipeline stalls decrease processor performance by increasing the average CPI.

Either the compiler or the hardware (or both) can perform instruction scheduling. *Static instruction scheduling* is instruction scheduling where the compiler performs dependence analysis and instruction ordering. Static instruction schedulers that look beyond basic block boundaries are called *global instruction schedulers*. Chapter 3 overviews the components of a global instruction scheduler, and it reviews a large number of existing algorithms. *Dynamic instruction scheduling* is instruction scheduling where the hardware performs dependence analysis and instruction ordering. A number of dynamic instruction scheduling techniques have been proposed [Thornton 1964, Tomasulo 1967, JESmith 1982]. Johnson [1990] provides a good survey of these techniques. If the hardware in a dynamic instruction scheduler can reorder the instructions in the instruction stream, the hardware supports *out-of-order execution*; if the hardware cannot reorder the instructions in the instruction stream, the hardware supports *in-order execution* [Johnson 1990]. Even though the compiler and the hardware can both perform instruction scheduling, there are strengths and shortcomings to each approach.

The biggest advantage that static instruction scheduling has over dynamic instruction scheduling is that static instruction scheduling is done without any overhead on the hardware; the hardware remains as simple and as fast as possible. In other words, improvements in a static instruction scheduler only affect the compile time of the application and not the cycle time of the processor. Improvements in a dynamic instruction scheduler, on the other hand, directly affect the complexity of the processor design, and this extra hardware complexity can impact both the design time and the cycle time of the processor. For instance, dynamic instruction schedulers with support for out-of-order execution generate instruction schedules with much better cycle counts than dynamic instruction schedulers with support for in-order execution. The cost of this improvement is increased hardware complexity because out-of-order execution requires significantly more hardware resources than in-order execution. In fact, Table 1-1 illustrates that good performance depends upon more than just a sophisticated hardware instruction scheduler.

		Clock rate	SPEC int92	SPEC fp92	SPEC89
RISC	HP Snake*	66 MHz	48.1	75.0	—
In-Order	Sun SuperSparc*	40 MHz	52.6	64.7	—
	MIPS R4000*	100 MHz	61.7	63.4	—
	DEC Alpha**	150 MHz	—	—	~110
Out-of-Order	IBM RS/6000*	50 MHz	42.0	85.6	—
	Motorola 88110***	50 MHz	~51.0	~73.9	~63.7

* Data taken from *Microprocessor Report*, Vol. 6, No. 7, May 27, 1992.
** Estimated data taken from *IEEE Spectrum*, Vol. 29, No. 7, July 1992.
*** Estimated data taken from *Microprocessor Report*, Vol. 5, No. 22, Dec. 4, 1991.

Table 1-1: Comparison of recent pipelined processors.

Since the amount of sophistication in the instruction scheduler directly impacts the complexity of the hardware in a dynamic instruction scheduler, people often implement dynamic schedulers with relatively simple scheduling algorithms. As a result of this simplicity, one advantage that a static instruction scheduler exhibits over a dynamic instruction scheduler is that the compiler often analyzes a much larger portion of the instruction stream. The size of the "window" into the instruction stream is important because it impacts the instruction scheduler's ability to find ILP. Bigger windows improve the potential for finding ILP. A dynamic instruction scheduler cannot match the analysis range of a static instruction scheduler because dependence analysis on a window of instructions grows super-linearly in the number of instructions. This super-linear factor impacts the complexity of the hardware faster than it impacts the complexity of the compiler.

Another advantage of a static instruction scheduler over a dynamic instruction scheduler is that the compiler can use more sophisticated heuristics for instruction ordering than the hardware can. Instruction schedulers use heuristics to choose the ordering of the instructions because the construction of an optimal schedule (one with the minimal cycle count) on a machine with a limited number of resources is an NP-complete problem [Gross 1983]. Compilers often use complex heuristics (such as the critical path through the application) to improve the instruction schedule. Because of hardware complexity, hardware schedulers often use simplistic heuristics to choose among the instructions that are ready for execution. For example, a recent implementation of the HPS processor, a dynamically-scheduled superscalar processor with out-of-order execution, simply chooses the oldest instruction in the set of ready instructions [Uvieghara et al. 1992]. The implementation shows that even this simple heuristic requires a significant amount of hardware resources.

Even with all these advantages, a static instruction scheduler is not an ideal solution to the problem of instruction scheduling. The biggest advantage of hardware scheduling over compiler scheduling is that the hardware can get exact information about the dependences between instructions and the resources required by the execution of an instruction. In other words, a dynamic instruction scheduler has access to run-time information that a static instruction scheduler cannot know, and this exactness in analysis information can improve the quality of the instruction schedule produced. Knowledge that a load operation causes a cache miss or that two memory operations do not access the same memory location are two examples of exact information which are difficult for a compiler to determine.Without exact information about the resources needed by an instruction or about the interaction between two instructions, a compiler makes a conservative decision and schedules for the worse-case situation.

1.2.2 Instruction scheduling with speculative execution

With speculative execution, one can build instruction scheduling techniques (such as branch speculation) which remove some of the program constraints and thus increase the

ILP in the program. The execution of a speculative instruction (i.e. speculative execution) is identical to the execution of a non-speculative instruction except that the effects of a speculatively-executed instruction should not corrupt the program state if the speculation is incorrect. For example in branch speculation, a speculatively-executed instruction should not corrupt the program state if its dependent branch condition is incorrectly predicted (Chapter 2 describes all of the cases of branch speculation in detail). Though the maintaining the correct program semantics independent of the outcome of the speculation can be the responsibility of the compiler or the hardware, the hardware is much better suited for dealing with the effects of a speculative operation that should not have occurred.

Specifically, the speculative execution of any instruction requires a buffer-and-undo mechanism to guarantee that the effects of a speculatively-executed instruction do not corrupt the program semantics when the speculation is incorrect. To achieve this buffer-and-undo mechanism, researchers have proposed a number of schemes for supporting speculative execution in hardware [JESmith and Pleszkun 1985, Hwu and Patt 1987, Sohi and Vajapeyam 1987, Johnson 1990]. Basically, these schemes involve the addition of extra buffering in the processor to hold the effects of speculatively-executed instructions until the hardware can determine whether the effects are useful or not. An incorrect speculation flushes the buffered effects while a correct speculation causes a transfer of the effects in the buffers into the user-visible program state.

The researchers usually couple this hardware-assisted speculative execution with a dynamic instruction scheduler because hardware-assisted speculative execution requires more information than is in a typical RISC instruction stream. That is, hardware-assisted speculative execution requires the instruction scheduler to indicate which instructions are speculative instructions (i.e. instructions requiring speculative execution), and it requires the instruction scheduler to indicate the speculative condition (e.g. the speculative instructions depend upon this conditional branch going this direction). By implementing both the instruction scheduler and the mechanism for speculative execution in the hardware, a designer can ensure that the instruction scheduler generates the information that the speculative execution hardware needs, and that the instruction scheduler can schedule any instruction for speculative execution.

Without hardware assistance to buffer-and-undo the effects of a speculatively-executed instruction, a static instruction scheduler is severely limited in its ability to schedule an instruction for speculative execution. Since a speculative instruction looks like any other instruction to the hardware, the compiler is entirely responsible for checking the speculative condition and undoing the effects of the speculative operations on an incorrect speculation. Yet, it is impossible for the compiler to prevent or undo exception processing (a side effect of some instructions). Even for the cases where the compiler can prevent or undo the effects of a speculative operation, it is often too expensive (in terms of run-time performance) for the compiler to insert extra code to check the speculative condition and undo the speculative effects. As a result of these limitations, a compiler usually settles for the conservative approach and only schedules an instruction for speculative execution if the effects of this instruction are harmless after an incorrect speculation.

Unfortunately, this conservative approach by the compiler greatly reduces the advantage of speculative execution. For instance, the one of the important advantages to branch speculation is the early execution of load operations since load operations often have longer latencies than the other instructions in the processor. However, the compiler cannot schedule a load before its dependent branch because the load is capable of causing an addressing exception which could halt the program.

In summary, the speculative execution of some instructions requires hardware assistance, and thus a pure software approach to instruction scheduling with speculative execution is unacceptable. A pure hardware approach is also unacceptable because the hardware complexity of a dynamic scheduler limits the sophistication of that instruction scheduler and thus the incremental benefit of speculative execution. For example, though the hardware can execute a load operation before its dependent branch, the hardware can only move the load up past the small number of instructions in its instruction window. Thus, neither a compiler-centric approach nor a hardware-centric approach are adequate solutions to the problem of instruction scheduling with speculative execution.

1.3 An integrated approach

To effectively support instruction scheduling with speculative execution, we require an integrated approach which considers the capabilities of both the compiler and the hardware. An integrated approach defines an architectural mechanism which bridges the gap between the compiler and the hardware so that the hardware can differentiate between a speculative and a non-speculative instruction. By considering the hardware and the compiler together, we can design an instruction scheduler with speculative execution that achieves the advantages of each of the pure approaches.

One way to build the instruction scheduler in this integrated approach is to actually build two full instruction schedulers—one in the compiler and one in the hardware. Melvin and Patt [1991] propose an architecture that follows this type of an integrated approach, and their study shows that the cycle-count performance (i.e. performance measured only as the product of CPI and instruction count) of this integrated approach does exceed the cycle-count performance of either pure approach. Though this type of integrated approach does produce solutions which extract and exploit the largest amount of ILP possible, this type of approach does so with a large amount of hardware complexity. In other words, this type of integrated approach attains the advantages of each of the pure approaches, but it also retains some of the shortcomings of dynamic instruction scheduling.

To reduce the hardware complexity, we need to focus the functionality of the instruction scheduler in the compiler and try to limit as much as possible the amount of hardware functionality. We should only include hardware complexity where absolutely necessary, i.e. where it provides functionality that the software cannot and where this functionality improves performance. Similarly, we want to develop an architectural mechanism which supports compile-time speculation with a small amount of hardware complexity. Guarding [Hsu and Davidson 1986] and non-excepting instructions [Colwell et al. 1987, Ebcioğlu 1988, Chang et al. 1991b] are examples of this type of an architectural mechanism. Overall, this integrated approach provides the scheduling range, heuristic sophistication, and hardware simplicity of a static instruction scheduler with the exact information and speculative freedom of a dynamic instruction scheduler. The instruction scheduler in this type of approach extracts and exploits more of the available ILP without necessarily impacting the instruction count of the application or the cycle time of the processor.

We also subscribe to this type of an integrated approach. Yet, instead of simply proposing another architectural mechanism to support instruction scheduling with speculative execution, this thesis describes a framework which facilitates the development of instruction scheduling techniques which exploit speculative execution. The framework for this integrated approach to instruction scheduling with speculative execution is called *opportunistic instruction scheduling*. This framework provides a straightforward method for creating a mechanism to speculate on any program constraint that is determined at run-time. This framework also allows us to think about speculative execution as an abstract mechanism available to the compiler. Thinking about speculative execution in this manner has an important consequence: it separates functionality from implementation considerations. This separation facilitates the evaluation of a range of cost/performance tradeoffs for a particular technique requiring speculative execution. That is, we rely on simulation and hardware analysis to determine the extent of the hardware support necessary in an architectural mechanism to achieve good processor performance at a low hardware cost. As an example of this approach, this thesis uses the opportunistic instruction scheduling framework to answer the question of how much branch speculation is necessary to achieve good performance in a simple superscalar processor.

The thesis is organized as follows. Chapter 2 contains a discussion of opportunistic instruction scheduling and a detailed description of *boosting*—our implementation of hardware-assisted speculative execution. The later chapters then focus on the development and evaluation of a specific implementation of boosting which supports branch speculation in a superscalar processor. Branch speculation is important because, as Section 1.1 mentioned, control dependence severely limits the amount of exploitable ILP in non-numerical applications. Specifically, we add boosting to a superscalar implementation of the MIPS R2000 architecture [Kane 1987], which we call TORCH.[1] Since opportunistic instruction scheduling relies heavily on the compiler for the scheduling of instructions and the exploiting of ILP, Chapter 3 describes our global scheduling algorithm for TORCH, and it discusses how boosting affects this algorithm. Chapter 4 then describes and evaluates a range of hardware support for boosting, and it shows that very little hardware support is necessary in the TORCH CPU to achieve good performance on non-numerical applications. Chapter 5 presents the conclusions of this research.

1. This thesis subsumes our earlier research on boosting and TORCH. MDSmith et al. [1990] and MDSmith et al. [1992] present the results of these early investigations.

Chapter 2

Opportunistic Instruction Scheduling

The biggest shortcoming of static instruction scheduling is that the compiler must remain conservative in its scheduling decisions. Whenever run-time information is needed to determine the dependences between instructions or the side effects of an instruction, the compiler must assume that the instructions are dependent or that the worst-case side effects will occur. In other words, the compiler assumes the existence of a program constraint where one may not actually exist at run time. This pessimism decreases the exploitable ILP, and thus it leads to lost opportunities for improved performance.

Opportunistic instruction scheduling is a framework which employs speculation to overcome this pessimism in the compiler. Speculation increases the exploitable ILP and it allows the compiler to take advantage of the potential opportunities for improving processor performance. Under opportunistic instruction scheduling, the compiler makes assumptions about the run-time information, and it then schedules the code under these assumptions. In other words, the compiler speculates on the value of the run-time information, and it relies on the hardware to guarantee that the program semantics are correctly maintained.

Of course, the framework is only useful if there exists an architectural mechanism which efficiently communicates the speculation information between the compiler and the hardware. Boosting is our implementation of speculative execution, and Section 2.1 motivates the usefulness of boosting by describing how boosting supports branch speculation. With this description as background, Section 2.2 discusses the general method for building compile-time techniques which rely on speculative execution and specifically boosting. Finally in Section 2.3, we show the versatility of this framework by constructing a speculative technique which allows the compiler to speculate on the data dependence constraint between two memory operations.

2.1 Branch speculation

Branch speculation increases the amount of exploitable ILP in an application by removing the control dependence constraints imposed by conditional branches. Without branch speculation, the instructions in the THEN and ELSE portions of an IF construct can only execute after the evaluation of the IF condition. Branch speculation involves the movement of the instructions in the THEN block or the ELSE block (or both blocks) above the IF condition and then the speculative execution of the moved instructions. Through branch speculation, the compiler can take advantage of any possible overlap between the execution of the instructions determining the branch condition and the instructions in the THEN or ELSE blocks.

As discussed in Section 1.1, branch speculation greatly increases the amount of exploitable ILP within non-numerical applications because these applications contain a high frequency of conditional branches which depend upon run-time information. Two recent studies on the limits of ILP illustrate the severity of the control dependence constraints within non-numerical applications compiled for RISC machines. In the limit, Wall [1991] found that his non-numerical applications contained less than 2 independent instructions per cycle (IPC) without branch speculation and approximately 4–6 independent IPC with branch speculation. Lam and Wilson [1992] found that their non-numerical applications contained a harmonic average of 2.1 independent IPC without branch speculation and a harmonic average of 6.8 independent IPC with branch speculation.[1] Both studies generated the ILP results with branch speculation by speculating on only one direction of each branch. Without branch speculation, the ILP results are especially discouraging considering that both studies completely removed storage conflicts, both studies assumed perfect memory disambiguation, both studies allowed for the parallel execution of (at least) 256 instructions, and both studies assumed a machine model where all instructions execute in a single cycle. Obviously, even in this ideal world, non-numerical applications require branch speculation to achieve any sizeable improvement in the amount of exploitable ILP.

The studies by Wall [1991] and by Lam and Wilson [1992] allowed for the movement of any instruction above its control dependent branch. Boosting is an architectural mechanism which fully supports this unrestricted model of speculative execution. This section describes how boosting fully supports branch speculation. First though, Section 2.1.1

1. Though the studies contained some of the same applications, Lam and Wilson's numbers are slightly higher because they removed some of the true data dependences in the applications (for example, they employed loop unrolling to remove the dependence chain caused by the induction variable update).

presents the requirements for the buffer-and-undo mechanism in branch speculation. Section 2.1.2 and Section 2.1.3 use the elements of this presentation to describe the specifics of boosting for branch speculation. Section 2.1.4 concludes this section with a comparison between boosting and other published approaches which reduce the control dependence constraints on instruction scheduling.

2.1.1 Achieving branch speculation

To fully understand why we require a buffer-and-undo mechanism in hardware, we describe what happens to the program semantics if we do not have this buffer-and-undo mechanism. Even if we assume that the movement of an instruction from below to above its control dependent branch preserves the correctness of the operands of the moving instruction, the execution of this instruction can still violate the program semantics in two ways. The combination of these two possible violations results in the four types of speculative movement which are graphically illustrated in Figure 2-1.

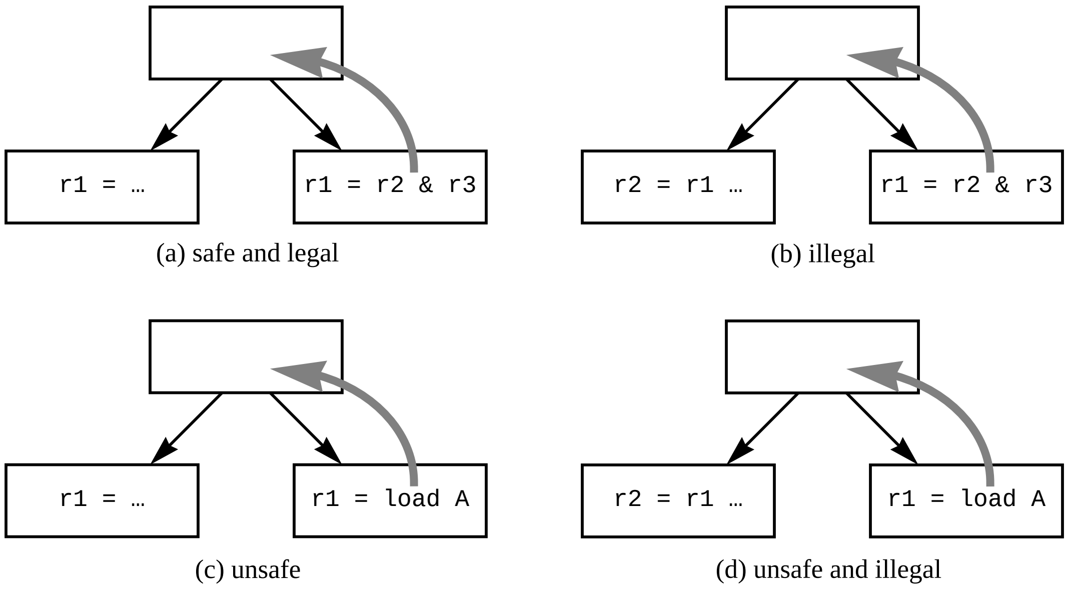


Figure 2-1: Types of speculative movement.

We say that a conditional branch is *correctly predicted* for a speculative operation if the basic block from which the speculative operation was moved is executed after the branch is executed; otherwise, we say that the branch is *incorrectly predicted*. Now, we say that a speculative movement is *illegal* if the speculative operation writes to a location whose previous value is needed by some instruction when the branch is incorrectly predicted. Illegal speculative movement can be thought of as violating a true data dependence constraint along the incorrectly-predicted path of the branch. Figure 2-1b is an example of an illegal speculative movement. We say that a speculative movement is *unsafe* if the execution of the speculative operation can cause an exception to occur. The exception signalled by an unsafe speculative execution should only occur if the branch is correctly predicted. Figure 2-1c is an example of an unsafe speculative movement since the load operation can cause a memory fault. A speculative movement can obviously be both unsafe and illegal as in Figure 2-1d. To preserve program semantics though, a speculative movement should only result in execution that is *safe and legal* as in Figure 2-1a.

Without hardware support to guarantee program correctness under speculative movements, the compiler takes all the responsibility for ensuring that the semantics of the program are maintained, independent of the run-time conditions. In other words, the compiler ensures that the worst-case effects of the speculative instructions do not adversely affect the program state when the branch is incorrectly predicted. This requirement constrains the code motions available to the compiler. The compiler, of course, can perform any speculative movement that is safe and legal. The compiler may overcome some speculative movements that are illegal by renaming the destination register of a speculative operation so that it does not conflict with the set of registers that are needed (i.e. the set of register that are *live*) on the incorrectly-predicted path of the branch. This renaming can require extra instructions later to select between multiple reaching values [Ebcioğlu and Nakatani 1989]. Register renaming, however, does not overcome speculative movements that are illegal due to a dependence through a memory location. Furthermore, a compiler can never transform an unsafe speculative movement into safe speculative execution, and thus a compiler alone cannot support the general movement of instructions above their control dependent branch.

One simple way to guarantee program correctness for all speculative movements is to include extra buffering in the hardware which either holds the effects of the speculative operations or backs up the state that was displaced by the speculative operations. This is how the dynamic instruction scheduling solutions discussed in Section 1.2.2 support all speculative movements. Hardware buffering can postpone the effects of a speculative operation (including any speculative exception) until the hardware resolves the speculative condition. If all branches that a speculative operation depends on are correctly predicted, the hardware updates the non-speculative state of the machine with the buffered effects of that speculative operation.

A *commit* of a speculative operation is the updating of the non-speculative state with the state and side effects of the speculative operation. If any dependent branch for a speculative operation is incorrectly predicted, the machine simply discards the state and side effects of that speculative operation. A *squash* or *nullify* of a speculative operation is the throwing away of the state and side effects of the speculative operation. We say that the hardware performs the commit action at the *commit point*. From an architectural point of view, this commit point in branch speculation occurs immediately before the machine executes any instructions in the predicted target basic block of the last dependent branch. The hardware performs the squash action at any incorrectly-predicted branch.

2.1.2 Boosting

Boosting is our architectural mechanism for unrestricted, hardware-assisted speculative execution. Boosting reserves bits in the instruction word so that the compiler can indicate which instructions are speculative. Thus, whenever the compiler moves an instruction above a control dependent branch, the compiler may *label* this speculative instruction as a *boosted instruction* (by correctly setting the reserved bits). This labelling encodes all the control dependence information needed by the hardware so that the hardware can determine when the effects of the boosted instruction are no longer speculative. For branch speculation, the labelling indicates which branch or branches the boosted instruction are the control dependent upon, and the labelling indicates the predicted direction of each of these branches.

The hardware support for boosting consists of extra buffering in the processor which holds the effects of a speculative operation. The compiler relies on the hardware to track the speculative condition for this operation and to properly manage the speculative and non-speculative state. This hardware support ensures that the semantics of a program is not violated by a boosted operation when that operation's speculative condition is incorrect. With boosting, we convert all of the types of speculative movement in Figure 2-1 into safe and legal speculative execution.

An instruction that has been speculatively moved above *n* branches and labelled with the control dependence information for these *n* branches is referred to as an instruction that is boosted *n* levels. Figure 2-2 contains an example of the most general form of boosting (which is not necessarily the best form to implement). The instruction `i2` in Figure 2-2 is an example of an instruction that is boosted two levels; this is indicated by adding a ".BRR" suffix to the instruction destination. A labelling of ".BRR" indicates that the instruction is dependent upon the next two branches going RIGHT. In this example, the number of Rs or Ls that follow the B indicate the level of boosting while each R (RIGHT) or L (LEFT) indicates the direction of the dependent branch. In general, an independent branch can be included in the sequence by inserting an X (DON'T CARE). A boosting suffix on a destination register implies that a future value has been generated for that register. A boosting suffix on the destination of a memory store operation implies that a future value has been generated for that memory location. In general, a boosting suffix names a readable and writable location for future values, and thus the sources of a boosted instruction may also have boosting-level suffixes as in the base register of instruction `i2`.

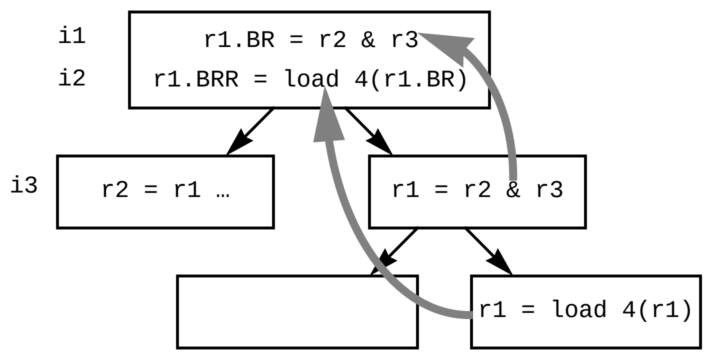


Figure 2-2: Boosting example.

Even though the effects of a boosted instruction are accessible by other instructions boosted along the same path, the speculative effects do not update the non-speculative state until after the execution of the last branch upon which the instruction depends. In other words, the result of instruction `i1` in Figure 2-2 is accessible to instruction `i2`, but the result returned by the load instruction is not committed to the non-speculative state (i.e. the value in `r1.BRR` is not accessible by the name `r1`) unless both branches in Figure 2-2 are correctly predicted. If either branch is incorrectly predicted, the effects of the load operation are prevented from affecting the non-speculative state. With these semantics, the effects (including the side effects) of the boosted operations only affect the non-speculative state if the flow of control would have executed those instructions anyway.

In terms of the types of speculative movement discussed in Figure 2-1, boosting effectively renames registers (`r1.BR` is different from `r1`) so that speculative movements that would have been illegal are now legal. Since the hardware is responsible for the commit action, there is no explicit instruction that transfers the value in the speculative register name (e.g. r1.B1) into the non-speculative register name (e.g. r1), and thus, boosting is more efficient than software renaming from an instruction count perspective. In addition to postponing the writing of the non-speculative register file, boosting postpones all of the speculative side effects so that speculative movements that would have been unsafe are now safe. That is, it postpones the writing of memory and the signalling of exceptions until the commit point. At the commit point, all the boosted side effects update the non-speculative state.

The most general form of boosting described above requires hardware exponential in the boosting level since speculative state is needed for each possible branch prediction path. To limit the hardware to a more reasonable level, the compiler should boost instructions so that they are speculative only on the most-frequently taken direction of a branch. This restriction is reasonable because the ILP studies discussed in Section 2.1 show that speculation on one direction of each branch has the potential to greatly increase the amount of exploitable ILP within a non-numerical application (we use this restriction throughout the rest of the thesis). Since boosting now applies to the most-frequently taken direction of each branch, the branch instructions can encode the prediction information (i.e. predicted-as-taken which is indicated by a ".t" suffix on the branch opcode or predicted-as-not-taken which is indicated by a ".n" suffix), and each boosted instruction can simply indicate that it is dependent upon the next *n* conditional branches (e.g. the labelling of the destination register of instruction `i2` in Figure 2-2 is simplified from ".BRR" to ".B2"). To simplify matters further, once an instruction is boosted to indicate dependence upon a conditional branch, that boosted instruction is assumed to be control dependent upon all subsequent branches it is moved above. By encoding the boosting level as a count of the number of these control dependent branches, the hardware can easily reconstruct the control dependence information. That is, a boosted instruction of level *n* is control dependent upon the execution of the next *n* conditional branches, and this boosted instruction is committed only if all of the next *n* conditional branches are correctly predicted. This constraint makes the boosting information easier to encode, and the hardware simpler to build.

The interaction of these boosting semantics with the global instruction scheduler is discussed in Chapter 3, while the implementation of the hardware mechanisms that support these boosting semantics and the smearing of the architectural commit point onto a pipelined processor are discussed in Chapter 4. Basically, this hardware consists of extra register file space, extra store buffering, and a mechanism to postpone speculative exceptions. The extra register file space and store buffering are relatively straightforward, though their implementation is somewhat complicated by a pipelined processor design. Exception processing, on the other hand, is tricky even in a simple, non-pipelined world.

2.1.3 Handling exceptions

In general, there are two types of exceptions: those that require handling and restart of the program (non-fatal exceptions) and those that terminate a program (fatal exceptions). Exception processing for non-fatal exceptions involves three basic mechanisms: *signalling* that the exception occurred; *handling* of the exception to correct the problem; and *restart* of the process after proper handling. Exception processing for fatal exceptions involves only the first two of these mechanisms: signalling that the exception occurred; and handling of the exception to kill the program. Obviously, exception processing for fatal exceptions is a special case of the processing of non-fatal exceptions. This subsection describes our approach which handles both of these types of exceptions in a processor that supports branch speculation. Our approach incurs very little overhead on the hardware, the software, and the running time of the application.

Since exception processing is quite disruptive to a program's execution, the ideal situation is to only do that exception processing which is necessary and to do that necessary processing as fast as possible. This implies the following two observations for a processor with branch speculation. First, the processor should only signal those speculative exceptions that will commit. In this way, the execution time of a program is never increased because of unnecessary speculative exceptions. Second, the exception handler in this processor should have access to the branch speculation mechanism. Branch speculation can improve the performance of the exception handler in the same way that it improves the performance of the application code.

Ensuring that a processor with branch speculation only invokes exception processing when absolutely necessary is straightforward. Just as the hardware can buffer the other side effects of speculative instructions, the hardware can suppress and buffer the signalling of a speculative exception until the commit point. In fact, a one bit queue is sufficient to guarantee the signalling of a speculative exception (a queue is necessary to support multiple levels of boosting). As long as the exception is fatal, this one bit queue is a complete solution. Unfortunately, this solution is not adequate for non-fatal exceptions where we need to restart the program after the handling of the exception.

2.1.3.1 Restart from a speculative exception

There are two general classes of solutions for solving this restart problem. The first type of solution signals and handles the speculative exception immediately, before it is determined whether the exception should occur. In this way, the restart problem is identical to the problem of restarting after a non-speculative exception. Speculative exceptions requiring restart are handled immediately, while those requiring program termination are postponed as in the one-bit solution. Regrettably, this solution is not ideal in that it possibly increases the run time of the program due to the signalling of speculative exceptions that should have never occurred. Furthermore, the exception handler cannot use the branch speculation mechanism since it cannot inadvertently (or even intentionally) destroy any of the speculative state. To ensure that the processor does not inadvertently destroy some speculative state, the processor must include a mechanism to inhibit the speculation hardware during the exception handler.

The second type of solution postpones the handling of all speculative exceptions until the commit point. By waiting till the commit point, an approach of this type ensures that no extra speculative exceptions are signalled; yet by waiting, the approach runs the risk of corrupting a large portion of the speculative state. For instance, assume that the compiler boosts two load instructions above a conditional branch, and the second load depends upon the result of the first load (e.g. pointer chasing). If the first load instruction causes a page fault and this speculative exception is postponed, then the result of second speculative load is trash. Any approach based on the postponement of the speculative exception must therefore not only re-execute the speculative instruction that caused the exception, but it must re-execute all subsequent instructions that depended upon the excepting instruction. Furthermore, this type of an approach must guarantee that the operands for these instructions are still available so that processor can re-execute them, and it must not re-execute any non-speculative instructions between the excepting instruction and the commit point since these non-speculative instructions might not be idempotent.

Our approach to handling speculative exceptions does postpone the handling of all speculative exceptions until the commit point. Our approach is based on the belief that the handling of a speculative exception will only occur infrequently (no more often than a non-speculative exception), and thus it is acceptable to slightly increase the time it takes to handle a speculative exception if this penalty greatly simplifies the mechanism for speculative exceptions. The key observation in our approach is that though the hardware is very good at postponing the signalling of a speculative exception, it is very bad at determining what instructions need to be re-executed and at ensuring that their operands are still available. The compiler on the other hand is very adept at analyzing dependences and scheduling for operand availability. In general, our approach relies on the compiler to ensure the operand availability of any instruction that might need re-execution and to generate a block of code that rebuilds the corrupted state. The hardware in our approach simply postpones the signalling of the speculative exception until the commit point.

The hardware in our solution consists solely of the one-bit queue mentioned earlier. At a commit point, any outstanding speculative exceptions are signalled, and the speculative state is discarded. Discarding the entire speculative state simplifies the determination of what instructions the processor needs to re-execute.[2] That is, the processor must re-execute any boosted instruction that was speculative upon the conditional branch which signalled the speculative exception. To simplify the re-execution of these instructions, the compiler (during scheduling) squirrels away a compact copy of the boosted instructions to re-execute. This block of instructions is called the *recovery code* since it recovers the speculative state that should have been committed. Since the compiler knows exactly which boosted instructions depend upon a conditional branch, the compiler can easily generate recovery code for each conditional branch. Furthermore, the compiler can monitor anti-dependences during instruction scheduling to ensure that the operands for the instructions in the recovery block are still available during the restart process. Section 3.5 discusses this scheduling constraint in more detail.

In parallel with the discarding of the speculative state, the processor invokes an exception handler that was generated by the compiler (it exists in the program’s text segment). Every type of speculative exception invokes the same single exception handler at this point, and thus the single bit queue in hardware is sufficient to indicate an outstanding speculative exception. The pseudo-assembly-code for this handler is listed in Figure 2-3. The handler simply uses the address of the commit point as an index into a jump table (also created by the compiler). The indexed jump table entry points to the recovery code for the branch that just tried to perform a commit. The speculative exception handler completes by jumping to the indicated block of recovery code.

So far, the cause for the original exception has not been handled, and thus the exception will reoccur when the copy of the boosted instruction is re-executed in the recovery code. Yet, this copy of the boosted instruction is now a non-speculative instruction (i.e. the processor “committed” the speculative state), and so, the processor handles this exception in

2. Discarding all of the speculative state on an exception has another important advantage; it allows the exception handler to use the branch speculation mechanism since there is no outstanding speculative state to worry about. This advantage shows that our approach can achieve both of the goals that were put forth under the ideal situation.

```
load  $r1,$expPC     # get address of commit point
subi  $r1,$r1,start  # translate addr to offset
srl   $r1,$r1,n      # translate word addr into packet addr
addi  $r1,$r1,jtable # calculate index into jump table
load  r1,0($r1)      # get address of recovery code
jump  $r1            # jump to recovery code
```

Figure 2-3: Pseudo-assembly-code for speculative exception handler.

the normal manner. The recovery code completes by unconditionally jumping to the predicted target of the original conditional branch.

Figure 2-4 illustrates the handling of a speculative exception on a boosted instruction using our recovery-code approach. In Figure 2-4, the boosted instructions are indicated through a ".B*n*" suffix (e.g. instruction `i7.B2` is boosted two levels). The instruction `i3.t` is the branch instruction. The branch is predicted to be taken, and label `L2` is the target of the branch. For this example, assume that the architecture has no branch delay slots and that instruction `i8.B1` is a load instruction which causes a page fault. Once the branch instruction `i3.t` is executed and found to be correctly predicted, a generic boosted exception is signaled, and the machine vectors to the speculative exception handler at the top of the program text segment. This handler uses the address of the exception program counter (`@i3`) to find the address of the recovery code (`L.i3`). Now, the cause of the speculative exception was not yet handled, but it will be when the instruction that caused the exception is re-executed in the recovery basic block (when `i8` in Figure 2-4 is executed). This instruction is now a non-speculative instruction (since the earlier branch was correctly predicted), and the exception can be handled like any other non-speculative exception. The recovery basic block ends in an unconditional jump to the predicted target of the earlier branch (`L2`), and execution continues. As the example in Figure 2-4 illustrates, this solution works for multiple levels of boosting.

The major cost of our recovery-code approach is that the size of the object file has increased in order to accommodate the jump table and the recovery code. The size of the increase depends upon the amount of speculative code in the program text segment and on the size of the parallel issue, but in the worst case, the increase is only about 100%. To understand this reasons for this increase, let us consider a very aggressive scheduling technique in which every other instruction in the program segment is a speculative instruction. The recovery code thus adds a 50% overhead to the size of the original code. For non-numerical code with branches every 4 to 5 instructions, it is not cost-effective to build a hash table for the jump table. Consequently, the jump table has an entry for every packet

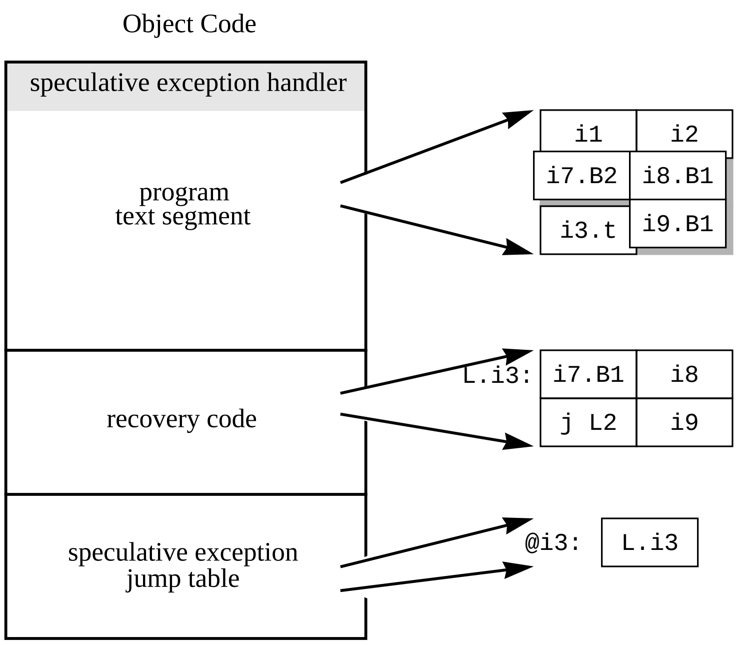


Figure 2-4: Speculative exception handling in TORCH.

of parallel instructions, and the smaller the issue size, the larger the jump table overhead. For a 2-issue superscalar processor, the jump table adds another 50% overhead to the size of the original code.

The increase in object code size requires more disk space to hold the program, but the program's instruction cache miss rate does not change significantly. This is because the majority of the execution time is spent in the program text segment, and the size of this segment is unaffected by our solution. The recovery code only enters the instruction cache during exception processing. The other cost is the overhead of the speculative exception handler. This handler takes approximately 10–12 cycles to execute, and thus adds minimal overhead to the cost of an exception.

Most exception processing routines, like a page fault handler, occur infrequently, and when they do occur, they run for a long time. Because of these facts, the overhead of our recovery-code approach is quite acceptable. In some systems though, there are exception handlers which are performance sensitive. In MIPS systems for example, TLB misses are handled in software, and the MIPS designers have taken great care in optimizing the performance of this exception handler. If the overhead of our solution is deemed too great for these performance-sensitive handlers, one could always special case these speculative exceptions and handle them immediately, as in the first class of solutions.

2.1.3.2 Restart from a non-speculative exception

Up to this point, we have focused on the handling of and restart after a speculative exception. This covers only half the problem though. The inclusion of speculative state and the adoption of hardware mechanisms such as boosting also impact the handling of and restart after a non-speculative exception. In an ideal world, the addition of the speculative techniques should not affect the coding of the non-speculative exception handlers; yet, the non-speculative exception handlers should have access to the speculative techniques so that their performance can also improve. What these goals imply is that the speculative state is non-essential state—state that is not saved across process switches. The machine discards the speculative state at every exception. By discarding the speculative state, the non-speculative exception handlers do not have to worry about maintaining the existing speculative state (e.g. no saving and restoring of speculative registers), and these handlers are free to use the speculative mechanism.

An unfortunate consequence of this scheme is that the non-speculative exception handler has destroyed the program's speculative state, and the program might need this speculative state. The solution to this problem is simple: the return-from-exception instruction for a non-speculative exception sets the speculative exception bit. If the program then tries to commit some discarded speculative state, the machine will simply go back and rebuild the speculative state using the speculative exception mechanism discussed above. A designer can minimize the overhead to a non-speculative exception by only setting of this speculative exception bit if some speculative state existed at the point of the non-speculative exception.

In summary, our recovery-code approach provides a very simple and efficient mechanism to handle both speculative and non-speculative exceptions. This approach also provides the architecture with precise exceptions [Hennessy and Patterson 1990]. That is, at any non-speculative exception, all the non-speculative instructions before the faulting instruction have completed, and all the instructions after the faulting instruction (including the instructions that were speculatively executed) can be restarted.

2.1.4 Existing mechanisms

Boosting is just one example of an architectural mechanism for speculative execution which supports branch speculation. The rest of this section compares boosting with two other architectural techniques for removing control dependence constraints: guarding [Hsu and Davidson 1986] and non-excepting instructions [Colwell et al. 1987, Ebcioğlu 1988, Chang et al. 1991b].

Guarding predicates a control dependent operation with its dependent branch condition, and this predicated operation is called a guarded (or conditional) instruction. If the predicate on a guarded instruction evaluates to true, the execution of the guarded instruction is allowed to complete; if the predicate evaluates to false, the execution of the guarded instruction is squashed. The guarding predicates can be quite complex and can encode the control dependence information for multiple branches. Guarding also allows for the possible elimination of branch instructions by guarding the instructions from both paths of the branch. Another advantage of this technique is that the required speculative state is very small, and it is held in the pipeline bypass registers that are already in the machine. Yet, guarding does not support unrestricted speculative execution since the scheduling of guarded operations is constrained by the availability of the dependent branch condition.

Non-excepting instruction architectures rely on hardware mechanisms to handle unsafe speculative movements and on software renaming to handle illegal speculative movements. Non-excepting instruction architectures label unsafe speculative movements as non-excepting instructions. The semantics of a non-excepting instruction is that this instruction never signals an exception. If it causes an exception, it simply generates a polluted result. Eventually, some later (regular) instruction may try to use this polluted value, and it is at this time that the exception is signalled. In this way, non-excepting instruction architectures can detect exceptions on speculative operations. These polluted values are often implemented by building a tagged-data architecture, and by carrying the address of the “excepting” non-excepting operation in the data field of the polluted operand, these architectures can indicate which instruction originally caused the exception.

Mahlke et al. [1992] describe one example of a tagged, non-excepting architecture which uses an instruction scheduling technique called sentinel scheduling. In their architecture, a bit in the instruction word differentiates the non-excepting version of an instruction from the excepting version, and an extra bit is placed on every register to indicate the presence of an outstanding speculative exception. When a non-excepting instruction causes an exception, the processor places the address of this non-excepting instruction in the result register of the instruction, and it sets the tag bit of that register to indicate an outstanding speculative exception. Any later non-excepting instruction that uses this result will simply pass on the exception in their tagged result register. For the machine to signal this speculative exception, some non-speculative instruction must try to use one of these excepting results. This non-speculative instruction is the sentinel in sentinel scheduling. Without this sentinel, detection of the original speculative exception is impossible.

Sentinel scheduling works quite well as long as the program does not need to restart after the handling of the postponed exception. For non-fatal exceptions, sentinel scheduling has the same restart problems that we discussed in Section 2.1.3.1 (i.e. how does one determine which instructions to re-execute and how does one ensure that the operands for these instructions are still available). Sentinel scheduling partially solves the restart problem by restricting the compiler's anti-dependence distance, i.e. the compiler cannot schedule two anti-dependent operations within *n* cycles of each other. This restriction ensures that an operand will be available for restart if the exception is signalled within *n* cycles, but it also limits the compiler's freedom in rearranging the code. For the unrestricted movement of non-excepting instructions, sentinel scheduling could rely on our recovery-code approach to exception processing described in Section 2.1.3.

In summary, guarding and non-excepting instructions are restricted forms of speculative execution. Guarding is basically a restricted form of general boosting where the compiler only boosts instructions into the pipeline "shadow" of the evaluation of the dependent branch condition. Non-excepting architectures are similar to boosting architectures in that they rely on hardware mechanisms to overcome unsafe speculative movements. Yet, they are different from boosting architectures because they rely on software renaming to overcome illegal speculative movements. Overall, boosting represents an approach which can fully support an unrestricted model of speculative execution through the addition of general buffering in the processor, while the other mechanisms opt for a restricted approach which adds only a small amount of specialized buffering (e.g. exception bits).

2.2 Building mechanisms for speculation

Branch speculation with boosting is one example of an opportunistic instruction scheduling technique because it allows the compiler to safely speculate on the run-time direction of a conditional branch. The general idea behind opportunistic instruction scheduling is to speculate on any piece of run-time information. The zeroth step in this process is to determine what run-time question is important enough to warrant a speculative technique. This step, though straightforward, encompasses all the design steps listed in the next paragraph. A designer must formulate a run-time question, design the technique and the mechanism for speculative execution, and then evaluate the cost of the implementation against its performance benefits. Thus, this step is more than just picking another compiler constraint to overcome; a designer must complete and analyze the opportunistic instruction scheduling technique to fully answer the question posed by this step. Chapter 4 presents the data necessary to answer this question for branch speculation with boosting.

With the implications of this initial step understood, the development of the speculative technique consists of support for the following four steps:

(1) compile-time determination of where speculation is needed;

(2) complete separation of speculative and non-speculative state;

(3) run-time determination of whether the speculation was correct; and

(4) run-time determination of correct program state after completion of speculative operation and run-time check.

In the first step, the compiler determines whether the current instruction scheduling situation will benefit from speculation. To analyze the situation, the compiler must approximate the potential performance benefit available through speculation, and it must compare this against the probable performance penalty incurred from incorrect speculation. If the benefits outweigh the penalties, then the compiler goes ahead with the speculation.

Once the compiler has determined where to apply speculation, the compiler encodes these assumptions and passes this information to the hardware. The most important aspect of this speculation is expressed in step two: the speculative state is kept separate from the non-speculative state. This separation provides an easy way to recover the non-speculative state at any time. This separation is the key to fast state recovery after an incorrect speculation, and it is the key to the simple handling of all exception processing.

Step three involves the run-time checking of the compile-time assumptions against the run-time information. Though some run-time checking can be accomplished through software checks [Nicolau 1989], the most comprehensive and efficient method of performing run-time checks is in hardware. As discussed in Section 1.2, only the hardware can effectively capture and efficiently check every operational side effect.

As long as the compile-time assumptions match the run-time information, the speculation is correct, and step four only consists of the update of the non-speculative state by the speculative state. If however the assumptions do not match the run-time information, either the speculation was incorrect or an exception occurred which corrupted the speculative state. In both cases, the speculative state is useless and therefore discarded. For the case of incorrect speculation, step four usually consists of the discarding of the speculative state and continuing with execution. For the case of an exception or in the case where the incorrect speculation requires recovery, step four requires the inclusion of a method for rebuilding the discarded state.

For example in branch speculation with boosting, step one requires the compiler to determine which conditional branches it wishes to move code across. Once the compiler creates a speculative operation through this upward code motion, the compiler marks this speculative operation as a boosted instruction. The marking consists of extra information included in the boosted instruction's encoding which passes control dependence information to the hardware. Step two is accomplished through extra buffering for the speculative values and speculative side effects so they do not interfere with the existing non-speculative state. In step three, the hardware checks to see if the static branch prediction matches the dynamic action of the branch. On a correct prediction, step four requires the hardware to commit the speculative values into the non-speculative state. On an incorrect prediction, the hardware squashes the speculative state.

The proper handling of exceptions is also included in step four. Section 2.1.3.1 described how we handle exceptions on boosted instructions. The basis of this method is the ability to rebuild the discarded state; the machine must recover this state because the branch was correctly predicted. This ability also provides a simple method for minimizing the impact of speculation on non-speculative exceptions. As explained in Section 2.1.3.2, a non-speculative exception destroys the existing speculative state so that the speculative state does not add overhead to the handling of the non-speculative exception and so that the exception handlers can use the speculative mechanisms. A consequence of this decision is that all mechanisms for speculation execution that adhere to this opportunistic instruction scheduling framework can support precise exceptions.

2.3 Speculative memory disambiguation

Static memory disambiguation determines when the compiler is able to reorder a set of memory operations to improve the instruction schedule. An ideal memory disambiguator either returns the answer YES (the memory addresses alias) or NO (the memory addresses never alias). Often in pointer-intensive code however, static memory disambiguation fails and the disambiguator returns the answer UNKNOWN. That is, the disambiguator is unable to statically determine whether two memory operations will ever access the same memory location. When this occurs, the compiler assumes that it cannot reorder the two memory operations; in other words, the compiler is conservative during instruction scheduling whenever static memory disambiguation fails.

In this section, we use the opportunistic instruction scheduling framework of the last section to develop a technique for speculative memory disambiguation. This technique allows the compiler to assume that two memory addresses do not alias[3] when the disambiguator fails. Under speculative memory disambiguation, the compiler is free to reorder the memory operations to extract more ILP. To simplify this discussion, we describe a mechanism for speculative execution that only supports the reordering of memory operations within a basic block. This base mechanism is easily extended, and it can be coupled with the boosting mechanism which supports branch speculation. In fact, speculative memory disambiguation relies on a mechanism for speculative execution that is very similar to the boosting mechanism of Section 2.1.2, except that this implementation of our mechanism for hardware-assisted speculative execution describes the compile-time assumptions for speculative memory disambiguation instead of those for branch speculation.

Figure 2-5a contains a basic block of instructions where the disambiguator cannot determine the relationship between the address `B` in instruction `i2` and the address `A` in instruction `i3`. Without speculative memory disambiguation, the compiler is forced to assume that the addresses could alias. If the loads in this architecture are delayed by one cycle, the best schedule for this block takes five cycles to execute. If the compiler was sure that the memory addresses were always different, then the compiler could reorder the instructions in the basic block as in Figure 2-5b and save one cycle per execution of the basic block.

```
i1   r2 = r3 + r4
i2   store B = r2
i3   r1 = load A
     <stall>
i4   r2 = r1 + r2
```

(a) without speculation

```
i3   r1.S = load A
i1   r2 = r3 + r4
i4   r2.S = r1.S + r2
i2   store B = r2
```

(b) with speculation

```
i3′   r1 = r2
i4′   r2 = r1 + r2
      jmp (@(i2)+1)
```

(c) recovery code

Figure 2-5: Example of speculative memory disambiguation.

3. One could also assume that the memory references do alias and optimize for that occurrence.

In opportunistic instruction scheduling, the compiler labels the speculative operations with the dependence information that the hardware needs to perform a run-time check of the compile-time assumption. In the case of speculative memory disambiguation, the compiler assumes that two memory operations are not dependent, and it creates speculative operations by moving the later memory operation and its dependent successors before the earlier memory operation. In Figure 2-5b, instructions `i3` possibly depends on instruction `i2`, but the compiler assumes that they are independent. The compiler has moved instructions `i3` and `i4` above instruction `i2`, and it has labeled `i3` and `i4` as speculative operations. These operations are dependent upon a run-time comparison of the load address in instruction `i3` and the store address in instruction `i2`.

The labelling is indicated by adding a '.S' suffix onto the destination register of the speculative operations (e.g. `r1.S` of instruction `i3` in Figure 2-5b). This is analogous to the labelling used in branch speculation, and as before, the labelling indicates that the instruction is generating a future or speculative value. Opportunistic instruction scheduling separates this speculative state from the non-speculative state so that the non-speculative state is easy to recover.

Unlike the run-time check in branch speculation, the run-time check in speculative memory disambiguation is slightly more complicated. It requires the hardware to keep track of the address used in the initial speculative load or store operation, and it requires the hardware to compare this address against the address used in the later (possibly dependent) store operation. Returning to the example in Figure 2-5, the hardware compares the load address it remembered from instruction `i3` against the store address in instruction `i2`. If the addresses are different, the speculation was correct, and the non-speculative state is updated with the speculative state (Chapter 4 discusses the specifics of the hardware implementation to satisfy the commit operation which is used by both branch speculation and speculative memory disambiguation). If the addresses are equal, the speculative state is wrong, and the correct non-speculative state must be generated. Notice that this corrective action is different from the squashing mechanism in branch speculation. In this case, the program wanted to do the load, the machine just did the load at the wrong time. Thus, the machine must re-execute the load to obtain the correct datum.

The machine can fix the non-speculative state by using the same recovery-code approach that fixes the state after a speculative exception. The compiler generates some recovery code which re-executes the previously-speculative load or store and any of its dependent operations that were also affected. Figure 2-5c shows the recovery code for the example.

In this recovery code, the compiler has optimized the code sequence by replacing the load instruction (instruction `i3`) with a move instruction (instruction `i3′`). This optimization is possible since the addresses `A` and `B` are identical when the machine is executing in the recovery code.

Up to this point, we have explained the specifics of steps (2) through (4) of Section 2.2 for the development of a speculative memory disambiguation mechanism. Steps (0) and (1), the determination of when and where speculative memory disambiguation is needed, are equally as important as the other steps. For branch speculation, an incorrect speculation simply discards the speculative state. Thus, if the hardware for supporting speculative execution exists, a compiler should always attempt to take advantage of branch speculation because the cost of an incorrect speculation is effectively zero. In speculative memory disambiguation, this cost/benefit tradeoff is not as obvious.

Let us first assume that the hardware already exists to support speculative memory disambiguation. The cost of an incorrect speculation under speculative memory disambiguation is the time it takes to fix the non-speculative state. This time is made up of the time it takes to vector to the recovery code handler plus the time it takes to execute the handler and the recovery code. Though this time is small (on the order of 10–20 cycles), the total cost is significant if the prediction accuracy is low. If the potential benefit is also low (e.g. 1 cycle per execution of the code as in Figure 2-5), the compiler should be fairly certain that the memory operations do not alias. For instance, if the average recovery cost is 15 cycles and the benefit is 1 cycle per execution, then the compiler should speculate if it believes that the memory operations dynamically alias less than 6% ($\approx 1/15$) of the time. In general, the more expensive the recovery cost, the less opportunities exist for speculation; conversely, the less expensive the recovery cost, the more often the compiler should speculate.

2.4 Summary

Opportunistic instruction scheduling is a framework for developing techniques which keep the compiler from making conservative scheduling decisions when it lacks exact dependence information, information that is only available at run time. The basis of opportunistic instruction scheduling is speculation. The compiler speculates on the value of some run-time information and then schedules the code to take advantage of this opportunity. This chapter described two opportunistic instruction scheduling techniques, branch speculation and speculative memory disambiguation. Because control dependence is such a large constraint on the amount of exploitable ILP in non-numerical applications, branch speculation is the more important of these two opportunistic instruction scheduling techniques, and thus the later chapters of this thesis focus specifically on branch speculation.

Each opportunistic instruction scheduling technique relies on an architectural mechanism which implements speculative execution with safe and legal semantics. This chapter described boosting, an unrestricted implementation of hardware-assisted speculative execution. Boosting contains three important ideas which ensure a complete and an efficient implementation: information transfer, separation of state, and partitioning of functionality. Information transfer refers to the fact that the compiler informs the hardware of its assumptions so that the hardware can easily check the run-time conditions against the compile-time assumptions. Until the hardware verifies that the compile-time assumptions are correct, the speculative and non-speculative states are kept separate. This separation of state simplifies the recovery process without complicating the commit process. The commit process is handled in the hardware so that the machine state is updated quickly and efficiently on a correct speculation. The commit process is the desired and expected operation so that the cost of a hardware method is justified. The recovery process, on the other hand, is an infrequent occurrence (hopefully). This process, which occurs after an exception (and sometimes after an incorrect speculation as in speculative memory disambiguation), is handled through a combination of hardware and software methods. That is, the hardware captures the exception signal, while the compiler generates and compacts the recovery code. The next chapters focus on the compiler support, hardware costs, and performance benefit of boosting for branch speculation.

Chapter 3

Global Instruction Scheduling

In our approach to instruction scheduling with speculative execution, the compiler is responsible for analyzing the program dependences to uncover ILP and for scheduling the code to take advantage of that exploitable ILP. The overriding goal of this compilation system is to exploit the available ILP within an application without adversely affecting the instruction count of that application. In order to effectively exploit ILP, we developed a new global instruction scheduling algorithm. A global instruction scheduler searches for and takes advantage of ILP across basic block boundaries, and thus a global instruction scheduler will benefit from an inter-block mechanism such as boosting. This chapter provides an overview of our global scheduling algorithm (more details of the algorithm can be found in the appendices). The emphasis of the overview is on the routines which uncover the exploitable ILP and which perform the code motions across basic block boundaries. These routines are the ones that benefit from boosting, and these are the ones that affect the instruction count of the scheduled application.

3.1 Background

Before we discuss the topic of global instruction scheduling, this section begins with a brief discussion of the issues and existing techniques for the scheduling instructions within a basic block. Basic block schedulers are simpler to understand than global schedulers because basic block schedulers do not deal with control dependence. Furthermore, a review of the prevalent approaches to global instruction scheduling show that a basic block scheduler usually lies at the heart of an effective global scheduler. Thus, an understanding of the tradeoffs in a basic block scheduler will aid us in the understanding of the important tradeoffs in our global scheduler.

3.1.1 Issues in basic block scheduling

A basic block scheduler only considers the interactions between instructions within a basic block, and because of this restriction, a basic block scheduler need only consider the effects of data dependence constraints and resource constraints. Since every instruction within a basic block is executed once a basic block is entered, a basic block scheduler never considers the effects of control dependence. Gross [1983] and Gibbons and Muchnick [1986] present a broad survey of the early techniques for basic block scheduling. This subsection reviews the important issues in a basic block scheduler, and it presents the choices we made for our basic block scheduler.

The data dependence relationship between instructions within a basic block is often represented by a directed acyclic graph (DAG). There is a node in this graph for each instruction in the basic block. A directed edge leads from node `n1` (*start* of the directed edge) to node `n2` (*end* of the directed edge) if and only if instruction `i2` (represented by node `n2`) is data dependent upon instruction `i1`. That is, instruction `i2` cannot execute earlier than instruction `i1`. The edges in the DAG indicate all of the ordering constraints between instructions in the basic block (independent of whether those orderings are due to dependences through registers, memory, or special structures such as condition code registers). DAGs are constructed either by scanning backward (as done by Gibbons and Muchnick [1986]) or by scanning forward (as done by our scheduling algorithm) across the instructions in the basic block.

During construction of the DAG, each directed edge is labelled with a number. This number indicates the delay (in cycles) associated with the ordering constraint. The delay on each edge can be different, and the number depends upon the type of data dependence constraint that the edge represents. The delay associated with a true dependence edge is simply equal to the latency of the operation at the start of the directed edge. The delays associated with an anti-dependence or output dependence edge are characterized by the structure of the pipeline in the processor. For the pipeline organizations in this thesis, each pipeline reads all of the operands for the instructions issued in the same cycle before it writes any of their results. Thus, the delay associated with an anti-dependence edge is zero. That is, an anti-dependent operation can issue in the same cycle as its dependent partner, but never earlier. The integer pipeline in this thesis is a fixed-length pipeline where all integer instructions flow through the same number and type of pipeline stages. The floating-point pipeline, on the other hand, is a variable-length pipeline where different floating-point instructions execute in a possibly different number of stages. Because of the variable length of the floating-point pipeline, the delay associated with an output dependence edge is the maximum of 1 and (l_1 - l_2 + 1), where l_1 is the result latency of the instruction at the start of the output dependence edge and l_2 is the result latency of the instruction at the end of the output dependence edge. This equation ensures that the overlapped execution of two floating-point instructions still updates the register file in the correct order. As an illustration of these concepts, Figure 3-1 contains a small basic block and its corresponding DAG. In this example, the latency of a load operation is two cycles while the latency of all other integer instructions is a single cycle. The floating-point multiply executes in six cycle while the floating-point add executes in three cycles.[1]

```
i1: r1 = load A
i2: r3 = r1 + 1
i3: r1 = r4 + 2
i4: f5 = fmult(f2,f4)
i5: store B = f5
i6: f5 = fadd(f6,f7)
```

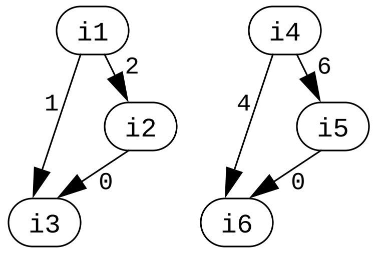


Figure 3-1: Example basic block and its DAG.

Given a DAG for the instructions in a basic block, an instruction is *ready* if all of its data-dependent predecessor instructions in the DAG have been scheduled and their latencies fulfilled. Instruction scheduling on the basic block is performed by repeatedly finding the ready instructions, and then placing those ready instructions as early in the schedule as possible.

Given a machine with infinite resources, it is trivial to construct an optimal schedule for a basic block. One need only perform a topological sort of the DAG. For real machines with limited resources, the construction of an optimal schedule is an NP-complete problem [Gross 1983]. Because of this, instruction scheduling follows some type of heuristic algorithm. Instructions in the DAG are given a priority which heuristically indicates the relative importance of scheduling one instruction before another (Smotherman et al. [1991] describes 26 different proposed priority functions). This priority is used by a heuristic scheduling algorithm to choose among the ready instructions. Davidson et al. [1981] compared a number of heuristic algorithms, and they recommended list scheduling as the best

1. Notice that the delays on the output dependence edges in this single basic block example are covered by the true and anti-dependence edge delays. For DAGs containing a sequence of basic blocks, the use of a register definition may not be in the sequence of basic blocks, and thus we must know the output dependence delay.

compromise. List scheduling generates reasonably short schedules within a reasonable amount of compile time. We use list scheduling to schedule our dependence graphs.

There are many variations on the basic list scheduling algorithm. For instance, one can vary the sophistication of the priority function, but usually some simple function like the maximum number of cycles from the node to the end of the DAG is sufficient. One can also vary the scheduling direction that the list scheduler takes over the DAG. So far, we have described *top-down scheduling* where predecessors in the DAG are scheduled before successors. Top-down scheduling is good for pulling an operation as early in the schedule as possible, leaving holes at the bottom of the schedule. The opposite of top-down scheduling is *bottom-up scheduling*. In bottom-up scheduling, an instruction is considered ready if all its successor instructions have been scheduled and the latency for the candidate instruction is satisfied for all scheduled successors. Bottom-up scheduling tries to push an operation as late in the schedule as possible, leaving holes at the top of the schedule. We use a combination of top-down and bottom-up scheduling to capture the advantages of both approaches (see Appendix A).

Another decision for list scheduling is whether scheduling proceeds on a cycle-by-cycle basis (often called *cycle scheduling*) or on an operation-by-operation basis (often called *operation scheduling*). In cycle scheduling, the scheduler fills instruction cycles in chronological order, from the first cycle to the last for top-down scheduling or from the last cycle to the first for bottom-up scheduling. On each cycle, ready operations are considered for scheduling in priority order, and an ready operation is scheduled only if its required resources are still available in the current cycle. In operation scheduling, the algorithm schedules ready operations in strict priority order; a ready operation at a lower priority is scheduled only after those at the higher priority. For each ready operation, the scheduler finds the earliest cycle for which all data-dependent predecessor results are available (or the last cycle so that its result is available to all data-dependent successors). This cycle is the bound, and the scheduler searches forward (or backward) from this point until a cycle is found with the necessary resources available. For machine models with all instructions having a single cycle latency, cycle scheduling and operation scheduling are equivalent approaches. As instruction latencies become longer and lower priority instructions are able to monopolize resources, operation scheduling will outperform cycle scheduling. Cycle scheduling is easier to code, and it simplifies the integration of register allocation and instruction scheduling. In cycle scheduling, registers are either free or not at the current cycle. Since the large majority of instructions for the machine model considered in this thesis have a single cycle latency, we use a cycle scheduling approach.

The last area of concern in basic block scheduling is how it interacts with register allocation. Traditional register allocation attempts to minimize the number of registers allocated by reusing a register as soon as its previous value is no longer needed [Aho et al. 1986]. Unfortunately, this approach maximizes the number of storage conflicts in the code, and thus reduces parallelism. A better approach for instruction schedulers that are run after register allocation (often called *postpass code scheduling* [Goodman and Hsu 1988]) is to use a round robin approach to assign registers. Our scheduling algorithm is based on a postpass scheduling approach with round robin register allocation.

Alternatively, some compilers employ *prepass code scheduling*. Prepass code scheduling performs code scheduling before register allocation, and it gives the scheduler the maximum freedom to generate a good schedule. Yet, prepass scheduling can create instances in the code where more registers are needed than there are registers available. Consequently, the register allocator adds spill instructions into the schedule which could possibly destroy advantages of prepass scheduling. Chang et al. [1991b] describe an instruction scheduler that runs before and after register allocation to attempt to get the advantages of both pre- and postpass code scheduling. As a more sophisticated alternative, Goodman and Hsu [1988] and Bradlee et al. [1991] have suggested an integrated approach to this phase-ordering problem. In this thesis, we separate the actions of register allocator and instruction scheduler. However, the implementation of our instruction scheduler is capable of scheduling instructions for an infinite register model. In this way, we can bound the performance advantage of an integrated register allocator and instruction scheduler.

3.1.2 Issues in global scheduling

Since a global instruction scheduler moves instructions across basic block boundaries, it must deal with control dependence. The flow of control between basic blocks in a program can be summarized using a *control flow graph* (CFG) [Aho et al. 1986]. A CFG is a directed graph where each basic block in the program is represented by a single node. An edge is drawn from block b1 to block b2 if program execution can proceed from b1 to b2. The gathering of dependence information for a CFG is known as *global dataflow analysis* [Aho et al. 1986]. Global dataflow information is used, as data dependence information is used in basic block scheduling, to ensure that the semantic correctness of the program is maintained during the movement of an instruction across a basic block boundary.

A *global code motion* is any type of inter-basic-block movement. Global code motion relies on a set of rules (or *transformations* [Nicolau 1985]) that govern the movement of instructions across the edges of a CFG. These transformations must always maintain correctness. The transformations define the set of allowable global code motions, and thus, the completeness of these transformations is one indicator of the sophistication of the global instruction scheduler. Architectural mechanisms such as boosting augment the capabilities of the transformations and thus increase the sophistication (and decrease the complexity) of the transformations.

A global code motion often requires the insertion of extra copies of the moving instruction to maintain the semantics of the program. Gross and Ward [1991] refer to these copies as *compensation code*. The amount of compensation code produced during a global code motion is another indication of the sophistication of the transformations. We refer to this attribute as the *spatial efficiency* of the global transformations. Spatial efficiency is a separate issue from the correctness and completeness of the transformations, and as we will see, spatial efficiency can sometimes impact the performance.

Several different global scheduling algorithms have been proposed. The earliest work on global instruction scheduling grew out of the work done on local microcode compaction techniques of the 1970s and early 1980s (see Tokoro et al. [1981] for a comprehensive reference list). The early attempts at global scheduling first scheduled each basic block individually, and then they optimized the program by repeatedly moving instructions between pairs of basic blocks to improve the basic block schedules. The culmination of these iterative scheduling algorithms is Percolation Scheduling [Nicolau 1985] which describes a complete set of semantics-preserving transformations for moving any operation between adjacent blocks. Ebcioğlu and Nakatani [1989] describe an enhanced implementation of percolation scheduling for VLIW machines with conditional evaluation capabilities. Under this iterative approach to global scheduling, a global code motion across a large number of basic blocks will only occur if each of the pair-wise transformations is beneficial. Fisher [1981] shows that this type of an incremental scheme does not always lead to a good global schedule.

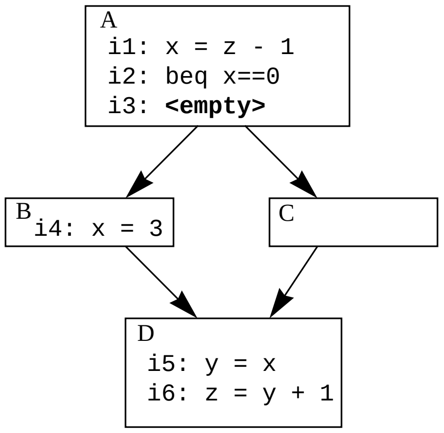
As a result of the finding by Fisher, research into global instruction scheduling has shifted toward global scheduling algorithms that consider the benefit of the entire global motion during instruction scheduling. The structure of the recent global schedulers is similar to the structure of the basic block scheduler discussed in the previous subsection. A global scheduler, like a basic block scheduler, repeatedly finds the set of ready instructions and then heuristically chooses the best of those ready instructions to schedule. The scope of the ready set in the global scheduler is simply larger than the scope of the ready set in the basic block scheduler. A global scheduler searches for ready instructions in a portion of the CFG, instead of just in a single basic block. The size of the scope of the CFG from which the scheduler can search for available instructions defines another key characteristic of a global instruction scheduler.

To generate the pool of ready instructions, a global scheduler finds all the instructions that are *available* for scheduling at a point in the CFG by determining if there is some *set* of global transformations which result in a ready instance of each of the instructions. In other words, the application of some number of global transformations results in an instance (or copy) of the original instruction, and this instance has all of its data-dependent predecessor[2] instructions scheduled and latencies fulfilled. For example, instruction `i4` in Figure 3-2a is available for scheduling in the instruction slot `i3` (the branch delay slot) because we can generate an equivalent instruction schedule (Figure 3-2b) through a particular global transformation along the path `AB`. Because of another set of global transformations along the path `ACD`, instruction `i5` is also available for scheduling in the instruction slot `i3` (Figure 3-2c). Notice that both of these transformations create compensation code in block `B`. Instruction `i6` is not currently available for scheduling in instruction slot `i3` because it is data dependent upon the unscheduled instruction `i5` on all paths between block `A` and block `D`. (The specifics of when an unscheduled instruction is and is not available will become clearer when we later talk about our specific global transformations.)

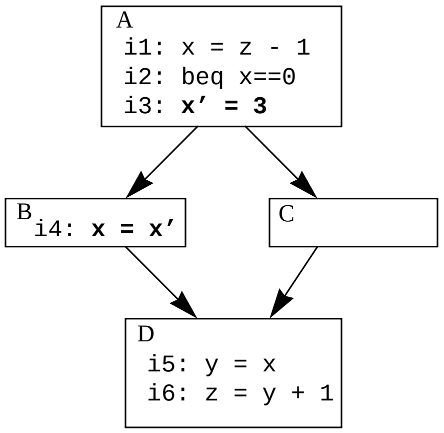
To review, at every point in the generation of an instruction schedule, a global scheduler first finds the available instructions, then uses heuristics to choose which of the available instructions will produce the best schedule, and finally invokes the global transformations to safely move the requested instructions to the current scheduling point. We can categorize the different algorithms for global instruction scheduling by the scope of their availability calculation and by the completeness and spatial efficiency of their global transformations.

In general, the differences in the support for these key aspects are a result of tradeoffs made between scheduler capability and scheduling complexity. For example, a complete set of transformations will allow for the reordering of branch and jump instructions. A complete set of transformations therefore provides a powerful mechanism for not only moving code, but also for changing the structure of the CFG during scheduling. Yet, the

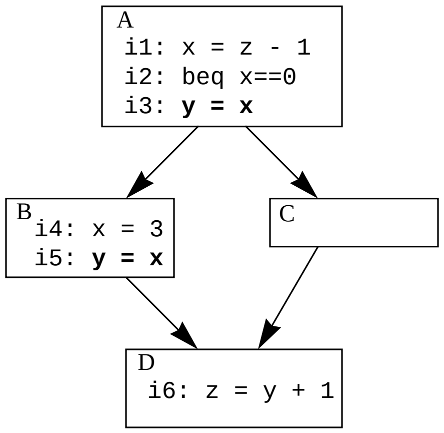
2. We assume top-down scheduling for this discussion. For bottom-up scheduling, the word ‘predecessor’ should obviously be replaced with the word ‘successor’. For the rest of the thesis, we assume top-down, cycle-scheduling unless a distinction is appropriate to the discussion.



(a) CFG before any global code motion.



(b) CFG after global movement of i4.



(c) CFG after global movement of i5.

Figure 3-2: Example of availability.

reorganization of branch and jump instructions can lead to cases of code explosion, and researchers often add special heuristics to minimize code explosion. In another example, the larger the scope of the CFG from which the scheduler can search for available instructions, the better chance the scheduler has of generating the best possible schedule. Yet, the calculation of the available set is at least proportional to the number of basic blocks examined, and thus compile time increases with increasing scope. More importantly, the global movement of an instruction changes the global dataflow information, and these changes must be propagated to the appropriate basic blocks within the scope in order for scheduling to continue. Recalculating all the global dataflow information after each global code motion is prohibitively expensive, and thus some schedulers attempt to incrementally update the global dataflow information [Ebcioğlu and Nicolau 1989]. Still, the complexity of the incremental update limits the scope of the availability calculation. In the end, where the tradeoff between capability and complexity is made is ultimately dictated by the assumptions about the target application domain and the target machine architecture.

3.1.3 Existing global schedulers

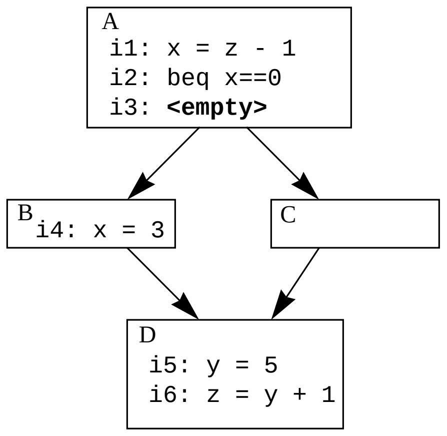
This tradeoff between capability and complexity is evident in the wide variety of published algorithms for global scheduling. By reviewing the tradeoffs made in the existing algorithms for global scheduling, we can better understand the tradeoffs necessary for our target application domain and machine architecture. Through this review, we find that, though some of the ideas are applicable, no single existing algorithm is sufficient to obtain our goal of exploiting the ILP within a non-numerical application without adversely affecting the instruction count of that application.

Bernstein and Rodeh [1991] describe a scheduling algorithm that is targeted toward superscalar processors with a very limited number of machine resources, and thus their algorithm tightly controls the creation of compensation code. This tight control on compensation code results in a fairly limited set of transformations. Bernstein and Rodeh [1991] are interested in the performance of non-numerical applications, and they believe that the conditional branches in non-numerical programs are not predictable. This belief gives them little incentive to search across multiple branches for available instructions. Consequently, their algorithm only looks for available instructions in a dynamically-adjacent basic block or in an *equivalent* basic block [Bernstein and Rodeh 1991]. Two basic blocks are equivalent if and only if the execution of one block implies the execution of the other block (e.g. blocks `A` and `D` of Figure 3-2 are equivalent). Though these decisions result in an algorithm which is very space efficient in its exploitation of ILP, these two decisions restrict the size of the available set, and this restriction can sometimes lead to a missed opportunity for a better schedule. For example in Figure 3-2, their limited set of global transformations results in a definition of availability called *M-ready* instructions [Bernstein et al. 1991]. Under this calculation of availability, instruction `i5` is not available for scheduling in the delay slot of the branch at the end of basic block `A` until instruction `i4` in block `B` is scheduled. Yet, we have already seen that it is possible to schedule instruction `i5` in block `A` by placing a copy of the instruction at the end of the unscheduled block `B`. We definitely want to perform this code motion even though instruction `i5` is executed twice along the path `ABD`. We have not lengthened the execution time of path ABD (the delay slot cycle exists whether the scheduler fills it or not), and we have shortened the execution time of path `ACD`.

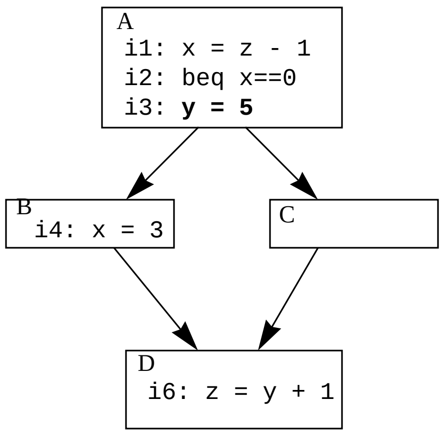
Trace Scheduling [Fisher 1981] was the first attempt at an instruction scheduler with a more global calculation of availability. Ellis [1985] and Colwell et al. [1987] both describe an implementation of Trace Scheduling. Trace Scheduling was originally developed for the compilation of numerical applications, applications in which conditional branches are quite predictable. Because of this predictability, Trace Scheduling relies on execution probabilities to select a *trace* (a major execution path) of basic blocks. From this trace, the algorithm builds a DAG which contains all of the data dependence constraints within the trace. The algorithm then schedules this DAG as if the trace was one large basic block, and the calculation of availability is simply a calculation of data-readiness within the DAG of the trace (the scheduler at this point looks a lot like a basic block scheduler). This calculation of availability is supported by powerful transformations that can even rearrange the order of the conditional branches in the trace and thus change the structure of the CFG. The simplicity of the availability calculation and completeness of the global transformations makes Trace Scheduling quite appealing.

Yet, this same combination of powerful transformations and an intense focus on the trace as a single basic block made the insertion of compensation code a conceptually difficult and compile-time expensive task. In fact, the original implementation of Trace Scheduling by Ellis [1985] did not try to limit the amount of compensation code produced. Any time that the code motion of an instruction moved past a point in the current trace where another trace entered the first trace, Ellis would place a copy of the moving instruction at the end of the entering trace. Figure 3-3 slightly modifies the example of Figure 3-2 so that instruction `i5` is capable of being directly scheduled in the branch delay slot without any duplication off the trace `ACD` (Figure 3-3b). Ellis's implementation of the trace scheduling algorithm automatically inserts a copy of instruction `i5` into block `B` (Figure 3-3c). For numerical applications, this excessive duplication does not noticeably affect performance because numerical applications often contain a single major trace and thus the space and time efficiency of the other traces is not of critical importance. Recently, Gross and Ward [1991] have described some modifications to Trace Scheduling to improve the transformations and optimize the compensation code.

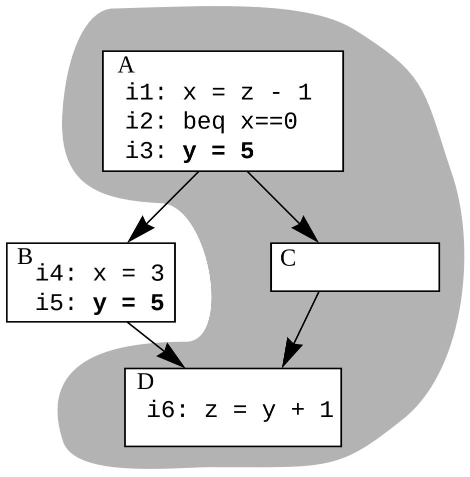
Like the Trace Scheduling compiler, the IMPACT compiler uses traces to obtain a scheduling algorithm with a probability-driven calculation of availability [Chang et al. 1991b]. Moreover, the IMPACT researchers noticed that the use of execution probabilities could improve other parts of the compiler, and thus they also use traces to direct the optimizer [Chang et al. 1991a]. In the IMPACT work, a trace of basic blocks is converted into a *superblock* by code duplication. A superblock indicates the path on which optimization



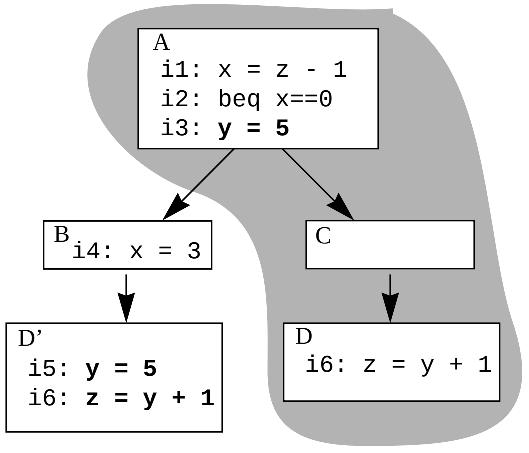
(a) CFG before any global code motion.



(b) CFG after efficient movement of i5.



(c) CFG after Trace Scheduling movement of i5 along trace ACD.



(d) CFG after IMPACT movement of i5 in superblock ACD.

Figure 3-3: Examples of a global code motion.

and scheduling is most important. A superblock is a block of code with a single entry at the top of the block and one or more exits throughout the block. The single entry point ensures that upward code motions in the superblock never require the creation of compensation code. The beauty of this approach is that it is extremely simple to implement since it eliminates the determination of whether duplication is required during the scheduling of a superblock. Yet, the schedules that are not part of the most-probable superblock may be space and time inefficient because all possible code duplications are made before any scheduling takes place. For example, Figure 3-3d shows the duplication of block `D` (by the creation of the superblock `ACD`) results in the unnecessary duplication of instruction `i5`.

Ebcioğlu and Nicolau [1989] discuss an approach to instruction scheduling called Percolation Scheduling with resources (PSr) that is more global than Trace Scheduling in its calculation of available instructions. In PSr, the available instructions (called the *unifiable-ops* [Ebcioğlu and Nicolau 1989]) for a basic block are calculated from all successor blocks on all paths from the current basic block (a path stops when it reaches a back edge in the CFG). That is, PSr computes availability as a dataflow calculation on a CFG with its back edges removed. PSr uses the very powerful Percolation Scheduling rules to determine availability and to transform the code after the scheduling of an operation. Like Trace Scheduling, PSr was developed for VLIW machines with a large number of resources, and it seems that the routines to transform the code after scheduling were developed only with a concern for correctness and not with a concern for the spatial efficiency of the compensation code. Recently though, Moon and Ebcioğlu [1992] published a modified PSr algorithm which is much more concerned with compensation code efficiency. In this modified algorithm, they have disabled the transformation rule which allowed for the reordering of branches, and they have prioritized the instruction selection process so that one execution path is not lengthened to optimize another. This work seems quite promising, and it is obviously heading in a direction similar to the work described in this thesis.

3.2 Issues in our global scheduling algorithm

In this thesis, we are particularly interested in effectively scheduling non-numerical applications for superscalar machines with few parallel resources. We focus on small superscalar machines because (as we show in Chapter 4) we can build them with cycle times that are nearly equivalent to the single-issue version of the same architecture. This type of a resource-limited machine model implies the use of global transformations which are spatially efficient so that the global scheduler does not generate a lot of unnecessary duplication (which will waste the scarce resources and greatly increase the instruction count). On the other hand, the global transformations must be fairly complete and the range of the availability calculation fairly large so that we can obtain our other goal of uncovering a large amount of ILP.

The trace-based approach of Trace Scheduling is quite appealing as the basis for our global scheduler because of the range and simplicity of a trace-based availability calculation. A trace-based approach is applicable because conditional branches are fairly predictable even in a non-numerical applications, and thus traces are a good first approximation of the instructions that we will most-likely execute next (see Chang et al. [1991b], Fisher and Freudenberger [1992], and Table 4-1 on page 83). A trace-based approach is appropriate because the basic blocks in non-numerical applications are small (typically 4 to 5 instructions in length [MDSmith et al. 1989]), and thus a trace-based approach allows the scheduler to look across many of the small blocks in order to find ILP.

Yet, the original Trace Scheduling algorithm is not entirely appropriate for our application domain or our scheduling goals. Remember that Trace Scheduling views a trace of basic blocks as a single basic block during instruction scheduling, and any compensation code created by the scheduling process is only considered during a separate *bookkeeping* phase [Ellis 1985]. As we discussed earlier, this separation makes it difficult to control the spatial efficiency of the algorithm, i.e. it makes it difficult to control the effects of the global code motions on the other traces. This control is important because a global code motion that is slightly beneficial to the current trace might greatly increase the path through another trace or it might produce an excessive amount of compensation code on that other trace. Since non-numerical applications contain many important traces and some 50-50 branches,[3] we should not overly penalize one trace for the benefit of another. As a result of these views, our trace-based scheduling algorithm closely integrates the scheduling and bookkeeping processes. We maintain the concept of individual basic block boundaries in our trace during instruction scheduling so that we can tightly control the compensation code produced and the penalties imposed during the scheduling of a trace. Section 3.3 overviews our trace-scheduling framework, and it discusses the important aspects of the algorithm that tailor it for the non-numerical application domain.

One important way of restricting the penalties on the other traces is to restrict the completeness of the set of global transformations. The major restriction on our global transformations is that the order of the conditional branches is unchanged during scheduling. This restriction ensures that we do not encounter the problem of exponential code explosion during a global code motion. However, one of the nice properties of the original Trace Scheduling algorithm is that the completeness of its global transformations makes the calculation of availability very simple and compile-time efficient, and we would like to maintain this simplicity and efficiency. Section 3.4 discusses how we preserve the simplicity and compile-time efficiency of the availability calculation while still restricting the completeness of the global transformations.

In addition to the major goal of producing effective schedules, we began the development of this global scheduling algorithm with a number of other minor goals in mind. These

3. Notice that this does not conflict with our prior statement that conditional branches in non-numerical applications are mostly predictable. This statement simply says that not all branches are 90-10 (or some other lop-sided ratio) and so we should also tailor the algorithm for the less-probable cases.

minor goals helped shape the structure of the algorithm. Basically, we wanted the algorithm to be compile-time efficient, easily extensible, and highly flexible. By compile-time efficient, we mean that the algorithm should schedule programs in a small fraction of the total compile time. This goal is achieved through a number of efficient, non-backtracking techniques. Scheduling decisions, once made, are never undone, and dataflow information is incrementally updated whenever possible. By easily extensible, we mean that the algorithm should interact well with ILP-increasing optimizations (such as procedure inlining [Chang and Hwu 1991]) and loop-level parallelization techniques (such as loop unrolling and software pipelining [Lam 1990]). This goal is achieved through the scheduling of loops as independent entities. In this way, the global scheduler works whether an inner-loop is scheduled through simple global scheduling techniques or through more advanced loop scheduling techniques. By highly flexible, we mean that the algorithm should efficiently schedule code for a wide range of machine models. For example, the global scheduler should intelligently handle instructions with non-unit latencies, instructions with delay slots, non-pipelined functional units, machine models with a fairly-wide range of functional unit distributions, and other non-trivial hardware constraints. This goal provides a vehicle with which it is possible to experiment with the tradeoffs between machine complexity and achieved performance.

3.3 A trace-scheduling framework

Figure 3-4 contains an outline of our global scheduling algorithm. At the outermost level, the algorithm schedules one procedure at a time. Within each procedure, scheduling proceeds from innermost to outermost loops, and each loop is scheduled as an integral entity. The parts of the procedure body which are not part of any loop are scheduled last. Because of this scheduling order, global code motions only occur from inner loops into outer loops. Within each loop, traces are selected and scheduled until no unscheduled basic blocks exist within a loop. At this point, our scheduler collapses the loop into a structure that looks like a single basic block, and it summarizes the dataflow information for this loop so that code motions can occur around this inner-loop. After the entire procedure is scheduled, our algorithm runs a compaction routine to remove as many static NOPs as possible (compaction is discussed in Section B.1 of Appendix B).

Our global scheduler builds the CFG in a single pass when the procedure is read in.[4] Each node in the CFG is a basic block, except that call instructions do not necessarily terminate a basic block. With interprocedural information, a call instruction looks like a simple ALU

```
foreach PROCEDURE {
   generate CFG;
   compute initial GLOBAL DATAFLOW INFO;
   find and order LOOPs;
   foreach REGION (innermost loop out to procedure level) {
      while (exists unscheduled BASIC BLOCK in REGION) {
         select next best TRACE;
         schedule TRACE;
      }
      collapse REGION;
   }
   compact PROCEDURE;
}
```

Figure 3-4: Overview of the trace-scheduling framework.

instruction which has many sources, destinations, and side-effects. The scheduler, and not the CFG builder, decides whether code motions can occur past a call instruction. In the CFG, a special node is designated as the ENTRY node of the CFG. The ENTRY node has no preceding basic blocks, and the first basic block in the procedure is the only successor of ENTRY. Another special node is designated as the EXIT node. The EXIT node has no succeeding basic blocks, and the predecessor blocks of the EXIT node are all the basic blocks which end in a procedure return. A basic block *d* in the CFG is said to *dominate* a basic block *n* if *d* is executed on every path from the ENTRY node to block *n* [Aho et al. 1986]. Domination is an example of global dataflow information that is required for scheduling. We present other pieces of global dataflow information as they are needed for the description of our algorithm.

In order to process the loops in a depth-first ordering, our global scheduling algorithm assumes that all loops in the application are *natural loops* [Aho et al. 1986], loops with a single entry point. This restriction is not an inherent property of our trace-based scheduling algorithm but a consequence of the organization of the surrounding framework. In the following discussion, a *loop edge* or *back edge* is any edge in the CFG where the basic block at the end of the edge dominates the basic block at the start of the edge. A *loop head* is the basic block at the end of a loop edges.

4. The scheduler expects the front-end of the compiler system to annotate each dynamic jump instruction (e.g. a jump-through-register instruction or a C-style switch statement) with all its possible successor blocks. If any dynamic jump instruction is not annotated, the scheduler builds an incomplete CFG. Since a global code motion on an incomplete CFG might unknowingly result in a violation of the program semantics, an incomplete CFG causes the scheduler to default to scheduling without global code motions.

Instead of discussing our entire trace-scheduling framework in detail, the rest of this section briefly describes our algorithms for building and scheduling a trace. Along the way though, we highlight those aspects of our trace-based scheduling algorithm which are significantly different from the previous trace schedulers. These differences translate into new capabilities (e.g. scheduling from already-scheduled basic blocks) and into special priority heuristics for the basic block scheduler.

3.3.1 Building and scheduling a trace

A trace is a sequence of basic blocks which are executed in order for some choice of input data. Execution profile information drives the selection of traces so that a trace represents the most probable sequence of basic blocks. Our global scheduler uses branch profile information or branch prediction heuristics to determine the most-likely successor to a basic block ending in a conditional branch. In the original Trace Scheduling algorithm [Fisher 1981], the traces are loop-free sequences of unscheduled basic blocks. In this global scheduling algorithm, a trace may contain a loop, and it often terminates with an already-scheduled basic block. Both of these capabilities are included to mitigate the usual lack of scheduling lookahead associated with the end of a trace.

The global scheduler chooses traces by searching through the loop or procedure body in the program order looking for the first unscheduled basic block. Once the scheduler has located the next unscheduled basic block, it grows a trace forward (in the direction of the edges in the CFG) with this basic block as the start of the trace. This approach is slightly different from the one taken by Ellis [1985]. In his compiler, the operation with the highest probability of execution is chosen as the *seed* of the trace, and the trace is grown both forward and backward from this seed. Figure 3-5 outlines our algorithm for building and scheduling a trace given a seed basic block.

During the construction of the trace, two data structures are built. The global scheduler uses these structures to determine scheduling priority and instruction availability. The first structure is called the *EDAG* (for *Extended DAG*), and it is a simple data dependence graph of all the instructions in the trace. The other data structure is called the *GCL* (for *Global Constraint List*), and it captures the control dependence and off-trace data dependence information needed for correct global code motion. The EDAG gives us a global view of instructions in the trace (it is our instruction window), and it encodes the data constraints along the trace. The GCL encodes the global constraints on the movement of instructions along the trace. We discuss the specifics of the EDAG in the next subsection, and we discuss the specifics of the GCL in Section 3.4 on availability and bookkeeping.

```
build_and_schedule_trace(BB *sBB /* “seed” of trace */) {
   trace and EDAG initially empty;
   nBB = sBB;
   while (nBB) {     /* build */
      add nBB to trace;
      insert instructions from nBB into EDAG;
      choose next nBB (if any);
      if (nBB) compute new global constraints for GCL;
   }
   prioritize EDAG;
   nBB = sBB;
   while (nBB) {     /* schedule */
      cBB = nBB;
      schedule cBB;
      update trace, EDAG, and GCL;
      special case operations for special case scheduling;
      nBB = next basic block in trace (if any);
   }
}
```

Figure 3-5: Algorithm for building and scheduling a trace.

Once the global scheduling algorithm has built the trace and its dependence structures, the next step is to prioritize the instructions in the EDAG and thus prepare for the scheduling of the trace. Priorities help the scheduler choose among the data-ready instructions in the EDAG. As we mentioned in the background section on basic block schedulers, a large number of priority heuristics exist, and the implementation of our global scheduling algorithm is set up so that one can easily change the priority function. Currently, the global scheduler prioritizes the instructions in the EDAG by *height* (or *maximum path length to a leaf*) for the top-down scheduler, and by *depth* (or *maximum path length from root*) for the bottom-up scheduler. Height and depth are complementary priority calculations, and they both attempt to balance the progress of the code scheduler [Smotherman et al. 1991]. The complementary values are necessary because our basic block scheduler uses a combination of top-down and bottom-up instruction scheduling (see Appendix A).

The trace is scheduled by individually scheduling each basic block in the trace in the order that they appear in the program. Each basic block is scheduled using a list scheduling algorithm (described in detail in Appendix A). In addition to the priorities assigned to the instructions in the EDAG, our list scheduling algorithm gives priority to the *native* instructions (those instructions that originally lived in the current basic block) over the non-native instructions. Furthermore, the basic block scheduling algorithm does not allow the native instructions to move down and out of the current basic block, and it tries to fill in the empty instruction slots in the current basic block schedule with instructions from basic blocks later in the trace. In this way, a basic block schedule is never lengthened by a global code motion (i.e. global code motions only occur to fill empty instruction slots), and therefore the other traces are never lengthened by a global code motion on the current trace. Because of this priority scheme, the global scheduling algorithm in this thesis may not produce as good a schedule as Fisher's Trace Scheduling algorithm would produce for a very probable trace. Yet, for the non-numerical application domain where there is no single major trace, we feel that our priority scheme is more appropriate because it produces reasonably good schedules for all of the traces.

Basically, the list scheduling algorithm uses the EDAG to quickly find and select among the data-ready instructions in the trace. However, a data-ready instruction from a later basic block in the trace is not guaranteed to be available (recall that availability is a function of the global transformations as previously illustrated by Figure 3-2), and thus the scheduler must first check the availability of the data-ready instruction before attempting to schedule it. This optimistic outlook on instruction readiness makes the design of the global scheduling algorithm simple; yet it opens up the potential for a compile-time inefficient scheduling algorithm. For example, if most of the high-priority, data-ready instructions are not available, then the basic block scheduler could spend much of its time dealing with a ready list in which most instructions are not actually ready. Our global scheduling algorithm avoids this problem through a combination of techniques. First of all, we give priority to the native instructions (which are always available) over the non-native instructions. Secondly, we precompute and summarize the availability of each instruction so that the scheduler does not consider an instruction for scheduling until it becomes available. Section 3.4 discusses our bookkeeping data structures and how we can precompute availability in a compile-time efficient manner.

After the basic block scheduler has completely scheduled a basic block, the global scheduler updates the trace and its constraints so that scheduling can proceed on the next block. There are a number of special case situations which occur during the scheduling of the trace, and these situations are handled at this point in the scheduling process. An example of a special case situation is when the next basic block to be scheduled is an already-scheduled basic block. The block is not scheduled again, but the trace must reflect the fact that instructions were taken from this already-scheduled basic block. Before we explain how we accomplish this and the other special capabilities of our trace-based approach, we need to describe how we construct the EDAG data structure and how we choose the basic blocks in the trace.

3.3.2 Building the EDAG

Whenever the scheduler adds a basic block to the trace, the scheduler adds the instructions within that basic block to the EDAG. These instructions are added in program order so that the correct data dependence relationship is maintained between instructions. As the scheduler inserts an instruction (called the *current* instruction), it adds a number of different dependence edges between this instruction and instructions already in the EDAG. For each source register in the current instruction, a true dependence edge is added from the instruction which last defined this register (if any) to the current instruction. Similarly, an output dependence edge is added from the instruction which last defined the current instruction's destination register (if any) to the current instruction. To simplify the determination of the latest definition of a register, the scheduler maintains a list of the instructions which last defined each of the registers in the machine. In order to determine anti-dependence constraints, the scheduler maintains another list of instructions which correspond to all of the uses of a register since the last definition of that register. Thus, given the current instruction's destination register, the scheduler places an anti-dependence edge from each of the last-use instructions for this register to the current instruction. As explained in Section 3.1.1, a true dependence edge is labelled with the latency of the instruction at the start of the edge, an output dependence edge is labelled with a latency of the maximum of 1 and (l_1 - l_2 + 1), and an anti-dependence edge is labelled with a latency of zero cycles. The global scheduler takes special care to ensure that implicit condition code registers, special registers, double registers, and other novel register structures are handled correctly by the data dependence analyzer.

If the current instruction is a memory operation, the scheduler checks to see if any additional constraint edges are required in order to maintain the dependence relationship between memory locations. Basically, the address of the current memory operation is checked against the addresses of all the previous memory operations already in the EDAG. A load address is checked against all previous store addresses to locate true dependences. A store address is checked against all previous load addresses to locate anti-dependences and against all previous store addresses to locate output dependences. A dependence edge is never placed between two load operations, and it is assumed that loads can always be reordered with respect to each other. For memory-mapped I/O, the global scheduler takes special care to ensure that non-idempotent load operations are never reordered, and that these I/O operations are not duplicated on a single execution path.

The scheduler labels the true and output memory dependence edges with their appropriate delay value. For the simple load/store architectures discussed in this thesis, both these values are equal to 1 cycle. The delay value associated with a memory anti-dependence edge is slightly trickier. Even with a simple load/store architecture, one cannot immediately assume that a concurrent load and store of the same memory location is handled correctly. Though there are pipelined memory architectures where this operation is possible, the dependence analyzer is conservative and labels anti-dependence memory edges with a delay of 1 cycle.[5]

The static determination of whether two memory operations address the same memory location is called *alias analysis*, and our scheduler supports three user-specified levels of alias analysis. The first and simplest level performs no analysis of the memory addresses. The only reordering allowed is between load instructions; all other orderings are kept in the original ordering. The second level of analysis is called simple analysis. Simple analysis differentiates between references off the stack pointer, references off the global pointer, and references off an arbitrary pointer. Simple analysis returns one of three answers to the question of whether two memory operations are aliases of each other: they are aliases; they are not aliases; or the analysis failed. For the case where analysis fails, the scheduler conservatively assumes that the memory addresses do alias. Figure 3-6 outlines our algorithm for simple analysis. The final level of analysis leverages off analysis done in earlier passes of the compiler. Basically, the compiler labels independent load and store instructions with unique identifiers that the alias analyzer can compare. For this final level, the scheduler actually performs simple analysis first, and if this fails, the scheduler then looks for the labels. The scheduler assumes that two memory addresses do not alias only if both have labels and the labels are different and unique.

The final type of constraint edge that the scheduler can add to the EDAG maintains the original program ordering for control transfer instructions (CTIs). Conditional branches, unconditional jumps, procedure calls, and procedure returns are all examples of CTIs. Upon seeing a CTI, the scheduler adds an edge from the last CTI to this current CTI, and the scheduler labels this edge with a delay of 0 cycles. These constraint edges represent a capability, and not a data dependence, constraint. These constraint edges enforce the

5. Notice that the *register-to-register* anti-dependence delay is still 0 cycles.

```
given two memory addresses of the form ‘base + offset’;
ans = failure;
if ($base_i$ == stack_ptr) {
   if ($base_j$ == stack_ptr) {
      if (offsets equal or overlap) ans = alias;
      else ans = NOTalias;
   } else if ($base_j$ == global_ptr) ans = NOTalias;
} else if ($base_i$ == global_ptr) {
   if ($base_j$ == stack_ptr) ans = NOTalias;
   else if ($base_j$ == global_ptr) {
      if (offsets equal or overlap) ans = alias;
      else ans = NOTalias;
   }
} else if ($base_i$ == $base_j$) {
   if (offsets equal or overlap) ans = alias;
   else ans = NOTalias;
}
return ans;
```

Figure 3-6: Algorithm for simple alias analysis.

previously-discussed policy of limiting the amount of code explosion possible during instruction scheduling.

Another important capability constraint deals specifically with the call instruction. To correctly allow for code motion past a call instruction, the scheduler requires interprocedural data dependence information. This information tells the scheduler which instructions are possibly dependent upon the call instruction. Without this information, the scheduler must conservatively assume that all instructions after the call are dependent upon the call. The scheduler has two choices in implementing this conservative approach. Either continue to build an EDAG after seeing a call instruction and add dependence edges from the call to all subsequent instructions, or simply stop building the EDAG once a call is inserted. Our scheduler follows the second approach. Calls cause the scheduler to temporarily suspend the building of a trace until the section of the trace above the call is scheduled. Once the scheduling of that part of the trace is complete, the trace continues from the point of the call. In other words, the main loop of the global scheduler (see Figure 3-4) invokes the procedure in Figure 3-5 with the remaining part of the basic block (starting after the trace-suspending call instruction).

3.3.3 Choosing the next basic block

Once the scheduler has chosen a seed basic block and built an EDAG from its instructions, the scheduler proceeds to choose a successor block of the current basic block and adds this new block to the trace. The action of choosing the next successor basic block for the current trace is referred to as “growing the trace”. Basically, a trace is grown from the seed basic block using branch probabilities (either from a execution profile or from heuristics) until one of four conditions is met: the next block is not in the current loop (e.g. a call); the next basic block is dynamically determined (e.g. an indirect jump); the next basic block is already scheduled; or the next basic block is already in the current trace (e.g. a loop edge). For the last two conditions, the trace is extended one more basic block to mitigate the usual lack of scheduling lookahead associated with the end of a trace. The scheduler’s actions under these last two conditions are described in detail in the next two subsections. The algorithm for choosing the next basic block in the trace is listed in Figure 3-7.

```
cBB = the currently-last basic block (BB) in the trace;
if (saw call || saw return || saw indirect_jump) nBB = NULL;
else if (saw branch) {
   if (branch profile predicted) {
      if (loop_branch)
         nBB = target BB of branch;   /* force prediction */
      else {
         if (predict take) nBB = target BB of branch;
         else nBB = lexical successor BB of branch;
      }
   } else {    /* heuristic prediction */
      if (loop branch) nBB = target BB of branch;
      else nBB = lexical successor BB of branch;
   }
} else {       /* saw fall-thru or unconditional jump */
   nBB = only successor of cBB;
   if (nBB is out of region) nBB = NULL;
   if (nBB is already scheduled || nBB already in trace) {
      specially add nBB to trace;
      nBB = NULL;
   }
}
return nBB;    /* next basic block in trace */
```

Figure 3-7: Algorithm for choosing the next basic block in the trace.

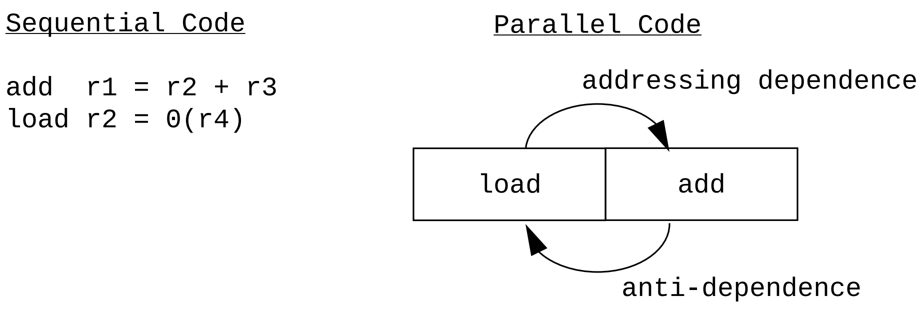
3.3.3.1 Scheduling from already-scheduled basic blocks

In the original Trace Scheduling algorithm, a trace ends if the next predicted basic block is part of an already-scheduled trace [Fisher 1981]. A consequence of this approach is that the global scheduler has fewer and fewer available instructions from which to choose as it approaches the end of the trace. Since the scheduler produces better schedules with a larger set of available instructions, we would like to maintain the size of the availability set even as the scheduler approaches the end of the trace. One way to achieve this is to pull instructions from the already-scheduled basic blocks. Yet, we must carefully constrain this capability so that we do not have to re-schedule the already-scheduled basic blocks (remember that one of our goals was to implement a global scheduler that never back-tracks).

Our global scheduler is able to pull instructions from a single, already-scheduled basic block whenever the basic block preceding the already-scheduled basic block ends in a jump or a taken branch. The lookahead of one block is often the maximum lookahead possible for our global scheduler because our scheduler does not rearrange CTIs and the already-scheduled basic blocks frequently end in a CTI. Basically, the global scheduler pulls copies of the instructions from the already-scheduled basic block in order of increasing instruction address. To restore program correctness, the scheduler then changes the target offset of the branch or jump instruction (in the last unscheduled basic block in the trace) by the number of copies pulled from the already-scheduled basic block and inserted into the current trace schedule. To ensure that the basic block scheduler only considers the copies of the already-scheduled instructions in order, the global scheduler places an *addressing* dependence edge from the last copy to the current copy as it inserts each copy into the EDAG. The addressing constraint ensures that a copy from later in the already-scheduled basic block is only considered for scheduling in the new trace if all preceding copies with lower addresses have been pulled and scheduled. The addressing dependence edge is given a latency of zero cycles so that it acts like a register anti-dependence edge.

As an aside, it is possible to make this mechanism even more sophisticated and therefore more applicable. The current restriction is that the last basic block in the trace must end in a jump or branch that is predicted to take so that we can change the target offset after the pulling of instructions. If the last basic block simply falls into the already-scheduled block, we could support scheduling across this edge if we simply added an unconditional jump onto the end of first block. For a branch which is predicted to not take, we could invert the branch's condition and the program layout of the THEN and ELSE blocks so that the branch is now predicted to take.

For asymmetric microarchitectures, our zero labelling of anti-dependence edges interferes with this approach to the scheduling of already-scheduled basic blocks. An asymmetric microarchitecture is one in which some instructions are not capable of issuing from any location in memory; VLIW architectures often employ an asymmetric microarchitecture. For example, assume that we are given a two-issue superscalar processor which can only issue a memory operation and an integer operation in parallel if the memory operation is the first instruction in the instruction fetch unit (i.e. the memory port is only connected to the integer pipeline on the “left” side of the machine). A legal execution unit for this processor could place a memory load instruction at a lower address than an anti-dependent ALU instruction, if the scheduler knew that the instructions would be fetched and executed together. The anti-dependence constraint in this schedule runs in the opposite direction from the addressing constraint (see Figure 3-8), and these two constraints cause a dependence loop in the EDAG. Under this condition, the EDAG is no longer an acyclic graph, and our basic block scheduling algorithm will fail. For this reason, the routine which inserts the copies of the instructions from an already-scheduled basic block detects these backward-flowing anti-dependence edges, and the routine halts the construction of the EDAG at this point. This action is conservative. If the scheduler knew (a priori) that the anti-dependent pair and all instructions scheduled between the pair would be scheduled in the new trace or if it could undo the scheduling of a basic block to recreate a single-entry point into the already-scheduled basic block, then the scheduler could build the EDAG without the addressing dependence edges and thus without the possibility of a dependence loop. Since our global scheduler does not know the future and since it does not backtrack, it simply halts the construction of the EDAG for any packet with a backward-flowing, anti-dependence edge.



```
add  r1 = r2 + r3
load r2 = 0(r4)
```

Figure 3-8: Example of a backward-flowing anti-dependence edge.

Another method of solving this problem is to label all of the anti-dependent edges with a delay of 1 cycle (i.e. no data dependences are possible within a packet of instructions). We chose to maintain the 0-cycle delay on the register anti-dependence edges because we found through experimentation that an anti-dependence delay of 0 cycles is more important to the performance of a non-numerical application than an unrestricted ability to schedule from already-scheduled basic blocks (using an anti-dependence delay of 1 cycle). The exact amount that our limited ability to schedule from already-scheduled basic blocks increases performance is dependent upon the particular application and upon the specifics of the hardware model, but even for a relatively simple superscalar processor[6], this ability improves the superscalar cycle counts of our benchmarks by 0–3%.

3.3.3.2 Dynamic completion of the EDAG

The scheduling of already-scheduled basic blocks tries to provide our global scheduler with a generic method for mitigating the end-of-trace penalty. Yet, there is one important loop-edge situation that our technique of scheduling from already-scheduled basic blocks does not cover. The situation occurs in a trace where the trace begins with the loop head and continues until it reaches a loop edge. At this point, the trace builder will attempt to re-add the loop head at the end of the trace. The instructions in the loop head are already part of the EDAG, so the scheduler can only add copies of these instructions to the EDAG. However, the copies require an addressing constraint (so the scheduler can ultimately change the offset of the loop branch), and this addressing constraint is dependent upon the scheduling order which has not been determined. The global scheduler is now left with a “chicken and egg” problem. The scheduler cannot complete the trace until the loop head is scheduled, and the scheduler cannot schedule the loop head until the trace is completed.

To provide the global scheduler with the biggest window for finding ILP, we want to construct as much of the EDAG as possible before we start any instruction scheduling. For the case where the loop head occurs twice in the trace, this means that the global scheduling algorithm should build all of the EDAG up to the point where the loop head is re-encountered. At this point, the global scheduler begins the scheduling of the EDAG, but as each cycle in the loop head is scheduled, the instructions in this cycle are inserted into the EDAG using the already-scheduled basic block techniques. Once the entire loop head has been scheduled (or a backward-flowing anti-dependence constraint is seen), the global scheduler considers the EDAG complete and scheduling continues in a normal fashion. We refer to this as the dynamic completion of the EDAG because the global scheduler begins the scheduling of the EDAG before the EDAG is complete.

The most difficult aspect of dynamically completing the EDAG is the dynamic update of the priority information kept in the EDAG. Unless the EDAG is dynamically completed, the global scheduler defines the priorities once and they remain constant during the scheduling of the trace. When instructions are dynamically inserted into the EDAG, the global

6. The machine model is a two-issue TORCH machine with limited functional units and hardware support for one level of boosting (see Chapter 4 for more details). During this experiment, we found that the compress, eqntott, and espresso benchmarks showed the greatest decrease in cycle counts, while the grep benchmark was at the other end of the spectrum with no change in cycle count.

scheduler incrementally updates the priorities to indicate the new state of the EDAG. Specifically, we only update the priorities of those instructions that are affected by the newly-inserted instructions. Since we only dynamically insert instructions at the end of the EDAG, we only need to recalculate the heights of the instructions which are data dependent upon the newly-inserted instructions (the depths of the instructions in the EDAG are unaffected). We accomplish the update through an upward scan of the EDAG starting with the newly-inserted instructions. This incremental update is efficient because the newly-inserted instructions often affect only a small subset of the nodes in the EDAG.

By dynamically completing the EDAG (without the ability to schedule from already-scheduled basic blocks), our global scheduler generated schedules that were 1–9% faster on a relatively simple superscalar machine.[7] By both dynamically completing the EDAG and scheduling from already-scheduled basic blocks, our global scheduler generated schedules that were 1–10% faster.[8] Close examination of the results shows that these two techniques are orthogonal in their impact on performance.

3.4 Availability and bookkeeping

To take advantage of the ILP that exists across the basic block boundaries, the global instruction scheduler relies on a set of global transformations which govern the movement of instructions across the edges of the CFG. These transformations are used to define the set of available instructions, and they used to determine where compensation code is required for program correctness. Section 3.4.1 describes the set of global transformations used in our global scheduling algorithm. Section 3.4.2 then discusses how we incorporate these transformations into our global instruction scheduling algorithm to produce the bookkeeping data structures. These structures simplify the determination of instruction availability and the calculation of compensation code.

3.4.1 Transformations to support upward code motion

Our global transformations only support the upward movement, movement against the direction of the edges in the CFG, of instructions. Since our trace-scheduling framework schedules the basic blocks in each trace in their order of execution and since it schedules

7. This experiment uses a machine model that is identical to the one used in the last subsection. During this experiment, we found that the awk, espresso, and grep benchmarks showed the greatest decrease in cycle counts, while the nroff benchmark was at the other end of the spectrum with little change in cycle count.
8. Again, the same machine description as before was used.

all instructions native to a basic block in or before that basic block, our global scheduler never attempts to move an instruction down and out of a basic block.

To provide the global scheduler with as much freedom during scheduling as possible, the repeated application of our global transformations can achieve the upward code motion of any non-CTI across multiple basic block boundaries. To limit the potential for code explosion during global scheduling, our global transformations do not support the reordering of CTIs. Though we constrain the upward movement of a CTI, we still allow for the speculative execution of a CTI in the delay slot of the preceding CTI. In this way, our global transformations try to balance the need for general code motions with the desire to avoid excessive code explosion.

A general algorithm for upward code motion using our global transformations follows three basic rules:

(1) a rule for intra-block motion,

(2) a rule for motion out of the top of a block, and

(3) a rule for motion into the bottom of a block.

The rule for intra-block motion simply involves the movement of an instruction over earlier instructions in the basic block. This motion is inhibited by any earlier instructions which impose data dependences upon the instruction being moved (the *current* instruction). For instance, an instruction cannot move above the definition of its operands. If the current instruction is not free to flow up to the top of the basic block, the upward code motion routine recursively applies itself to all of the preceding, data-dependent instructions of the current instruction.

Once the current instruction is at the top of the block, it is free to move to the bottom of the preceding blocks. A copy of the instruction is placed at the end of each preceding basic block so that the instruction still executes on every path that reaches the current basic block. Thus, the motion of an instruction out of the top of a block can require *duplication*.

On the other hand, the motion of an instruction into the bottom of a preceding block can require *hardware-assisted speculative execution*. If the only successor of a preceding basic block is the original basic block, then the current instruction is always *useful* because the instruction is always executed independent of the control flow. If instead the preceding block has multiple successors, then the current instruction is a speculative operation which is only useful if the execution proceeds down the CFG edge that the current instruction just traversed. Hardware-assisted speculative execution (such as boosting) ensures that this speculative instruction does not corrupt the program state if the control flow proceeds down a different CFG edge. With hardware-assisted speculative execution, we are at a point where we can re-apply the first rule. Through the successive application of the three basic rules, we can continue to move the current instruction up through the CFG. These three rules are sufficient because they cover all possible entry and exit configurations for a basic block in a CFG.

Without hardware-assisted speculative execution, this generic upward code motion algorithm is limited by any global code motion that results in unsafe or illegal speculative execution. To adapt this algorithm for environments without hardware-assisted speculative execution, we have the algorithm perform the following checks before moving an instruction above a conditional branch. If any of these checks are true, the algorithm halts the progress of the upward code motion. The algorithm checks for an unsafe speculative execution by checking if the current instruction is capable of signalling an exception. If it can cause an exception, we cannot speculatively execute this instruction in a safe manner. The algorithm checks for an illegal speculative execution by checking the set of values that are needed when the other edge or edges of a preceding block are traversed. We can obtain this set of values through live-variable analysis (see Aho et al. [1986], pp. 631–632). By checking this set of values against the destination register of the current instruction, the algorithm can determine when a movement would result in illegal speculative execution.[9] As Section 2.1.1 discussed, the compiler can rely on software register renaming techniques to transform some of these movements into legal speculative execution, and our algorithm can check for these cases.

These simple rules succinctly describe our core global transformations—the set of transformations that are sufficient to support the upward code motion of any non-CTI. Yet, this set of transformations is pessimistic because it assumes that every instruction that moves above a conditional branch is dependent upon the conditional branch. As Lam and Wilson [1992] quantitatively show, precise knowledge of the control dependence during global scheduling can noticeably improve the instruction schedule and thus machine performance. To capture some of this benefit, our algorithm for upward code motion includes one more transformation. This new transformation works on equivalent basic blocks [Bernstein and Rodeh 1991]. This transformation determines when a global code motion

9. Any illegal speculative executions due to the overwriting of a value in memory (i.e. a speculative store instruction) are caught by the unsafe speculation check since all store operations can cause an exception.

across a conditional branch does not require speculative execution or off-trace duplication because the moving operation is independent of the execution of the conditional branch.

To explain the transformation, we need to explain some terminology. Two basic blocks are *control equivalent* if one block of the pair *dominates* the execution of the other block, and the other block *post-dominates* the execution of the first block. Recall that a basic block *d* dominates a basic block *n* if *d* is executed on every path from the ENTRY node to block *n* [Aho et al. 1986]. Similarly, a basic block *p* post-dominates a basic block *n* if *p* is executed on every path from block *n* to the EXIT node [Aho et al. 1986]. As an example, blocks `A` and `D` of Figure 3-3a on page 43 are control equivalent because the execution of one implies the execution of the other. Two control-equivalent blocks are *data equivalent with respect to the moving instruction* if the moving instruction is free of data dependences with any instruction along any path between the control-equivalent blocks. If two blocks exist that are both control equivalent and data equivalent with respect to a moving instruction, our new transformation will simply move the instruction between the two blocks without any duplication or checks for illegal or unsafe speculative execution. The movement of instruction `i5` from block `D` to block `A` in Figure 3-3b is an example of this new transformation. Control equivalence guarantees us that the execution of instruction `i5` is always useful in block `A`, and data equivalence guarantees us that there are no data dependences violated along the paths between blocks `A` and `D`. It is interesting to note that our core set of transformations would produce the CFG in Figure 3-3c when moving instruction `i5` from block `D` to block `A`. Thus, this new transformation not only improves the execution time of the scheduled code, but it also improves the spatial efficiency of our set of global transformations.

Figure 3-9 outlines an algorithm for upward code motion that is based on our global transformations. This algorithm does not assume the existence of any hardware to assist in the movement of speculative operations; obviously the checks for illegal and unsafe speculative execution are unnecessary if we have hardware-assisted speculative execution.

3.4.2 Bookkeeping

Through repeated application of the algorithm in Figure 3-9, our global scheduler can move an instruction across multiple basic blocks. Since instruction availability depends upon finding a set of global transformations that result in a data-ready instance of the instruction, we could invoke the repeated use of this algorithm every time that our global scheduler inquired about the availability of an instruction. Yet, a shortcoming of this

```
given an instruction I in block A to move;
given a path from block B to block A;
while (A != B) {
   move I to top of A;
   if (control/data equivalent pair to A exists on path) {
      move I to bottom of pair;
      A = equivalent pair on path;
   } else {
      /* check for illegal and/or unsafe speculation */
      if (code motion into any predecessor of A results
          in unsafe speculative execution) stop;
      if (code motion into any predecessor of A results
          in illegal speculative execution) {
         rename destination of I;
         insert copy at original position of I;
      }
      foreach (predecessor C of A) {
         duplicate I at end of C;
      }
      remove I from A;
      A = predecessor of A on path;
   }
}
```

Figure 3-9: Algorithm for upward code motion.

straightforward approach for determining availability is that the scheduler performs a great deal of redundant analysis. As we will see, the global movement of an instruction along the trace of basic blocks not only determines that instruction's availability for the current basic block, but it also determines the earliest point along the trace when that instruction is available. Since a single global movement can determine earliest availability, it is wasteful to recalculate availability at every availability inquiry. Instead, our approach to global scheduling precalculates and summarizes the availability of each instruction so that an availability inquiry becomes a compile-time efficient operation. Our global scheduling algorithm also summarizes the global dataflow constraints along the trace so that the global movement of an instruction is a compile-time efficient operation. In this subsection then, we describe the organization and the handling of the bookkeeping information for instruction availability and for our global transformations.

3.4.2.1 Support for duplication and speculation

To summarize the constraints on the global movement of an instruction, we need to first answer the question of what causes an instruction to be unavailable for scheduling. One reason why an instruction might not be available for scheduling is that the instruction is not data-ready along the trace. Figure 3-10 contains an unscheduled CFG where the global scheduler has chosen the trace `ABDE`. If no instructions have been scheduled for this trace, then instruction `i4` is an example of an instruction that is not data-ready along the trace since it depends upon the result of the on-trace instruction `i2`. The EDAG in our trace-scheduling framework nicely summarizes the on-trace data dependences of an instruction so there is no need for a new data structure to re-summarize this information.

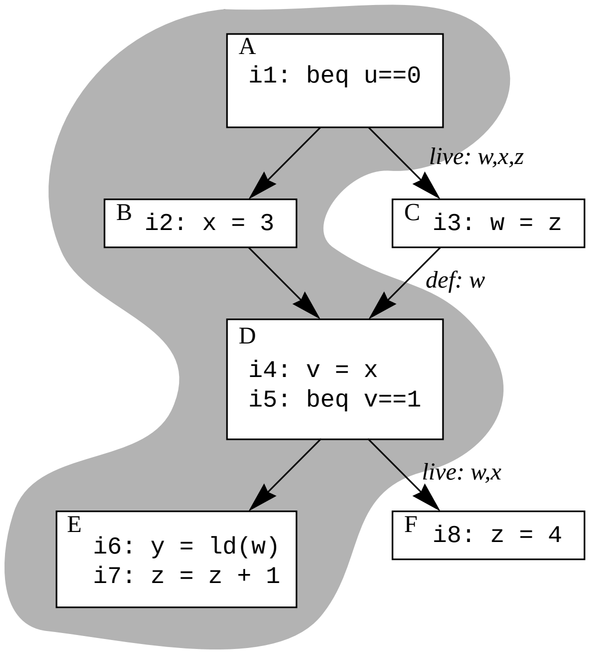


Figure 3-10: Example for availability constraints.

A second reason why an instruction might not be available for scheduling is that the movement of the instruction results in an unsafe or illegal speculative execution. This constraint on availability is a property of the hardware and compiler support for speculative execution. Without support for hardware-assisted speculative execution, the global movement of instruction `i6` out of basic block `E` in Figure 3-10 causes an unsafe speculative execution (because the load can cause an exception). As another example, the global movement of instruction `i7` from block `E` to block `A` results in illegal speculative execution (because the register `z` is live on entry to block `C`). Thus, both instructions `i6` and `i7` are unavailable for scheduling in block `A`. Once we schedule block `A` though, instruction `i7` becomes available for scheduling; on the other hand, instruction `i6` is only available for scheduling in block `E`. With some hardware assistance for speculative execution, both instructions `i6` and `i7` could become available in block `A`.

A third reason why an instruction might not be available for scheduling is that the movement of the instruction requires a duplication of the moving instruction that the global scheduler cannot satisfy. This constraint on availability is solely a property of the sophistication of the routine that actually performs the global code motions and creates the compensation code. To clearly explain our duplication constraints, we need to present a few more definitions. A *join block* is a basic block with more than one predecessor block, and a *branch-ending block* is a basic block with two successor blocks.[10] The *on-trace predecessor (successor)* is the predecessor (successor) block of the join (branch-ending) block that is on the trace. The *off-trace predecessors (successor)* are (is) the predecessor blocks (successor block) of the join (branch-ending) block that are (is) not on the trace.

Duplication can occur whenever an instruction is moved up and out of a join block. To perform the duplication, we would like to simply place a copy of the moving instruction at the end of each predecessor block. We refer to this action as *duplicate and remove (DaR)*, and it is illustrated in Figure 3-11a. This action is possible as long as the predecessor block is not already scheduled or as long as the predecessor block does not end in a conditional branch. To safely place the duplicate in a predecessor block that is already scheduled, we would have to reschedule that predecessor block. Since our scheduling algorithm does not backtrack, we do not allow duplications into already-scheduled basic blocks. To safely place the duplicate in a predecessor block that ends in a conditional branch, we would have to make sure that the duplicate does not destroy the operands of the branch and that the speculative execution of this duplicate is safe and legal. Since it is difficult and expensive to perform this analysis in our implementation of the global scheduling algorithm and since our global scheduler will schedule for hardware implementations without hardware-assisted speculative execution (otherwise we could always label the instruction as a speculative instruction), we do not allow duplications into predecessor blocks that end in conditional branches.

Still, we can get around both of these restrictions through the insertion of a new basic block between the join block and the offending predecessor block. Since the straightforward insertion of a new basic block requires the scheduler to change the branch target of the off-trace predecessor block and it requires the scheduler to add an unconditional jump to the end of the new basic block (deemed too expensive for our application domain), our algorithm actually creates the new basic block by splitting the join block into two blocks.

10. Our global scheduler stops a trace at a dynamic CTI, and thus we never need to consider a global code motion into a block with more than two successor blocks

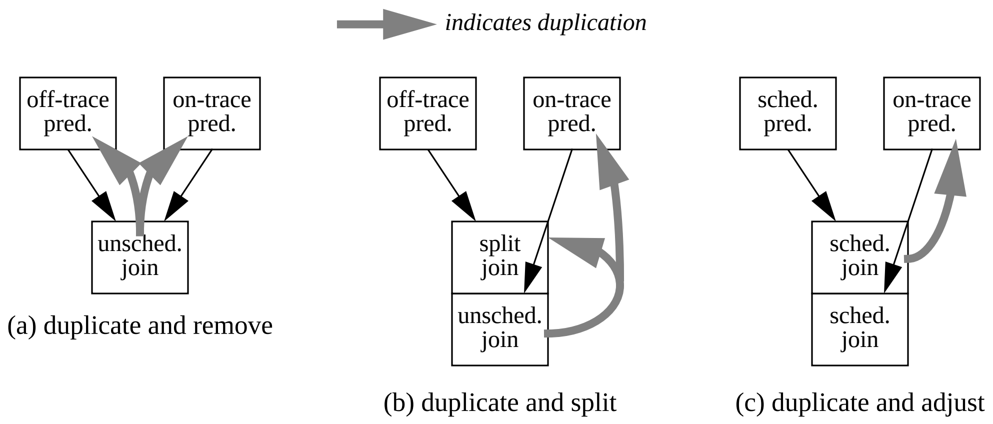


Figure 3-11: Examples of each duplication scheme.

We refer to this action as *duplicate and split (DaS)*, and it is illustrated in Figure 3-11b. The tradeoff though is that we need to be able to adjust the branch target of the on-trace predecessor. Consequently, our global scheduler only fails to duplicate an instruction when the following two conditions both hold: the on-trace predecessor does not end in a branch or jump instruction, and one of the off-trace predecessors is already scheduled or ends in a conditional branch. As an aside, Figure 3-11c shows how our global scheduler handles duplication from an already-scheduled join block; *duplicate and adjust (DaA)* is very similar to DaS except the duplicate is already scheduled. Figure 3-12 presents a code segment for determining the type of duplication required.

```
if (join block scheduled) {
   duplicate_and_adjust;
} else if (any predecessor block is scheduled) {
   if (on-trace predecessor ends in branch/jump)
      duplicate_and_split;
   else
      duplication_not_possible;
} else if (off-trace predecessor ends in branch) {
   /* cannot duplicate into branch-ending basic block */
   if (on-trace predecessor ends in branch/jump)
      duplicate_and_split;
   else
      duplication_not_possible;
} else {
   duplicate_and_remove;
}
```

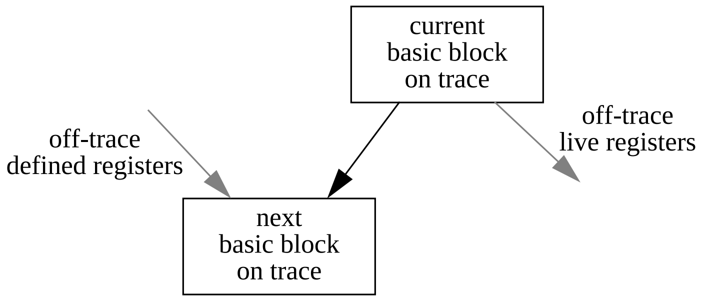
Figure 3-12: Code for determining the type of duplication required.

Since the global movement of a data-ready instruction in the EDAG can only cause the duplication and/or the speculative execution of that instruction, the constraints on these two actions are the only constraints on the availability of that instruction. Thus, we can summarize the availability of an instruction by checking the duplication and speculative execution constraints on the global movement of that instruction. The *earliest availability* of an instruction is the first point along the trace (proceeding from the home basic block of that instruction to the seed basic block of the trace) where the compiler and/or the hardware does not have enough sophistication to support the movement of the instruction across the next CFG edge. Remembering the basic block at this point is the easiest way to summarize earliest availability. For example in a system without hardware-assisted speculative execution, the earliest availability of instruction `i6` in Figure 3-10 is block `E` because the movement of this instruction across the edge `DE` would result in unsafe speculative execution, and the earliest availability of instruction `i7` in Figure 3-10 is block `B` because the movement of this instruction across the edge `AB` would result in illegal speculative execution.

3.4.2.2 Our structures for bookkeeping

To explain how we initialize and update this availability information, we first need to explain how we summarize the constraints on the global code motions along the trace. Our global scheduler summarizes these constraints to further improve the compile-time efficiency of our algorithm. For the global scheduler to calculate availability and perform global code motions, it requires information about control dependence and off-trace data dependence. Control dependence and off-trace data dependence indicate when duplication and speculative execution are necessary. This dependence information is a property of the CFG, and we therefore summarize it with the trace. The data structure which summarizes this dependence information is referred to as the *global constraint list* (GCL).

When the trace only contains the seed basic block, all of the instructions in the EDAG are available because only intra-block code motion is possible. At this point, the GCL is empty. Once the global scheduler chooses a new basic block to add to the end of the trace (called the *next* basic block by Section 3.3.3), the scheduler creates what we call a set of *global constraints*, and it adds these global constraints to the GCL. These global constraints summarize the constraints on a global code motion across the CFG edge between the current basic block and the next basic block. The type or types of global constraints generated depend on how the current basic block ends and on how the next basic block can be entered (see Figure 3-13). If the next basic block has multiple predecessors, the scheduler inserts a DEF constraint into the GCL. This DEF constraint contains the off-trace global dataflow information necessary for the scheduler to determine if it should duplicate an instruction when it moves that instruction across this trace edge. If the current basic block has multiple successors, the scheduler inserts a LIVE constraint into the GCL. This LIVE constraint contains the off-trace global dataflow information necessary for the scheduler to determine if an instruction results in illegal speculative execution when the scheduler moves that instruction across this trace edge. If the only successor of the current basic block is the next basic block and the only predecessor of the next basic block is the current basic block, then the global scheduler inserts a *NULL constraint* into the GCL. A NULL constraint is simply a placeholder that indicates that there is no restriction on the movement of instructions across this CFG edge. Of course, a single CFG edge can require both a DEF and a LIVE constraint.


current basic block on trace
off-trace defined registers
off-trace live registers
next basic block on trace

(a) Off-trace edge possibilities from bottom of current basic block and top of next basic block.

		No. of successors of current basic block: 1	No. of successors of current basic block: 2
No. of predecessors of next basic block	1	NULL constraint	LIVE constraint
	many	DEF constraint	LIVE & DEF constraint

(b) Table of global constraints created for each combination of off-trace edge possibilities.

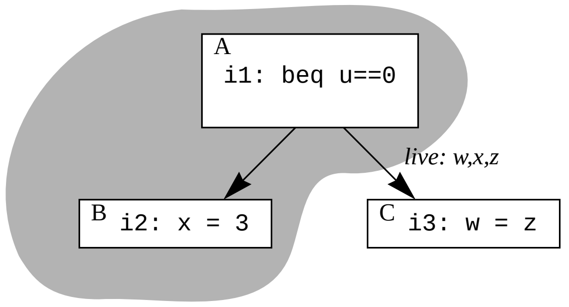
Figure 3-13: Summary of global constraints.

3.4.2.3 Building the bookkeeping information

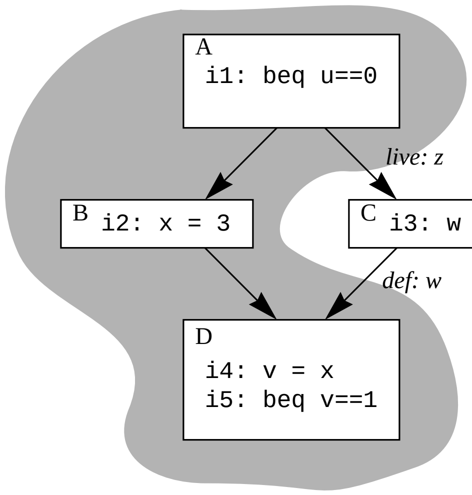
As an example of the GCL and its use, we describe how we construct the GCL for the trace in Figure 3-10 (see Figure 3-14). The GCL is constructed as the trace and the EDAG are constructed. Initially, the trace only contains block A (our seed block), the EDAG only contains the instructions in block A, and the GCL is empty. When we add block B to the trace A (Figure 3-14a), the multiple successors of block A indicate that speculative execution is required for any code motion across the CFG edge between blocks A and B (block A is a branch-ending block). To determine whether the speculative execution of an instruction which moves across this CFG edge is safe and legal, the global scheduler requires exception information about the moving instruction and global dataflow information about the registers that are live on entry to block C (the off-trace successor of block A). Since the exception information is an attribute of the instruction, the global scheduler only needs to summarize the off-trace information about the registers which are live on entry to block C. This information is kept in the LIVE constraint for that CFG edge. In Figure 3-14a, the LIVE constraint between blocks A and B contains the register set {w,x,z}; these are all of the variables that a speculatively-executed instruction from block B could destroy. Since block B has only a single predecessor, this LIVE constraint is the only global constraint on the CFG edge between blocks A and B.

Once we know what is required to globally move an instruction from block B to block A, the scheduler uses the GCL to determine the earliest availability for each instruction in block B by checking the duplication and speculative execution constraints along the trace. That is, as the scheduler inserts instructions into the EDAG, it also simulates the global code motion of that instruction from block B (home) to the block A (seed). This simulation involves the updating of the GCL due to a duplication (we will talk more about how the GCL is updated in the next subsection), but the simulation does not actually perform the duplication. Each global code motion is simply done to determine the earliest availability for that instruction.

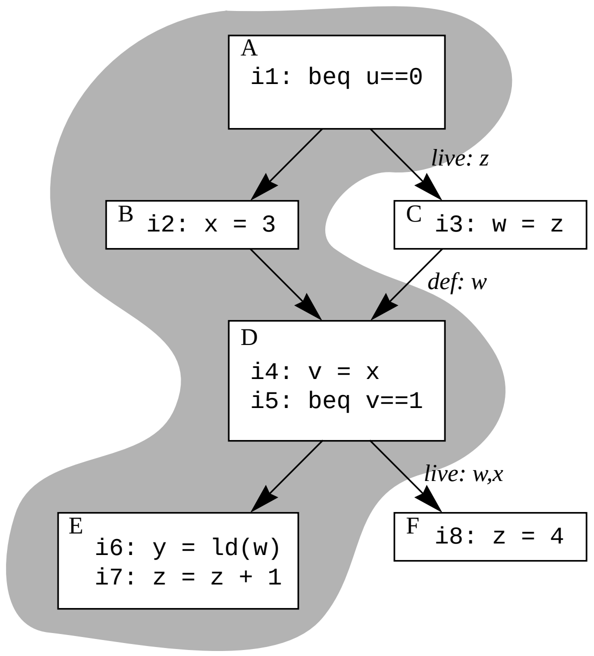
When we next add block D to the trace AB (Figure 3-14b), the multiple predecessors of block D indicate that duplication might be required for a code motion across the CFG edge between blocks B and D (block D is a join block). For this CFG edge then, we create a DEF constraint with a register set that contains all of the registers that are defined in the off-trace predecessors of block D. The global scheduler uses this DEF register set to determine when duplication is necessary. Specifically, the global movement of an instruction above a join block requires duplication if the sources (a true data dependence) or the destination (an output data dependence) of the moving instruction are in the DEF register set. For the example in Figure 3-14, block C only defines register w so the DEF constraint on the edge BD only contains the register w. The movement of instruction i4 from block D to block A (assuming prior movement of instruction i2) does not require duplication because the early execution of i4 is not affected by the instructions in block C. On the other hand, the movement of instruction i6 from block E to block A does require duplication.



(a) Added block `B` to trace `A`



(b) Added block `D` to trace `AB`



(c) Added block `E` to trace `ABD`

Figure 3-14: Example of GCL construction.

Alternatively, if the trace had begun with block B instead of block A, then the DEF constraint on the edge BD would contain the *universal* set (the set of all of the registers in the architecture) so that duplication always takes place. Unconditional duplication is necessary in this case because no block in the trace BDE dominates block C; unconditional duplication ensures that the operation is executed when the control flow enters the trace from this off-trace edge.

Returning to the example in Figure 3-14b, notice that when we added the basic block D to the trace AB we changed the register set of the LIVE constraint on the edge AB. This change may seem strange since we just finished saying that a global constraint for a CFG edge is only dependent upon the entry and exit conditions of the blocks at the ends of that edge. Well, the type of global constraint is determined as we previously discussed, but the contents of the register set depend upon another factor. The reason for a change in the register set of a global constraint is to support the equivalence optimization of Figure 3-9.

When we add block D to the trace AB, we can immediately determine that the blocks A and D are control equivalent. We would like the GCL to summarize the off-trace data dependence information that we need in order to determine if these two blocks are data equivalent with respect to a moving instruction. If they are data equivalent, then the global movement of that instruction does not require speculative execution or off-trace duplication. As we said earlier, the EDAG captures all of the data dependences on the trace, and thus the GCL only needs to capture those data dependences off the trace. The DEF register set for the edge entering the join block of the control-equivalent pair (edge BD in Figure 3-14b) captures the off-trace true and output dependences because it contains all of those registers that are defined in the off-trace predecessors of block D. The off-trace anti-dependences are summarized by the set of live registers in the off-trace subgraph which is reached by taking the off-trace successor of the branch-ending block of the control-equivalent pair (edge AC in Figure 3-14b). Before we changed the LIVE register set of CFG edge AB, it indicated the live-register set for all of the blocks reachable from the CFG edge AC. By changing this set to contain only those registers which are live in the off-trace subgraph between the control-equivalent blocks, we can use these two global constraints to determine whether data equivalence holds for this off-trace subgraph.

Actually, the blocks A and D in Figure 3-14b are what we call two *ideally* control-equivalent blocks, and the implementation of our global scheduler only applies the equivalence optimization to ideally control-equivalent blocks. To help in the precise definition of ideal control equivalence, assume that we are given a branch-ending block A and a join block B where A is the immediate dominator of B and B is the immediate post-dominator of A so that A and B are control equivalent. Blocks A and B are *ideally control equivalent* if the following two conditions hold true: A does not immediately dominate any other join block between A and B; and B does not immediately post-dominate any other branch-ending block between A and B. Ideal control equivalence guarantees that the two subgraphs, rooted at the branch edges of A and terminated at the join edges of block B, are completely separate and that they have a single entry and exit point. These conditions guarantee that our global scheduler can easily and accurately summarize the dataflow information for the off-trace subgraph. We could improve our scheduler so that it attempts to apply the equivalence optimization to all of the control-equivalent blocks in our applications by transforming a pair of control equivalent blocks that are not ideally control equivalent into ones that are. This transformation simply involves the duplication of blocks and the splitting of join blocks as demonstrated in Figure 3-15. We chose not to include this optimization at this time because a large fraction (over 55%) of the control-equivalent pairs in our benchmark applications are also ideally control equivalent. In summary, the type of global constraint on a CFG edge in the trace is only dependent upon the entry and exit conditions of the blocks at the ends of that edge, but the contents of the register set of a DEF (LIVE) constraint are dependent upon whether or not the successor (predecessor) of the CFG edge is part of an ideal control-equivalent pair.

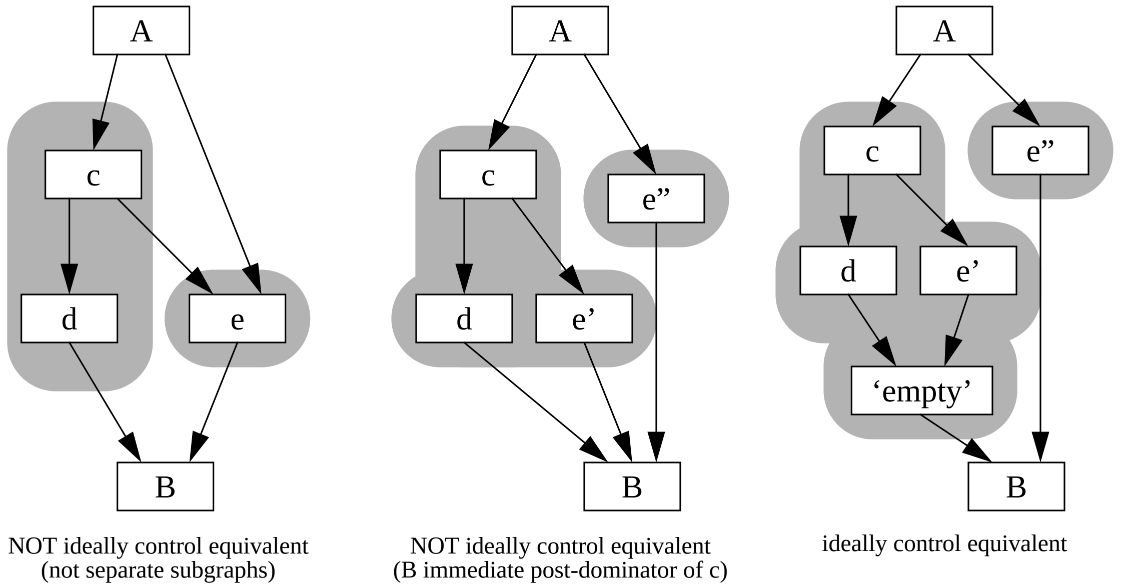

A
c
d
e
B
NOT ideally control equivalent
(not separate subgraphs)
A
c
e”
d
e’
B
NOT ideally control equivalent
(B immediate post-dominator of c)
A
c
e”
d
e’
‘empty’
B
ideally control equivalent

Figure 3-15: Transforming control-equivalent blocks into ideally control-equivalent blocks.

Once we determine the global constraints for moving an instruction from block D to block A, the global scheduler simulates this global movement to determine the earliest availability of each instruction in block D. With this done, we can finally add block E to the end of our trace, and the creation of its global constraint proceeds in a similar fashion to the addition of block B (Figure 3-14c). Again, as the instructions in block E are inserted into the EDAG, the earliest availability of each instruction is determined by simulating the global movement of that instruction from block E to block A.

Figure 3-16 lists an algorithm that builds and initializes the GCL as the trace is constructed; this algorithm is called by the algorithm in Figure 3-5 on page 49. Since the construction of our trace proceeds in the direction of the edges in the CFG, the algorithm in Figure 3-16 only checks for an ideal control-equivalent pair when a DEF constraint is inserted into the GCL. The discovery of an ideal control-equivalent pair of blocks results in a recalculation of the pair's LIVE constraint register set as we discussed in the example in Figure 3-14. Once the global scheduler has constructed the trace, EDAG, and GCL, it proceeds to schedule each basic block in the trace in a top-down manner as described in Figure 3-5.

```
if (current_basic_block ends in conditional branch) {
   /* movement requires check for speculative execution */
   add LIVE constraint to GCL;
   reg_set = live-var set of off-trace edge;
}

if (next_basic_block has multiple predecessors) {
   /* movement requires check for duplication */
   add DEF constraint to GCL;
   if (immediate_dominator of next_basic_block is on trace
       && immediate_dominator is ideally control equivalent) {
      reg_set = all_def set for off-trace paths between
                immediate_dominator and next_basic_block;
      LIVE_constraint reg_set of immediate_dominator =
                live_var set for off-trace paths between
                immediate_dominator and next_basic_block;
   } else {
      reg_set = universal function; /* always duplicate */
   }

   determine type of duplication required (Figure 3-12);
}
```

Figure 3-16: Algorithm for building and initializing the GCL.

3.4.2.4 Updating the bookkeeping information

Up to this point, the discussion has always assumed that the register sets of the GCL contained the appropriate global dataflow information. Each time the global scheduler performs a global code motion though, the program graph changes and therefore the global dataflow information changes. These changes require the scheduler to update the register sets in the GCL to reflect the new global dataflow information. Our scheduler incrementally updates the dataflow information in the GCL so that the updates are done in a compile-time efficient manner.

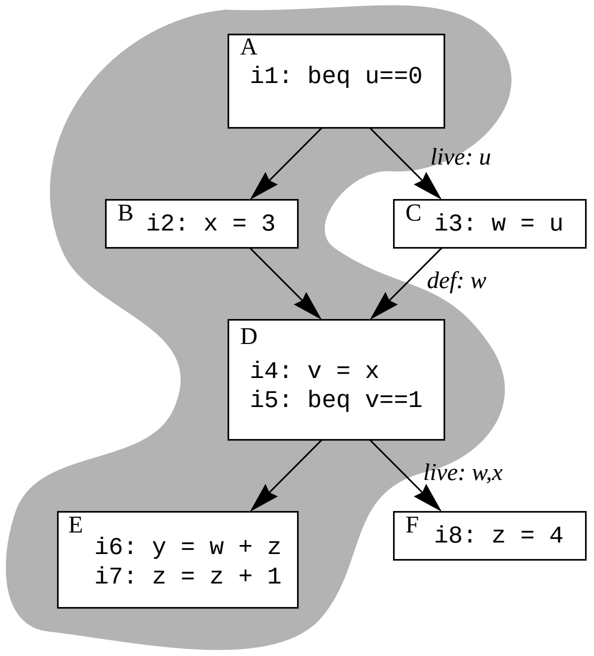
Even though the global dataflow information changes whenever a global code motion occurs, our global scheduler only needs to update those global constraint register sets that are generated from an ideal control-equivalent pair. The following two claims support this statement. The first claim is that a global code motion from a block later in the trace to the block at the head of the trace can never increase the register set of a DEF constraint that is not generated from an ideal control-equivalent pair. The proof of this claim is straightforward. A register set of a DEF constraint that is not generated from an ideal control-equivalent pair is currently the universal set so duplication into the off-trace predecessor blocks cannot possibly increase this set. The second claim is that a global code motion from a block later in the trace to the block at the head of the trace can never increase the register set of a LIVE constraint that is not generated from an ideal control-equivalent pair. The proof of this claim implies that a global code motion cannot increase the set of *off-trace* live variables. The global movement can obviously increase the set of live variables for the on-trace edge by simply moving up a definition of a previously dead register. However, to affect the set of off-trace live variables, the moving instruction must have started in or after a join block so that there exists a path from the off-trace branch edge to the home block of the moving instruction. Now, either the uses of the moving instruction are in the live-variable set or they are not, and a duplication at the join cannot change this off-trace fact. The only way to increase the off-trace live-variable set is to move the instruction up so its destination is now live. If the join block was not part of an ideal control-equivalent pair, the moving instruction is automatically duplicated and so the destination register cannot become live. If the join block is part of an ideal control-equivalent pair so that duplication does not have to occur, the pair must be the branch block of the ideal control-equivalent LIVE constraint, and this claim does not apply.

For register sets of global constraints that are generated from an ideal control-equivalent pair, our global scheduler updates the register sets in these constraints when a duplication occurs into the off-trace subgraph. A duplication into the off-trace subgraph adds the duplicated instruction's destination register to the DEF constraint's register set—the duplication causes a new definition in the off-trace subgraph. A duplication also affects the LIVE constraint's register set because the duplication might change the off-trace live-variable set. We update the live-variable set of the LIVE constraint by incrementally recalculating the live-variable sets of the basic blocks in the off-trace subgraph; a single upward walk of the subgraph is sufficient.

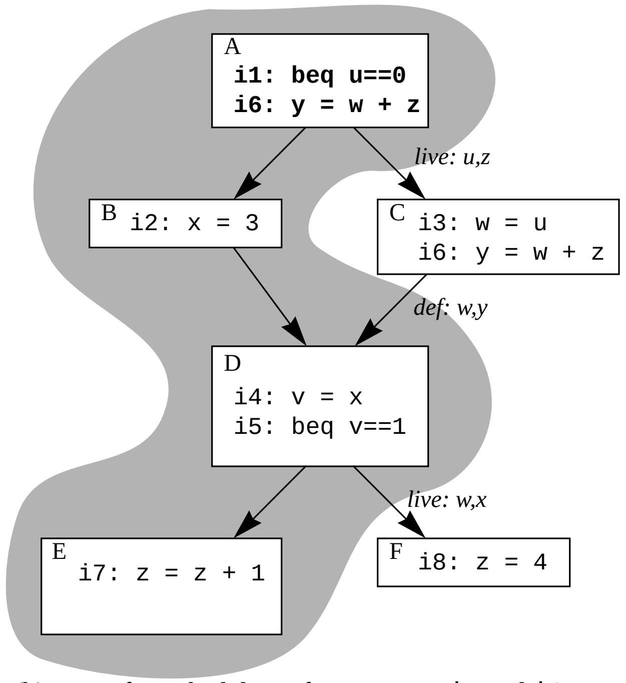
Figure 3-17 illustrates how our global scheduler updates the bookkeeping data structures during the scheduling of the trace `ABDE`. (This example CFG is almost identical to the one used in Figure 3-14, except that we changed instructions `i3` and `i6` so that we could move `i6` to block `A` without encountering an unsafe or illegal speculative execution.) The global scheduler begins by scheduling block `A`. Figure 3-17b shows the state of the CFG and the GCL after the scheduler issues instruction `i1` and a copy of instruction `i6` in block `A` (we show issued instructions in bold). The global code motion of instruction `i6` required the scheduler to place a duplicate of `i6` (compensation code) in block `C`, since one of the operands of instruction `i6` was redefined in block `C`. This duplicate then caused the scheduler to update the register sets of the LIVE and DEF constraints which depend upon the control-equivalent pair `AD`. Notice that this update restricts the upward movement of instruction `i7`. If the scheduler had not updated the global constraints, the global constraints in Figure 3-17a would have allowed instruction `i7` to issue in block `A` (thus causing an error).

Figure 3-18 is a continuation of Figure 3-17, and it shows the state of the CFG and the GCL after the scheduler has completed the scheduling of block `A` and after it has moved on to begin scheduling block `B` (we show scheduled blocks in bold). When we finished scheduling block `A`, we removed the global constraints for the trace edge `AB` from the GCL since the global scheduler will no longer move any instructions across this edge. Since block `A` was part of an ideally control-equivalent pair, the register set of the DEF constraint between blocks `B` and `D` has reverted to "top" (the universal set) to indicate that duplication should always occur for a global code motion across this edge. If block `A` was not part of an ideally control-equivalent pair, then the other global constraints in the GCL would not be affected by the removal of block `A` from the trace. During scheduling, the GCL always tracks the current state of the CFG and the trace.

Figure 3-19 inserts all of the bookkeeping specifics into our algorithm for building and scheduling a trace (previously Figure 3-5). To review, our global scheduling algorithm



(a) CFG after trace construction and before scheduling.



(b) CFG after scheduling of instructions `i1` and `i6`.

Figure 3-17: Example of GCL update.

performs *at most two* global movements for each instruction in the trace. The first global code motion occurs during the building of the trace/EDAG/GCL (this global code motion is used to determine the earliest availability of the instruction), while the second global

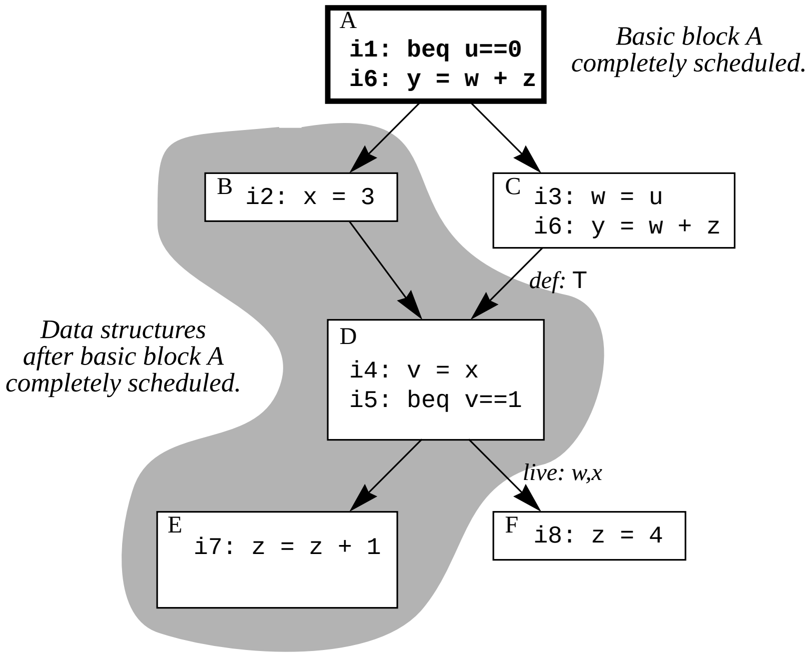


Figure 3-18: Example of GCL and trace update.

code motion occurs during the scheduling of the trace (this global code motion is only necessary if the issued instruction originally existed in a basic block other than the one that we are currently scheduling). For the determination of earliest availability, our global scheduler only simulates the global movement of an instruction. This simulated global movement may require an update of the GCL, but it does not produce any compensation code because we have not yet determined the actual instruction schedule. Once the global scheduler has completely built the trace and its data structures for bookkeeping, the global scheduler is ready to schedule the trace. Since the GCL should now represent the program graph before any global movements, each global constraint in the GCL reverts its register set to the state that the register set was given when we first created the global constraint (i.e. the state that originally existed before we simulated any global movements across each CFG edge). The global scheduler then schedules the basic blocks in the trace in a top-down fashion. As the global scheduler completes the scheduling of a basic block, it removes the block from the trace, updates the EDAG instruction lists, and deletes the global constraints of the completed basic block and trace edge.

3.5 Scheduling support for boosting

Opportunistic instruction scheduling techniques such as branch speculation are orthogonal to and independent of our global scheduling algorithm. Opportunistic instruction

```
build_and_schedule_trace(BB *sBB /* “seed” of trace */) {
   trace, EDAG, and GCL initially empty;
   nBB = sBB;
   while (nBB) {     /* build */
      foreach (instruction i in nBB) {
         insert i into EDAG;
         simulate move of i from nBB to sBB;
         remember earliest availability of i;
      }
      choose next nBB (if any);
      if (nBB) compute and add new constraints to GCL;
   }
   prioritize EDAG;
   re-initialize GCL;
   nBB = sBB;
   while (nBB) {     /* schedule */
      cBB = nBB;
      schedule cBB (perform any global code motions necessary);
      remove cBB from trace;
      delete global constraints associated with cBB from GCL;
      special case operations for special case scheduling;
      nBB = next basic block in trace (if any);
   }
}
```

Figure 3-19: Bookkeeping specifics for building and scheduling a trace.

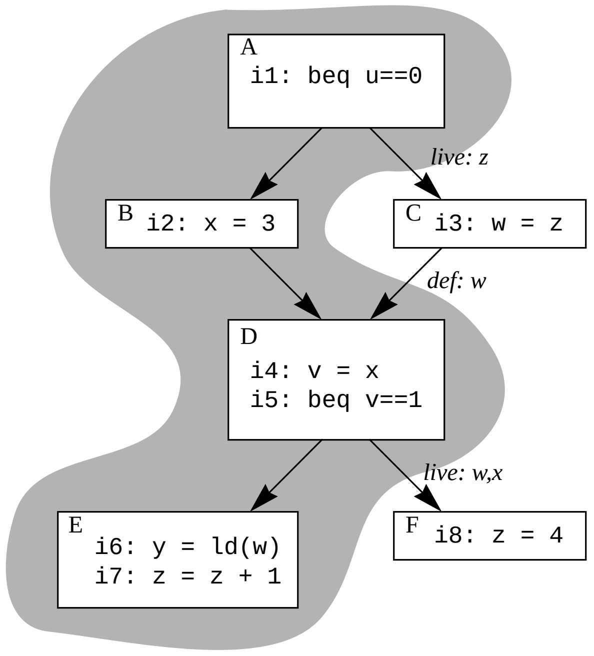
scheduling techniques simply loosen the run-time-dependent scheduling constraints between instructions so that the compiler is not conservative in its scheduling decisions. The architectural mechanisms like boosting increase the power of the global scheduler by permitting a greater range of global code motions. This section focuses on how boosting augments the capabilities of our global scheduling algorithm.

Boosting is an architectural mechanism for hardware-assisted speculative execution which provides the compiler with the ability to move any instruction above its control dependent branch. In relation to the upward code motion routine listed in Figure 3-9, this capability means that code motion into the bottom of a basic block is always possible. With boosting, unsafe and illegal speculative movements do not inhibit the upward movement of an instruction. We simply label any speculative instruction that would result in an unsafe or illegal speculative execution as boosted instruction, and the hardware guarantees that the effects of this instruction only affect the non-speculative state of the machine if the instruction commits.

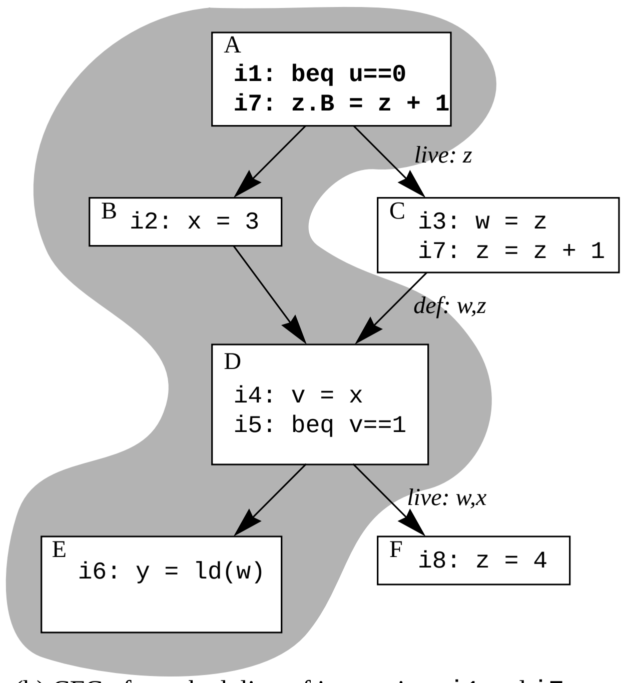
The discussion now turns from how boosting affects the upward code motion routine to how boosting affects the our global scheduling algorithm. As it turns out, our trace-based approach to global scheduling and our trace-based simplification of boosting (as described in Section 2.1.2) mesh quite well together. Whenever the global movement of an instruction above a conditional branch would result in an unsafe or illegal speculative execution, the scheduler labels that instruction as a boosted instruction which is dependent upon the conditional branch. From that point on, the instruction is considered dependent upon every other conditional branch that the global scheduler moves it above. This automatic dependence is because boosting encodes control dependence information as a count of the following branches. The number of branches that the instruction finally depends upon is the boosting level of the instruction. If the hardware can support this many levels of boosting, then there is no speculative execution constraint on the availability of the instruction. For hardware configurations with support for fewer levels of boosting, the earliest availability of the instruction is the point before the CFG edge that requires a level of boosting greater than that supported in the hardware (assuming no earlier duplication that the compiler cannot support).

Boosting also affects the actions of the global scheduler during the generation of compensation code. As previously stated, the processor discards the effects of a boosted instruction when a dependent branch for that instruction is incorrectly predicted. This action differs from the execution of a safe and legal speculative instruction (a speculative instruction that is not boosted) because this speculative instruction affects the program state independent of the execution of the dependent conditional branch. Thus, our global scheduler must check to make sure that the boosted operation was not needed on the non-predicted path of the branch. If it is needed, the global scheduler must insert a duplication to recalculate the operation even if the global constraints did not indicate that duplication was necessary. Figure 3-20 illustrates a case where duplication into block `C` is necessary for the global movement of instruction `i7`, but the DEF constraint on the trace edge `BD` did not indicate the need for this duplication.

Specifically, our global scheduler handles this check and duplication in the following manner. First remember that the scheduler always duplicates into the off-trace predecessors of a join block that is not part of ideally control-equivalent pair. Thus, the only way duplication could possibly not occur is if the branch-ending block (that caused the scheduler to label the instruction as boosted) is part of an ideally control-equivalent pair, if the boosted instruction moved through the join of this ideally control-equivalent pair on its way up the trace (i.e. the boosted instruction's home block is that join block or some block later in the trace), and if the DEF constraint of that join block did not force a duplication. If these three conditions hold, as they do for Figure 3-20, then the scheduler performs the



(a) CFG after trace construction and before scheduling.



(b) CFG after scheduling of instructions `i1` and `i7`.

Figure 3-20: Example of boosting-initiated duplication.

appropriate duplication into the off-trace predecessor of the join block. Notice that the third check is required to ensure that duplication does not occur twice, i.e. the DEF constraint did not force duplication on its own. To summarize, the movement of a boosted instruction across a pair of ideally control-equivalent blocks always requires a duplication into the off-trace predecessor of the join block of that pair. This duplication is a result of the semantics of boosting and is independent of the DEF constraint check. Since our scheduler maintains a set of pointers between the global constraint of a ideally control-equivalent pair, a check for this special duplication is simple to perform.

To support precise exceptions under boosting, Section 2.1.3 described our recovery-code scheme where exception processing for a boosted instruction is postponed until the commit point. This scheme requires the hardware to re-execute the previously boosted instructions and all of their data-dependent successor instructions that were also moved above the commit point. To support this re-execution, none of these instructions can destroy a value that might be required during exception processing. Boosted instructions never destroy a value needed during exception processing because they only modify speculative state. Yet, a safe and legal speculative operation that uses a boosted value might inadvertently destroy a value needed during exception processing. The simplest way to ensure that this does not happen is to label as boosted any instruction that uses a boosted operand or that is anti- or output dependent upon a boosted instruction. This solution also simplifies the determination of what instructions need to be copied into the recovery code; the recovery code consists of every boosted instruction. To summarize, the scheduler labels a speculative instruction as a boosted instruction if any of the following five conditions are true:

(1) the speculative instruction is already boosted (hardware constraint),

(2) the speculative instruction can cause an exception to occur (unsafe speculative execution),

(3) the speculative instruction destroys a live value on the off-trace edge of the branch (illegal speculative execution),

(4) the speculative instruction uses a value produced by a boosted instruction (true data dependence with a boosted instruction), or

(5) the speculative instruction writes a register that is read or written by a boosted instruction (anti- or output dependence with a boosted instruction).

Though boosting changes the global constraints on availability and the compensation code produced during a global code motion, boosting does not change the bookkeeping data structures. Boosting, like any other opportunistic instruction scheduling mechanism, simply minimizes the constraints on code motion, and provides the compiler with more opportunities for generating a good instruction schedule.

3.6 Summary

Compilers perform static instruction scheduling with the goal of minimizing program execution time by minimizing processor stalls. For parallel machines such as superscalar processors, the instruction scheduler uses the exploitable ILP in an application to hide the latencies from resource conflicts, data-dependence stalls, and control-dependence stalls. Because most non-numerical applications contain very little exploitable ILP within a basic block, we rely on global instruction scheduling techniques to find extra ILP across the basic block boundaries. This chapter discussed the components of a typical global scheduling algorithm, and it reviewed some previous works in the field of global scheduling. Though the previous works differed in their support for global transformations and in their determination of which instructions are available for scheduling, all of these algorithms tried to uncover as much ILP as they thought it was feasible to uncover.

The chapter also described a new global scheduling algorithm that is tailored for the uncovering of ILP within non-numerical applications. We designed the algorithm to be compile-time efficient, easily extensible, and highly flexible so that it is useful on a wide range of machines and so that it complements other techniques for uncovering and exploiting ILP (e.g. specialized optimizations such as software pipelining and procedure inlining). Our global scheduling algorithm can be thought of as conscientious trace scheduling because we chose our limited set of global transformations so that we minimized the potential execution-time penalties in the off-trace blocks due to an on-trace global code motion (i.e. penalties from excessive compensation code). Though a trace-based approach limits the size of the set of available instructions (from which we find ILP), a trace ensures that this set contains those instructions of the larger set which are most beneficial to the creation of a good global schedule. Our trace-based approach also suggests solutions for many of the problems associated with the original Trace Scheduling algorithm such as a lack of overlap between traces and a complex bookkeeper. The key to our simplification and optimization of the bookkeeping process is the observation that the availability of an instruction is an attribute that the global scheduler can precompute and summarize.

Chapter 2 argued the need for opportunistic instruction scheduling techniques because a compiler's pessimism about run-time-dependent information lead to lost opportunities for exploiting ILP. This chapter showed how boosting could augment the capability of the global scheduler by increasing the scope of the availability calculation. The next chapter uses our global scheduling algorithm to investigate the hardware costs and performance benefits of boosting.

Chapter 4

Evaluating Hardware Support for Scheduling

In the last two chapters, we looked at the compiler’s ability to exploit ILP through global instruction scheduling and at a way of enhancing the capabilities of a global scheduler through boosting. In this chapter, we investigate the cost of the hardware required to support boosting, and we evaluate how the level of hardware support for boosting affects overall performance. As we mentioned in the introduction, the addition of extra hardware features like boosting will only improve the overall performance of a processor if we can implement these features without adversely affecting the cycle time of that processor. To appropriately quantify the effects on cycle time, we discuss the organization and the cycle time impact of the hardware structures required to support boosting in a specific processor microarchitecture which we call TORCH. TORCH is a superscalar processor based on the MIPS R2000 processor architecture [Kane 1987].

4.1 Background

To better understand the simulation system that we use to generate the results in this chapter, this section briefly overviews our compilation environment, our benchmark programs, and our simulation methodology.

4.1.1 Environment

Our experimental setup consists of a compilation system and a simulation system which are both based upon the MIPS R2000 processor architecture. Table 4-1 lists the set of benchmarks that we ran through this experimental system to generate the results presented in this thesis. All of these programs are written in C, and all were run to completion. Three of the benchmarks (eqntott, espresso, and xlisp) are from the original SPEC benchmark suite [SPEC 1990], while the other four benchmarks (awk, compress, grep, and nroff) are standard UNIX utilities.

	Total R2000 Cycles	Avg. R2000 IPC	Branch Prediction Accuracy
awk	52.6M	0.88	82.0%
compress	29.3M	0.87	82.7%
eqntott	1.0M	0.93	72.1%
espresso	101.4M	0.88	75.7%
grep	28.6M	0.81	97.9%
nroff	67.0M	0.81	96.7%
xlisp	1.0M	0.89	83.5%

Table 4-1: Benchmark programs and their simulation information.

We ran each benchmark on two different input data sets. We used the first input to generate the branch profile used by the global scheduler, while we used the second input to run the actual superscalar simulation. Like the study done by Fisher and Freudenberger [1992] on profiled branch prediction, we tried to find input data sets with very different characteristics so that our branch prediction accuracy would not be overinflated. We chose the data set pairs so that the characteristics of our pairs varied in both execution time and execution profile. Table 4-1 presents the total execution cycles and average instructions per cycle (IPC) values for each benchmark when that benchmark was executing the second of its input data sets on a MIPS R2000 processor with a perfect memory system (i.e. the caches never miss). The final column in Table 4-1 is the overall accuracy of the static branch prediction during this run.

We compiled all of the benchmark applications under the SUIF compiler system which was developed at Stanford University [Tjiang et al 1991]. The SUIF compiler takes a source file and generates an optimized assembly file; the optimizer in this compiler implements all of the standard optimizations [Tjiang and Hennessy 1992]. To generate the MIPS R2000 numbers in Table 4-1, we scheduled the assembly file with the commercial MIPS assembler (version 1.31), and then we ran this object file under pixie to collect the execution statistics [MDSmith 1991]. To generate the performance metric for our superscalar machine models, we also scheduled this optimized assembly file with our global scheduler. We talk about the specifics of the global scheduler implementation and the superscalar simulator in the next two subsections. The performance metric for the superscalar machine models is *speedup,* where speedup is defined as the total number of R2000 processor cycles taken to execute a program divided by the total number of superscalar processor cycles taken to execute the same program.

SUIF is an experimental compiler system, and thus we need to put the results produced by this system in some perspective. The advantage of an “in-house” compiler system is that the structure of the compiler is flexible and easy to change; the disadvantage of an “in-house” compiler system is that the code produced by this compiler may not always run as fast as the code produced by a good commercial compiler. To determine how the SUIF compiler compares with a good commercial compiler, we re-compiled and re-optimized (at optimization level -O2) our benchmark programs with the commercial MIPS cc compiler (version 1.31). After running these new object files with the second of the input data sets, we discovered that the SUIF-compiled object files were 10–35% slower than the MIPS-compiled object files. The concern with this situation is that the SUIF object files might contain extra, redundant work that is easily executed in parallel on the superscalar processor, thus overinflating the performance numbers of the superscalar machine models. Fortunately, this is not the case.

The reasons for the longer execution times in the SUIF-compiled object files stem from a variety of causes, but the main causes are longer instruction sequences for some operations and poor register allocation in important loops. We found that, even though the absolute execution time of a SUIF-generated object file is greater than a MIPS-generated object file, the relative performance improvement of both object files is the same when scheduled for and run on the same superscalar machine model. That is, speedup of the superscalar machine model is unaffected as long as we start both the MIPS scheduler and our global scheduler with the same optimized assembly file. To prove this point, the implementation of global scheduler scheduled a number of MIPS-cc-generated assembly files. The speedups produced by these MIPS-compiler-based simulations varied within the range of +15% to -15% of the speedups produced by the SUIF-compiler-based simulations. The reason that we use the SUIF compiler instead of the commercial MIPS compiler is that the SUIF compiler supports more experiments than are possible with the commercial MIPS compiler. For instance, the SUIF compiler can generate an assembly file with an infinite register model, and this is not easily possible with the commercial MIPS compiler.

4.1.2 Specifics of scheduler implementation

Twine is a C++ implementation of our global scheduling algorithm from Chapter 3. We structured Twine so that it easily handles a wide variety of superscalar machine models. Within Twine, one can change the types of functional units, the distribution of the functional units within the issue packet, the issue and result latency rates of the functional units, the size of the parallel issue, the handling of exceptions, etc. Of course, one can also vary the degree of hardware support for speculative execution (e.g. the number and types of hardware buffers included to support boosting).

The input to Twine is a MIPS assembly language file, and from this file, Twine produces a globally-scheduled object file. To assist in the global scheduling process, the MIPS assembly language file may contain branch profile information. If the profile information is not present, Twine predicts the direction of conditional branches with the simple heuristic that backward branches are predicted to take and forward branches are predicted to not take. This simple heuristic achieves a branch prediction accuracy of only 60–70% for most programs, and it noticeably limits the speedups of the superscalar processors. Since accurate branch prediction is very important for our trace-based global scheduling algorithm, the SUIF compiler automatically inserts branch profile information into the MIPS assembly language files. All of the results in this thesis use branch profile information, and as we mentioned in the previous subsection, the SUIF compiler generates this profile information with a different input than the input that is run in the superscalar simulation.

Since Twine is part of an experimental compiler system, Twine only implements those key features which are required to perform instruction scheduling. We know of a number of additional techniques which we did not implement in Twine that could possibly improve the performance of our scheduled code. For instance, we could perform integrated register allocation and instruction scheduling to minimize any register reuse conflicts. Currently, Twine simply accepts the register allocation that is performed in the earlier phases of the SUIF compiler (register allocation in SUIF follows a round-robin allocation scheme for temporary variables, and it uses a coloring scheme for global variables [Chow and Hennessy 1990]). As another potential optimization, Bernstein and Rodeh [1991] have shown that scheduling with a global priority does not always produce the best basic block schedule, and they suggest the local rescheduling of a basic block after it has been globally scheduled. Currently, Twine only schedules each basic block once (using a global priority). Also, we could change the CFG during scheduling (e.g. change branch directions and add/remove jumps) to improve the scheduler's ability to globally move instructions. For the most part, we expect that these techniques would have only a minor impact on the numbers in this chapter. There are other techniques such as interprocedural analysis, procedure inlining, and loop-level optimizations (e.g. software pipelining [Lam 1988]) that we expect to have a larger influence on performance, and we are planning to investigate these techniques in the future.

4.1.3 Simulation methodology

We used three different simulation tools in our research. The first is a fully-functional, gate-level simulator of our superscalar hardware.[1] The second is an instruction-level simulator that runs our globally-scheduled code.[2] The last is a trace-driven simulator that quickly generates cycle counts for our globally-scheduled programs, given the characteristics of our superscalar hardware. Though we have used the gate-level simulator to verify the correctness of our TORCH hardware and the instruction-level simulator to verify the correctness of some of our globally-scheduled programs, we used the trace-based simulator to generate the results presented in this chapter. To generate these results, the trace-based simulator takes advantage of the fact that Twine does not reorder conditional branches. Because the branch ordering is the same in the globally-scheduled object file as it is in the R2000 object file, we can calculate the cycle count of a globally-scheduled application from a trace of the conditional branch directions in the R2000 object file.

Specifically, the trace-based simulation system works as follows. A benchmark program is compiled by the SUIF compiler to produce an optimized MIPS assembly language file. This file is then scheduled by the commercial MIPS assembler to produce a scalar object file and by Twine to produce a superscalar object file. During global scheduling, Twine generates statistics about each basic block in the application. These statistics include such items as the cycle count for a basic block, the total instruction count in that basic block, and the change in the instruction and cycle counts due to changes in the branch/jump target (a result of scheduling from already-scheduled basic blocks). Because Twine only analyzes individual basic blocks and because the execution of some instructions extends beyond the end of the basic block, each basic block actually has a minimum and maximum cycle count. The minimum cycle count is the number of execution cycles required by this basic block if none of the instructions whose execution time extends beyond the end of the basic block cause a hardware interlock or stall; the maximum cycle count is equal to the

1. The gate-level simulator was written by Tom Chanak, John Maneatis, Don Ramsey, and Drew Wingard.
2. The instruction-level simulator is based on the MIPS unimable simulator by Peter Davies. Phil Lacroute converted the unimable simulator so that it executed TORCH instructions.

minimum cycle count for the basic block plus the longest possible stall time caused by an instruction scheduled immediately after the basic block. (Since there was only a small variation between the minimum and maximum numbers, the results in this chapter report speedup as the minimum cycle count for the R2000 processor divided by the minimum cycle count for the superscalar processor.) To support statistic gathering under boosting, Twine also calculates the change in the instruction count of a basic block due to an incorrectly-predicted branch, i.e. this number represents the count of the boosted instructions whose execution was useless due to the incorrectly-predicted branch. An incorrectly-predicted branch does not affect the cycle count. Twine dumps these statistics and a copy of the CFG into a superscalar statistics file.

To generate dynamic statistics, our simulation system reads both the scalar object file and the superscalar statistics file, and it collects the basic block trace of the scalar object file as it runs on a scalar MIPS machine. We use the MIPS pixie utility to generate the scalar basic block trace [MDSmith 1991]. Our simulation system uses this scalar trace to generate a superscalar basic block trace. Though there is not a one-to-one correspondence between the basic blocks in the CFG of the scalar program and the basic blocks in the CFG of the superscalar program, the ordering of the branches in the scalar program matches the ordering of the branches in the superscalar program, and thus we are able to generate a basic block trace for the superscalar processor by checking which way each conditional branch goes. To cover the case where we only globally schedule the application code and not the libraries, our simulation system also recognizes procedure call and return instructions in the scalar trace so that it can trace through any procedures that were not globally scheduled.

Our trace-based simulation system only models the execution time of the CPU since it assumes a perfect memory system where caches never miss. Obviously, the true advantage of a superscalar processor over a scalar processor depends upon the effectiveness of the memory system. The more effective the memory system, the closer the CPU speedups presented in this chapter will represent the true speedups of the entire superscalar system.

4.2 Boosting hardware

Boosting is a powerful architectural mechanism for hardware-assisted speculative execution, and as such, boosting requires some specific hardware support. This section investigates a number of different ways of organizing that hardware support within the TORCH microarchitecture. We first look at the hardware support required for an unconstrained implementation of boosting, and we present a straightforward way of building this hardware. As we will see though, this unconstrained model of boosting requires a large amount of hardware buffering. Consequently, we describe two orthogonal approaches for reducing the amount of speculative buffering in our processor. In Section 4.3, we will use these two approaches to explore the overall performance benefit of boosting.

4.2.1 Basic support

To support boosting within the TORCH microarchitecture, Chapter 2 states that the hardware must support a method of information transfer, a separation of state, and a resolution of the speculative action. We transfer boosting information from the compiler to the hardware through extra bits in the TORCH instruction words. Section B.2 of Appendix B describes one method of encoding the boosting information.

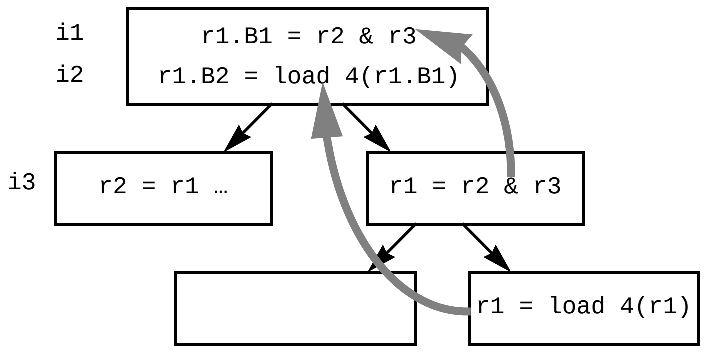
To accomplish the separation of the speculative state from the non-speculative state, we add extra buffering to the microarchitecture of TORCH. We refer to the extra buffers as the *shadow* structures since they hold the speculative state. Specifically, the shadow structures hold the effects of a boosted instruction from the time that the boosted instruction is executed until the time that the boosted effect is squashed or committed. Since our global scheduler can label any instruction as a boosted instruction, the shadow structures must capture all of the possible effects of the instructions in the MIPS R2000 architecture. The R2000 architecture is a load/store architecture [Kane 1987], and as such, there are three possible effects from instruction execution: a register is written, a memory location is written, or an exception is signalled. From this list, it is obvious that the hardware requires a shadow register file and a shadow store buffer. Our hardware includes shadow register locations for every register destination in the machine, and this includes the floating point register file and any special system registers (e.g. the floating-point condition code register). Due to limited movement of branches in the global scheduler, the hardware squashes all of the boosted branch effects in the pipeline; and thus, the program counter does not require a shadow structure. Lastly, TORCH handles boosted exceptions with our recovery-code scheme (Section 2.1.3), and thus the hardware also includes a single-bit queue.

To resolve the outcome of the speculative action, we have the hardware in TORCH check the predicted direction of the branch against the actual direction of the branch. On an incorrectly-predicted branch, the hardware invalidates all of the values in the shadow structures. On a correctly-predicted branch, the hardware updates the non-speculative state with the appropriate speculative state. That is, the hardware logically transfers a speculative value in a shadow location that commits to the non-speculative location which is paired with that shadow location. The mechanism which cheaply implements this logical move is discussed in detail in the next two subsections.

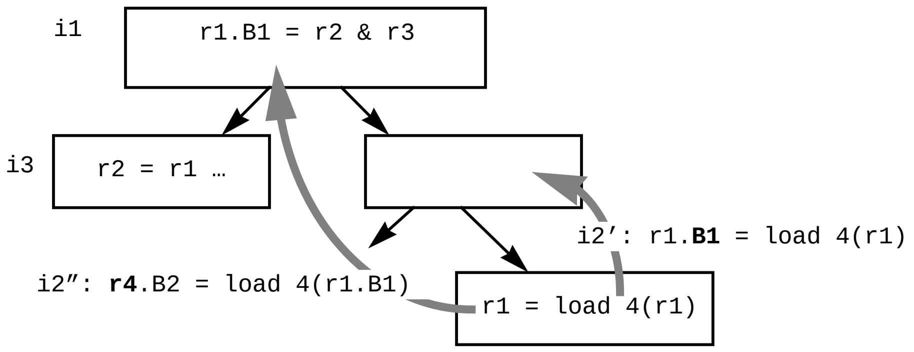
4.2.2 Full support

In the simplest case, we can think of the shadow structures as copies of the non-speculative structures. For instance, we could pair a shadow register location with each non-speculative register location. However, a single shadow location per non-speculative location only supports the speculative execution of an instruction that is dependent upon a single conditional branch (i.e. only supports a single level of boosting). To support unconstrained boosting (i.e. boosting for dependence upon multiple conditional branches) in this straightforward approach, we need multiple shadow locations per non-speculative location. Thus, if we allow *n* levels of boosting, the hardware must contain *n* shadow register locations for each sequential register location. Figure 4-1a illustrates a legal instruction schedule that is possible when we fully support two levels of boosting (FULL boosting). With a full complement of shadow structures, `r1`, `r1.B1`, and `r1.B2` are each separate physical locations.

The shadow structures for unconstrained boosting demand a large amount of buffering hardware since we require *n* buffer locations for each non-speculative location. Even so, this hardware is straightforward to implement. The only difficult aspect of the shadow structures is the implementation of the commit process, and the key insight in the implementation of this process is to realize that the data needs only to logically move on a commit. This logical move is implemented by a technique that is similar to register renaming [Keller 1975]. For each non-speculative register in the architecture, we physically build a register and counter pair for each level of boosting. The counters contain the logical name of each physical register. The register with a count value of 0 holds the non-speculative state, the one with a count value of 1 holds the boosted-level-one state, etc. Additionally, a valid bit is kept with each register to indicate whether a valid boosted value exists for this register. On a correctly-predicted branch, each boosted value shifts down one level of boosting. Rather than moving the data, each counter is decremented since each counter represents the boosting level of its register. The most complex part of this hardware is to make sure that the non-speculative register is only updated if the boosted-level-one register contains valid data. If it does not, then the non-speculative register counter is not



(a) FULL boosting – multiple shadow locations per register name.



(b) LIMITED boosting – one shadow location per register name.

Figure 4-1: Example schedules for different levels of hardware support.

decremented (it stays at 0) and the boosted-level-one counter (which was at 1) is set to the maximum boosting level (essentially it is decremented twice).

The shadow store buffer consists of a similar hierarchy of renameable buffer locations. In this case though, there exists an ordering among the shadow locations which hold stores operations with the same boosting level, and this ordering complicates the construction of the entire store buffer. To complicate the problem even further, the store buffer has only a limited bandwidth to the data memory system, and this limited bandwidth can easily become a bottleneck at a commit point. For example, when a commit occurs, the shadow store buffer can release a large number of stores to the memory system. Obviously, the memory system cannot handle all of these stores at once, and so they are buffered in a non-speculative store buffer. Since this situation can cause resource problems (e.g. more stores are committed than there is space left in the non-speculative store buffer), our hardware in TORCH checks for store buffer overflow before committing the speculative stores. If overflow would occur, the hardware in TORCH causes a boosted exception. Another problem with a store buffer is that performance suffers greatly if all subsequent loads have to wait for the store buffer to empty before proceeding. To overcome these performance bottlenecks, there are a number of previously-proposed solutions which allow loads to bypass buffered stores and even for loads to be satisfied by a buffered store [Johnson 1990]. For our experiments which include a shadow store buffer, the trace-driven simulation system assumes that the hardware can satisfy a load with a queued value in the store buffer (i.e. loads never stall on memory or the store buffer).

4.2.3 Partial support

Providing full support for the movement of any instruction above multiple conditional branches requires a large increase in the amount of hardware dedicated to the register file and to the store buffer. We can reduce the amount of buffering necessary if we constrain the speculative code motions that our global scheduler is allowed to perform. For example, if we limit the global scheduler to one level of boosting, we only need one copy of the register file and the store buffer. In general, there are two orthogonal methods for reducing the hardware support for boosting: reduce the amount of support for speculative register values, or reduce the amount of support for speculative store values. We have chosen what we feel are three interesting options along this spectrum.

In option 1, we propose to completely remove the shadow store buffer.[3] Without a shadow store buffer, the scheduler cannot label any store instructions as boosted instructions, but since the MIPS architecture is a load/store architecture, the scheduler is still free to label any non-store instructions as boosted instructions. Without boosted stores, the global scheduler cannot “boost” a calculation that involves a store to memory and then a load of that value from memory. Furthermore, the lack of store movement will exacerbate any performance penalties due to imperfect memory disambiguation (i.e. the boosting of loads will be constrained by store instructions whose address we cannot disambiguate). However, if the processor architecture has a large enough number of registers and if the compiler uses an effective register allocator and memory disambiguator, the lack of store boosting should minimally impact the overall performance.

In option 2, we propose to collapse the multiple shadow register files into a single shadow register file that is capable of handling multiple levels of boosting. Without a distinct storage location for each possible level of boosting, `r1.B1` and `r1.B2` in Figure 4-1a refer to

3. As an aside, our TORCH machine model without a shadow store buffer only includes a single-entry store buffer to implement a *pipelined-store* or *delayed-write* scheme [Fu et al. 1987]. This scheme pipelines store operations so that a data memory operation can occur on every cycle.

the same physical storage location, and the compiler must handle this output-like dependence when it schedules the code. If the compiler generates the schedule in Figure 4-1a and thus writes both speculative values to the same shadow location, there is some path (depending upon which speculative value was written last) through the CFG that completes with the wrong value in register `r1`. Figure 4-1b illustrates the choices that the global scheduler has in how it can legally boost instruction `i2`. The global scheduler can limit the boosting of instruction `i2` (`i2′`) or it can rename the destination register of `i2` (`i2″`). In both these cases, the speculative values do not contend for the same physical register. Since Twine always limits the boosting of the instruction under option 2, the performance penalty of this scheduling restriction should be minimal if the output-like dependence occurs infrequently or if the limited overlap of operations is sufficient.

Option 2 significantly reduces the amount of hardware necessary for supporting multiple levels of boosting because it requires only two registers, one counter, and one valid bit for each sequential register name. Figure 4-2 illustrates how the hardware for this scheme is organized (this figure only illustrates functionality and not implementation specifics). The counter maintains the current boosting-level of the value in the shadow register. This counter is decremented each time a branch is correctly-predicted. If the count field is one and the register holds valid data, then on the next correctly-predicted branch the flip-flop is toggled to “pong” the registers—the shadow register becomes the non-speculative register and vice versa. This hardware implements the shadow state and only adds a single gate to the register file access path.

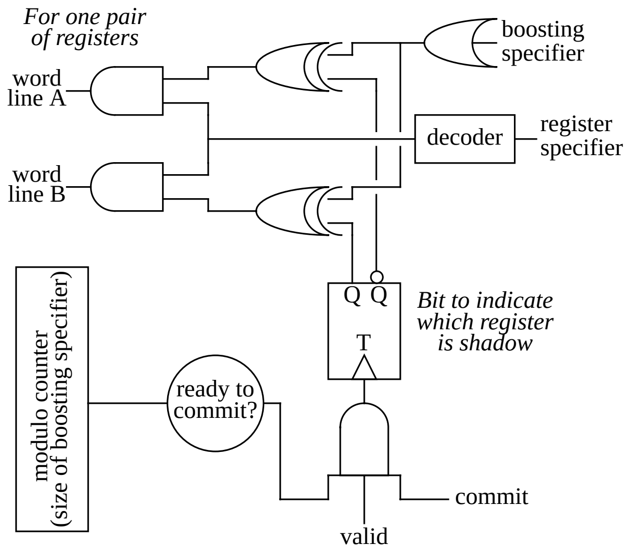
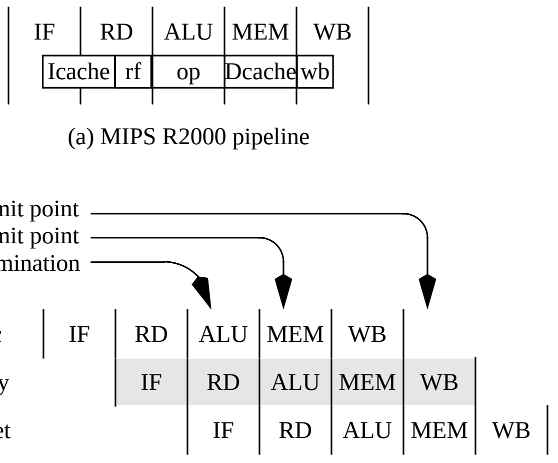

For one pair of registers
word line A
word line B
boosting specifier
decoder
register specifier
Q Q
T
Bit to indicate which register is shadow
modulo counter (size of boosting specifier)
ready to commit?
commit
valid

Figure 4-2: Hardware functionality of the Option 2.

In option 3, we propose to completely remove the shadow register file and shadow store buffer. To still allow for the boosting of operations, we can augment the processor's pipeline control so that the scheduler can "boost" into the "shadow" of the conditional branch. This scheme is basically an extension of squashing branches where instructions are nullified in the cycles following the branch if the branch was incorrectly predicted. With boosting, only the boosted instructions in the cycles following the branch are nullified, not all the instructions in those cycles. Without any extra shadow storage in the machine, the scheduler is constrained to boosting only a few cycles into a branch-ending basic block. For the MIPS R2000 pipeline, the instructions issued with the branch and in the branch delay slot are simple to squash in the pipeline. This scheme requires the least amount of hardware support, but it also imposes the greatest constraints on instruction scheduling.

4.2.4 Exceptions and the commit process

TORCH is a pipelined processor, and the interaction between pipelining and the commit process is complicated by exceptions. To see why, we need to look at the TORCH pipeline in detail. The pipeline in TORCH is identical to the pipeline in the MIPS R2000 processor. It consists of five stages: instruction fetch (IF), register fetch and instruction decode (RD), instruction ALU execution (ALU), data memory load or store (MEM), and register writeback (WB). Figure 4-3a illustrates this pipeline. Figure 4-3b contains a pipeline diagram showing when the condition of the delayed branch is determined and when the commit process occurs.



(a) MIPS R2000 pipeline

(b) Example of when commit phases actually occur

Figure 4-3: Smearing of the commit point onto a MIPS R2000 pipeline.

One might initially think that the commit process should occur immediately after the branch condition is determined, but because of the pipelined nature of our processor, the entire commit process is actually smeared across a number of pipeline stages. The commit process begins in the RD stage of the branch target instruction so that this instruction reads the correct operands when it does its operand fetch. If it were not for exceptions, this pipeline stage would complete the commit process. To correctly handle exceptions though, the commit process should not really occur until the WB stage of the branch delay instruction. At this point in the pipeline, the processor can be sure that an exception did not occur on any instruction in the current basic block (Figure 4-3b assumes that the hardware checks for exceptions at the end of the MEM pipeline stage), and thus before this point, the hardware is unsure whether or not the speculative state is correct. To satisfy the needs of these two pipeline stages, we actually rename the buffer locations during the early commit, and we include some extra hardware to roll back the state if an exception is signalled before the late commit. Remember that our recovery-code scheme for boosted exceptions discards and rebuilds the speculative state on any type of exception, so we do not want to commit the speculative state if an exception occurs between the commit points.

So, an exception between the two commit points requires the processor to undo the early commit. Because our hardware accomplishes the early commit process by only renaming buffer space and never destroying any pieces of data, the hardware can handle the rollback of the late commit point by buffering the commit information for a few cycles (i.e. the hardware remembers which register names performed a commit). For instance, the hardware recognizes a boosted exception at the end of the MEM stage of the delay slot, and it discards the speculative state by first un-renaming the renamed locations and then invalidating the current speculative locations. We have designed the logic that performs this commit buffering and roll-back, and in fact, our design can handle multiple early commits between any early and late commit pair. We found that this more complicated design still only adds a single gate delay to the access time of the register file in option 2. As a final note, the signalling of a boosted exception during WB of the delay slot causes us to not complete the execution of any non-speculative instructions in the delay slot. Consequently, our recovery-code scheme places a copy of these non-speculative instructions at the beginning of the recovery code for that branch.

4.3 Performance evaluation

This section describes the cost and performance of various different TORCH machine models. By collecting cycle-time independent performance numbers and by understanding the complexity of the hardware, we can evaluate the tradeoffs between hardware support for instruction scheduling and performance. This section shows that we can achieve good performance in TORCH with very little hardware support for speculative execution.

4.3.1 Superscalar base model

Our superscalar machine models contain the absolute minimum amount of scheduling hardware required to build a multiple-issue processor out of a single-issue processor. That is, our superscalar machine models do not include any hardware to check for dependences between instructions in a fetch packet; they assume that the global scheduler is responsible for ensuring that instructions fetched together can execute together. Yet, our superscalar machine models do include hardware for interlocks; a packet is stalled in decode if any of its operands are not ready. In this way, our superscalar machine models look a lot like some VLIW machines (e.g. see Fisher [1983]). Unlike some VLIW machines though, all of our superscalar machine models are single-program-counter machines which do not include any type of multi-way branch instruction [Fisher 1980]. Because of this restriction, every superscalar processor in this study is capable of executing at most one CTI per cycle. Furthermore, our *base* superscalar machine models do not include any hardware support for speculative execution (i.e. no boosting hardware). To fairly compare the performance of our superscalar machine models with a MIPS R2000, all of our superscalar machine models have a delayed load and delayed CTI instruction, and their functional units have the same characteristics as the functional units in the MIPS R2000 processor.[4]

The complexity of a superscalar machine model is mainly dependent upon the size of the parallel issue. Some aspects of the superscalar machine design scale linearly with the size of the parallel issue (e.g. register file ports), but some other parts of the design grow super-linearly in the size of the parallel issue (e.g. the bypass logic). If an *i-issue* machine is a superscalar processor that issues *i* instructions in parallel into execution, then the bypass logic of an *i*-issue machine grows as O(i^3) (i.e. O(i^2) for comparisons between the instructions and another O(*i*) for determining priority). To limit the cost and complexity of our

4. For the few floating-point operations in our benchmarks, our scalar and superscalar simulators use a simplified model of the MIPS R2010 floating-point pipeline where the floating-point functional units do not share any resources.

superscalar hardware, we only investigate machine models which issue a small number of instructions per cycle.

For our studies, a *full-issue* machine model is a superscalar machine model with no restrictions on its parallel issue. Each issue slot in a full-issue machine is supported by a full compliment of functional units and processing resources. This superscalar machine model duplicates the instruction fetch logic, the decode logic, the register fetch logic, the execute logic, the data memory logic, and the write-back logic of the MIPS R2000 processor for each parallel issue slot. Consequently, any number of the same (non-CTIs) instructions may issue and execute together.

Even with the limitation of a single CTI per cycle though, a full-issue model has undesirable properties. A full-issue machine is expensive to build because some resources, like data memory ports, are expensive to duplicate. Furthermore, a full-issue machine is inefficient in its use of the available hardware resources because the duplication of some resources, like the shifter, do not noticeably increase performance. A more cost-effective solution distributes the functional units among the parallel issue slots and duplicates only those inexpensive functional units which noticeably increase performance.

A *limited-issue* machine model is a superscalar machine model with restrictions on its parallel issue. Each issue slot in a limited-issue machine is supported by some subset of the functional units. This machine also assumes that an instruction that is fetched in a particular issue slot can execute in that slot. The compiler is responsible for scheduling instructions in the correct issue locations; there is no swap logic as there is in the DEC 21064 [DEC 1992].

The only limited-issue model used in this thesis is for a two-issue superscalar machine. This limited-issue model is organized such that one side of the two-issue machine contains an integer ALU, a branch unit, a shifter, an integer multiply/divide unit, and a floating-point unit; the other side contains just an integer ALU and a memory port. This limited-issue model is inexpensive because it only duplicates the integer ALU. All other operations execute only on one side of the machine, and thus the parallel execution of some pairs of instruction classes is restricted. For example, this limited-issue machine cannot execute a branch and a shift operation in parallel. We chose the position of functional units in this limited-issue machine by the frequency of issue pairs in a full-issue machine model so that those pairs of instruction classes which cannot execute in parallel were those classes with the lowest probability of parallel issue.

Table 4-2 presents the cycle-count speedups (over the MIPS R2000 processor) for a variety of base superscalar machine models. The table references three different superscalar processors: a 2-issue, limited-issue model (2i.L); a 2-issue, full-issue model (2i.F); and a 4-issue, full-issue model (4i.F). Twine schedules each application for the three superscalar processors. The first time that Twine schedules the applications, it performs post-pass, basic block scheduling (scheduling after register allocation with no global code motions). For the other two times, Twine uses the global instruction scheduling algorithm to move instructions past basic block boundaries, but Twine only permits safe and legal speculative executions to occur since the base superscalar machine model does not contain hardware to support boosting. The difference between the global scheduling cases is whether the compiler performs register allocation or not. In the post-pass scheme, register allocation is done before global scheduling. In the infinite register scheme, register allocation is not done for the assembly language input to the global scheduler (though it is done for the assembly language input to the MIPS assembler). Since the intermediate form of our compiler is an infinite register model, the performance of this machine model is as if the superscalar processor had an infinite number of registers (i.e. storage conflicts between and among temporary variables and different user variables are eliminated).

	Post-pass Basic Block Scheduling			Post-pass Global Scheduling			Global Scheduling with Infinite Registers		
	2i.L	2i.F	4i.F	2i.L	2i.F	4i.F	2i.L	2i.F	4i.F
awk	1.13	1.16	1.18	1.16	1.19	1.21	1.24	1.26	1.26
compress	1.15	1.19	1.21	1.33	1.39	1.44	1.40	1.45	1.46
eqntott	1.23	1.35	1.38	1.25	1.38	1.43	1.43	1.54	1.61
espresso	1.16	1.21	1.25	1.22	1.29	1.33	1.38	1.44	1.51
grep	1.07	1.09	1.09	1.30	1.35	1.36	1.36	1.37	1.37
nroff	1.10	1.13	1.14	1.23	1.28	1.30	1.30	1.32	1.33
xlisp	1.14	1.21	1.26	1.20	1.29	1.34	1.31	1.35	1.35
h. mean	1.14	1.19	1.21	1.24	1.31	1.34	1.34	1.38	1.40

Table 4-2: Superscalar speedups achievable without hardware support for boosting.

Basic block scheduling achieves only a small cycle-count speedup across all of the machine models in Table 4-2. This result supports our statement in Chapter 1 that non-numerical applications contain very little ILP within a basic block. Our global instruction scheduler looks beyond these basic block boundaries for ILP and thus produces better cycle-count speedups. The 2i.L model in Table 4-2 is the simplest and cheapest of our superscalar machine models without boosting, and it achieves a a harmonic mean speedup of 1.24 under post-pass global scheduling. Our hardware designs show that we can implement a 2i.L machine model with a cycle time that is probably identical to the cycle time of the MIPS R2000 processor, and thus the cycle-count speedup of 2i.L is a good indication of the true performance of that superscalar processor over the R2000 processor. Since we are interested in a design approach that improves processor performance without adversely affecting the processor cycle time, we use this 2i.L machine model in the next subsection as the foundation for our superscalar simulations with boosting.

4.3.2 Superscalar models with boosting

Section 4.2.3 described two orthogonal approaches for increasing the amount of hardware support for boosting: increase the complexity of the shadow register file buffering, or increase the complexity of the shadow store buffering. In this subsection, we use some combinations of the options discussed in Section 4.2.3 to build five augmented machine models. We call these augmented models: *Squashing*, *MinBoost1*, *Boost1*, *MinBoost3*, and *Boost7*. Table 4-3 presents the percentage improvement in the cycle-time speedup of these augmented superscalar processors over the cycle-count speedup of the 2i.L machine model. The numbers in this table correspond to scheduling with register allocation (i.e. post-pass scheduling).

	Squashing	MinBoost1	Boost1	MinBoost3	Boost7
awk	10.7%	14.1%	15.0%	16.4%	16.9%
compress	6.1%	8.8%	8.8%	10.6%	10.6%
eqntott	6.8%	12.8%	14.1%	14.7%	15.3%
espresso	9.5%	17.7%	17.7%	21.0%	22.1%
grep	15.6%	27.7%	27.7%	41.5%	41.5%
nroff	11.2%	23.8%	24.0%	30.7%	35.9%
xlisp	6.9%	13.0%	13.3%	12.4%	13.3%
h. mean	8.7%	14.8%	15.2%	16.4%	17.1%

Table 4-3: Improvements in cycle-count speedup for various degrees of boosting support over post-pass global scheduling (machine model 2i.L).

Basically, the augmented machine models include an increasing amount of hardware support for boosting as we read across Table 4-3 from left to right. The *Squashing* model contains no shadow structures and boosting is only supported by a squashing pipeline. This scheme adds the smallest amount of hardware possible for a scheme that supports boosting. With this limited boosting ability, Twine is constrained to only label instructions as boosted if they fall into the delay branch cycle of a branch-ending basic block, as discussed in Option 3 of Section 4.2.3. The Squashing model achieves less than a 10% improvement in performance over global scheduling without boosting.

The *MinBoost1* model adds only a small amount of buffering to the processor. It contains a single shadow register file that can only support a single level of boosting, and it does not contain any type of shadow store buffer. This schemes basically doubles the size of the register file, and it supports the general speculative movement of any non-store instruction over a single conditional branch. Referring to the hardware in Figure 4-2, the shadow register file hardware for the MinBoost1 model does not contain any counters, and the gate which clocks the T flip-flop is simply an AND of *valid* and *commit*. Since the boosting specifier is only one bit, the OR-gate of Figure 4-2 is also not necessary. The MinBoost1 model achieves nearly a 15% improvement in performance over global scheduling without boosting.

The *Boost1* model is the same as MinBoost1 except that it supports the boosting of stores. That is, Boost1 includes both a shadow register file and a shadow store buffer, and both of these structures only support a single level of boosting. The Boost1 model achieves slightly better than a 15% improvement in performance over global scheduling without boosting.

The *MinBoost3* model is like MinBoost1 in that it does not contain any type of shadow store buffer. Unlike MinBoost1 though, MinBoost3 contains a single shadow register file that supports boosting with dependence information for three conditional branches. This scheme adds the smallest amount of hardware possible for a scheme that supports boosting over a large number of branches. For this scheme, the scheduler is constrained as discussed in Options 1 and 2 of Section 4.2.3. The MinBoost3 model achieves more than a 16% improvement in performance over global scheduling without boosting.

The *Boost7* model approximates an upper limit on the performance gain available from boosting with our global scheduler. The Boost7 model contains all the hardware necessary to support boosting with dependence information for seven conditional branches, and because the vast majority (over 99%) of our static traces contain fewer than eight conditional branches, Boost7 basically provides the global scheduler with an unrestricted model of speculative execution. This “unrestricted” model of speculative execution achieves about a 17% improvement in performance over global scheduling without boosting.

Overall, our “unrestricted” model of speculative execution (Boost7) contains an amount of hardware support for boosting that is obviously unreasonable, and Table 4-3 shows that this amount of extra hardware does little to improve performance over the other boosting models with much less hardware support. Table 4-3 also shows that the boosting of store instructions provides only a small increment in performance; Boost1 improves the harmonic mean performance of MinBoost1 by less than 0.5%. The decrease in the cycle count obtained with the shadow store buffer (even a simple one) does not seem to justify the cost and complexity of that shadow store buffer. The best schemes for cost-effective performance seem to be MinBoost1 and MinBoost3. Both schemes are basically advocating a duplicated register file.

As Section 4.2.3 discussed, the addition of a single shadow register file causes the register file access time to be approximately one gate delay longer than the access time of the simple scalar register file. Since the register file is not currently in the critical path of the our TORCH implementation (determined by looking at critical paths in the gate-level simulator), we do not expect the complexity of the shadow register file to increase the cycle time of the TORCH processor without boosting. Also, the additional hardware required by the more complex register file is not large. The decoder for a MinBoost1 machine with 32 sequential registers contains only 33% more transistors than a normal decoder for a register file with 64 registers (50% more transistors are required for a MinBoost3 implementation).

The MinBoost1 and MinBoost3 schemes double the size of the register file and then set aside this extra register space to handle unsafe and illegal speculative execution. An interesting question to ask is whether the MinBoost1 or MinBoost3 schemes actually do better than a global scheduling scheme which uses software register renaming with a 64-entry register file. Though the SUIF register allocator and Twine cannot directly generate code for this enhanced machine model, we can place an upper bound on the performance of a global scheduler with software register renaming by checking the performance achieved by the global scheduling scheme with an infinite register model (see Table 4-2). Global scheduling with infinite registers (no boosting) achieves an 8.1% performance improvement (harmonic mean for machine model 2i.L) over the post-pass global scheduling scheme. This is a smaller improvement than that achieved by MinBoost1 and MinBoost3. Thus hardware support for unsafe speculative code motions improves machine performance beyond the best performance achievable with a pure software scheme.

Table 4-4 summarizes the hardware requirements and the cycle-count speedups (speedup over the R2000 processor) for each of the schemes with some support for speculative execution. In addition to the 2i.L machine model, Table 4-4 also contains cycle-count speedups for the 4i.F machine model. These numbers reinforce the conclusion that small amounts of boosting hardware capture the majority of the cycle-count benefit.

		Global Scheduling	Squashing	MinBoost1	Boost1	MinBoost3	Boost7
Extra boosting hardware	shadow register files	—	—	1	1	1+	7
	shadow store buffers	—	—	—	1	—	7
H. mean speedup	model 2i.L	1.24	1.36	1.45	1.45	1.49	1.50
	model 4i.F	1.34	1.53	1.57	1.59	1.65	1.68

Table 4-4: Summary of hardware requirements and cycle-count speedups for the different speculative execution schemes.

4.3.3 Comparison with a dynamically-scheduled model

To put the results of the last subsection into perspective, Table 4-5 compares the speedup of the 2i.L model under MinBoost3 to the speedup of a dynamically-scheduled, out-of-order-issue superscalar processor with support for speculative execution (all speedups are relative to the MIPS R2000 processor). The speedup numbers for the dynamically-scheduled machine were gathered from the trace-driven simulator built by Johnson [1990]. The dynamically-scheduled machine is functionally equivalent to the 2i.L, MinBoost3 machine model. Like a 2i.L machine, the dynamically-scheduled machine fetches and decodes two instructions per cycle. To implement out-of-order execution with speculative execution, the dynamically-scheduled machine uses a total of 30 reservation station locations [Tomasulo 1967] and a 16-entry reorder buffer [JESmith and Pleszkun 1985]. These are enough locations to guarantee that the machine never stalls waiting for a reservation station or reorder buffer location. To predict branches, the dynamically-scheduled machine uses a 2048-entry, 4-way set associative branch target buffer. This structure is aggressive enough to achieve a branch prediction accuracy that is slightly better than that achieved by our static profiling. Finally, the dynamically-scheduled machine has the same number of functional units as a 2i.L machine, but since the dynamically-scheduled machine uses reservation stations, it can issue up to 6 instructions per cycle. For the statically-scheduled machine model, Table 4-5 first presents the speedup numbers for post-pass MinBoost3 scheduling (i.e. global scheduling with boosting after register allocation); Table 4-5 then presents the numbers for MinBoost3 scheduling with an infinite register model. Simiarly for the dynamically-scheduled machine model, Table 4-5 first presents the speedup numbers without any hardware register renaming (i.e. dynamic scheduling constrained by the register allocator); Table 4-5 then presents the numbers with perfect hardware register renaming.

	Statically-scheduled 2i.L Model		Dynamically-Scheduled “2i.L” Model	
	Post-pass MinBoost3	MinBoost3 with Infinite Registers	No Register Renaming	Perfect Register Renaming
awk	1.36	1.45	1.51	1.59
compress	1.45	1.55	1.60	1.66
eqntott	1.43	1.63	1.37	1.52
espresso	1.48	1.62	1.57	1.66
grep	1.84	1.96	1.67	1.68
nroff	1.61	1.72	1.57	1.67
xlisp	1.35	1.52	1.42	1.56
h. mean	1.49	1.62	1.52	1.62

Table 4-5: Cycle-count speedup comparison of MinBoost3 with a dynamic scheduler.

4.4 Summary

Boosting relies on hardware buffers and some control logic to ensure the safe and legal speculative execution of any instruction. In this chapter, we looked at three aspects of the hardware support required for boosting: the organization of the hardware support, the benefit of the hardware support on the cycle count of an application, and the impact of the hardware support on the cycle time of a processor. For our load/store architecture, boosting requires hardware support in two distinct regions of the processor. Boosting requires buffering in the register file to postpone the effects of the boosted register operations, and it requires buffering in the data memory subsystem to postpone the effects of the boosted store operations.

Through a trace-driven simulation of our superscalar machine model, we found that boosting can decrease the total cycle count of our benchmark applications beyond the cycle-count performance that we could achieve in the machine model without boosting. Though the performance benefit from boosting was greater in the parallel-resource-rich machine models, we did achieve noticeable improvements in those machine models with a very limited number of parallel resources. Furthermore, by varying the amount of hardware support for boosting in each machine model, we found that only a small amount of hardware support is necessary to approach the performance of a machine model with “unrestricted” speculative execution.

For the boosting configurations that only added speculative buffering in the register file, we found that we could create a shadow location for every register location in the machine and only change the access time of the register file by one logical gate delay. With this single shadow register file, we described a design by which the compiler was still able to label speculative instructions with dependence information for multiple conditional branches. The combination of these two factors provided us with a design in which the compiler is able to look across multiple conditional branches for a greater amount of ILP and in which the cycle time of the processor is negligibly impacted by the hardware support for boosting. As a result, we believe that the 1.49x cycle-count speedup of the 2i.L, MinBoost3 machine model over the MIPS R2000 processor is an accurate indicator of the overall improvement in CPU performance gained from boosting.

Chapter 5

Conclusion

The commercial acceptance of superscalar and superpipelined microarchitectures has created a strong need for instruction schedulers to extract and exploit a larger amount of the ILP within an application. In the domain of non-numerical applications, this need for more ILP has indicated the need for instruction schedulers which effectively support speculative execution. For this thesis, we followed an integrated design approach so that we could determine a distribution of instruction-scheduling functionality between the hardware and the compiler that best decreases the average CPI of a non-numerical application while negligibly increasing the instruction count of that application and the cycle time of the processor. The result of our research is a design framework called opportunistic instruction scheduling, and under this framework, we developed an architectural mechanism for hardware-assisted speculative execution called boosting which noticeably increases the performance of our superscalar CPU.

Opportunistic instruction scheduling provides the compiler with an effective method of speculating on a run-time event. Under opportunistic instruction scheduling, the compiler informs the hardware about its speculative decisions so that the hardware can check the appropriate run-time variables and correctly update the program state. The advantages of this cooperative approach are twofold. First, we can use sophisticated compiler techniques to efficiently uncover and schedule for ILP, and since the compiler is no longer pessimistic in its scheduling decisions, the compiler produces better schedules. Second, because the compiler does all of the instruction analysis and scheduling, the hardware in our superscalar machine remains simple and fast. The hardware never weighs options to determine what instructions to execute next; it just executes the schedule that the compiler creates.

Specifically, this thesis described and investigated boosting, an opportunistic instruction scheduling mechanism for branch speculation. Boosting provides the compiler with the ability to specify the speculative execution of any instruction, and boosting relies on the hardware to ensure that a speculative value does not corrupt the machine state if the compiler-specified speculation is incorrect. To accurately evaluate boosting, we developed a new global instruction scheduling algorithm which is tailored for the non-numerical application domain. This algorithm provides one base line for how effectively a compiler can globally schedule code without boosting. Since boosting only augments the capabilities of the global scheduling algorithm and does not change its structure, the algorithm provides an effective platform for evaluating a range of hardware support for boosting. In this thesis, we also developed a simulation system to collect cycle-count numbers for a variety of superscalar machine models, and we analyzed the hardware complexity of these models to accurately determine their overall performance.

Table 5-1 summarizes the major findings of our evaluation. The results of the evaluation are encouraging for they show that boosting can improve the performance of a simple superscalar processor. Furthermore, our results showed that a small amount of hardware support for boosting captures almost all of the improvement achievable with unrestricted speculative execution. In fact, a statically-scheduled superscalar processor using boosting achieves a cycle-count performance rating which is nearly equal to that achieved with a dynamically-scheduled, out-of-order-issue superscalar processor that supports speculative execution, and boosting does so without the hardware overhead of dynamic scheduling.

	Cycle-count speedup	Cycle-time impact
Global scheduling without boosting	1.24	none
Global scheduling with MinBoost3	1.49	none
Dynamic scheduling with speculative execution	1.52	some

Table 5-1: Summary of performance evaluation (machine model 2i.L).

5.1 Areas of future research

Rather than viewing this thesis as an end point to an area of research, we believe that the work in this thesis provides a foundation for further research into tradeoffs between functionality in the hardware and functionality in the software. There are many experiments and many more architectural investigations that we could explore with the instruction scheduler and the simulation system developed in this thesis. This section focuses on just two specific areas of future research that try to uncover even more of the exploitable ILP in non-numerical applications (thus further improve the performance of our superscalar processor).

The first area deals with the analysis and optimization of the existing system and techniques. Given a working compiler for a superscalar processor, we can now analyze the generated code to discover where more compile-time effort would produce noticeably better schedules. Chapters 3 and 4 mention a number of possible scheduling optimizations (e.g. a more sophisticated priority function) and a number of ILP-increasing optimizations (e.g. procedure inlining) that could lead to better schedules. Analysis of the current superscalar code would help us determine which of these optimizations are necessary and which are most beneficial.

In fact, we believe that the cycle-count speedups in Table 4-5 are a lower bound on the achievable cycle-count speedups since a number of avenues exist for decreasing the cycle counts of either scheduling approach. For instance, one could improve the cycle-count speedup of the dynamically-scheduled machine by using our global scheduling algorithm to preschedule the code. Prescheduling improves the utilization of the processor's resources and minimizes stalls. Alternatively, we could modify the pipeline of the base processor to remove the branch or load delay slots that lengthen the instruction schedules and consume some of the available ILP. For example, the designers of the SUN SuperSparc arranged their pipeline so that the typical load-to-use delay is replaced with a less-frequent load-to-load-address delay [Case 1991]. We could further improve the static schedules produced for either approach by using ILP-increasing optimizations, such as loop unrolling, to carefully uncover more ILP. We have performed some preliminary experiments with a loop unroller which unrolls all the loops in a program module. Though cycle counts for the MinBoost3 model did decrease slightly, the improvement was well below what we expected. Upon closer inspection of the code, we discovered that no one problem is limiting the performance. What is required is an effective mix of solutions to problems in many areas (e.g. loop-level optimizations, procedure inlining, and better memory disambiguation).

Furthermore, opportunistic instruction scheduling mechanisms such as boosting open up entirely new opportunities for peephole optimization. For example, the unconditional jump at the end of a THEN block in an IF-THEN-ELSE construct can become unnecessary if the global scheduler is able to boost all of the code in the THEN block. A peephole optimizer for boosting should remove the unconditional jump, invert the condition of the conditional branch, change the target of the conditional branch so that it branches over the ELSE, and predict the conditional branch to be taken. Figure 5-1 contains another example where a peephole optimizer for boosting could reduce the instruction count of the application. The original code is swapping the values in registers `r1` and `r2`. Simple sequential RISC code requires a temporary register (`r9`) to perform this swap. Direct translation of this sequential code into speculative code (assuming `r1` and `r2` are live on the non-predicted edge of the conditional branch) is shown in Figure 5-1b. Since `r1.B` and `r2.B` are temporary registers that are different from `r1` and `r2`, the temporary register `r9` is not needed as shown in Figure 5-1c. Of course, a superscalar processor with anti-dependence length of zero cycles can interchange `r1` and `r2` by scheduling "`r1 = r2;`" and "`r2 = r1;`" in the same cycle. Boosting provides the global scheduler with more freedom since the global scheduler does not have to schedule the two instructions in Figure 5-1c in the same cycle.

```
    r9 = r1;              r9 = r1;             r1.B = r2;
    r1 = r2;            r1.B = r2;             r2.B = r1;
    r2 = r9;            r2.B = r9;

(a) Original code      (b) Boosted code      (c) Optimized code
```

Figure 5-1: Example of a peephole optimization on boosted code.

Finally, the current simulation system focuses on the performance of the CPU, and it assumes a perfect memory system. This simulation system therefore provides a first-order approximation of the performance of the real superscalar processor. The inclusion of a real memory model in the simulation system would provide accurate answers on how the global scheduler and boosting affect the performance of the memory system.

The inclusion of a real memory model also provides an excellent starting point for the second area of future research, i.e. investigating other architectural techniques for improving the performance of non-numerical applications on superscalar processors. For instance, a memory model is required for the evaluation of speculative memory disambiguation (Section 2.3). Furthermore, the memory system is the next big performance bottleneck in superscalar processors. The current superscalar machines contain only a single data memory port that stalls the machine on a cache miss. Research is needed to determine the cheapest and most effective ways of providing multiple memory pipes and of tolerating cache misses. Looking even further, the next big performance bottleneck is a result of the single CTI per cycle limitation. Are multi-way branches cheap and effective, or should superscalar architectures include some sort of guarding mechanism to remove a number of the conditional branch instructions?

Eventually, this type of research needs to evaluate how the fine-grain parallelism techniques for superscalar processors interact with the coarse-grain parallelism techniques found in parallelizing compilers for multiprocessors. This information will help determine when we should stop building a large-issue superscalar processor and start building a multiprocessor consisting of small-issue superscalar processors.

For all of these further investigations, the best approach for cost-effective performance comes from a design approach that considers the needs and capabilities of the hardware and the software together. In this thesis, we have demonstrated how this design philosophy is applicable to the area of instruction scheduling for high-performance processors.

Appendix A

Scheduling a Basic Block

This appendix describes the structure of the basic block scheduling algorithm that lies at the heart of our global scheduling algorithm. It also describes some of the important implementation issues involved in the scheduling of instructions for our target microarchitecture.

As outlined in Figure 3-5 on page 49, our global scheduler schedules the basic blocks in each trace in a top-down ordering, i.e. it schedules the basic block at the head of the trace before it schedules any basic block later in the trace. The basic block scheduling algorithm is a list scheduling algorithm which uses a combination of top-down and bottom-up scheduling techniques to cycle schedule a basic block. The actual instruction scheduling is done by a top-down scheduler which attempts to pull instructions as early in the basic block as possible. Top-down scheduling is convenient in this context because a top-down scheduling approach maps well onto our top-down approach to the scheduling of the basic blocks in the trace. Recall that the top-down approach to the scheduling of the trace was done because our global transformations only move code upward in the CFG, and that speculative techniques such as boosting require this upward code motion.

Yet, our microarchitecture contains delayed CTIs, and top-down scheduling makes it difficult to know when the algorithm may schedule a delayed CTI. Since bottom-up scheduling pulls instructions as far down the schedule as possible, bottom-up scheduling is good for filling the CTI delay slots. Thus, to effectively handle this aspect of our microarchitecture, our basic block scheduling algorithm first uses a bottom-up *prescheduler* to preschedule the CTI delay slots, if any. Once the delay slots are scheduled, the prescheduler marks the instructions in the prescheduled delay slots and undoes the bottom-up scheduling. The priority on these marked instructions is decreased slightly so that a list scheduler has a higher probability of scheduling the non-marked instructions first. Now when the top-down scheduler runs, it knows that a CTI is available for scheduling if all non-marked instructions have been scheduled (i.e. it has predetermined what instructions are capable of scheduling within the CTI delay). Figure A-1 outlines the algorithm which schedules the basic block at the head of the trace. The next few sections describe some specifics of this basic block scheduling algorithm.

```
if (basic block ends in delayed CTI) {
   preschedule CTI delay slots in bottom-up manner;
   lower priority of prescheduled instructions;
   undo prescheduling;
}

while (exists unscheduled instruction in basic block) {
   top-down list schedule next cycle;
   issue scheduled instructions;
   update EDAG;
}

if (BB ends in delayed CTI && CTI not fully delayed) {
   for (i = 0; i < no_of_unfilled_cycles; i++) {
      top-down list schedule next cycle;
      issue scheduled instructions;
      update EDAG;
   }
}
```

Figure A-1: Algorithm for scheduling the first basic block in the trace.

A.1 Instruction lists

Our basic block scheduling algorithm relies on three sets of instruction lists. The top-down scheduler uses two of the sets, and the bottom-up prescheduler uses the third set. Each set consists of a *ready* list, a *waiting* list, and an *anti-dependent* list. An instruction belongs to a ready list if all of its dependent predecessor[1] instructions in the EDAG are scheduled and their latencies fulfilled, i.e. a ready list contains only ready instructions. To simplify the list scheduling routines, the scheduler orders the instructions in each ready list by decreasing priority.

An instruction belongs in a waiting list if all of its dependent predecessor instructions in the EDAG are scheduled, but not all of their latencies are fulfilled. If the scheduler schedules a waiting instruction in the current cycle, the hardware must stall the instruction's execution for the number of cycles that are required to generate the waiting instruction's

1. The descriptions in this appendix are written for a top-down list scheduler. The simple translation for a bottom-up list scheduler is left to the reader.

operands. In other words, the scheduling of waiting instructions requires hardware interlocks. The scheduler orders the instructions in each waiting list by increasing interlock time.

An instruction belongs to the anti-dependent list if all of its preceding true and output dependent instructions in the EDAG are scheduled and their latencies fulfilled, and all of its preceding anti-dependent instructions in the EDAG are either scheduled or in the ready list. The instructions in the anti-dependent list are not yet ready to be scheduled in the current cycle, but they will be available for scheduling in the current cycle if all of their unscheduled anti-dependent partners are scheduled in the current cycle. The scheduler does not bother to order the instructions in the anti-dependent lists.

The global scheduler initializes the set of bottom-up lists whenever it invokes the prescheduler. The global scheduler initializes both sets of top-down lists during the creation of the prioritized EDAG. One set of the top-down lists holds instructions for the current basic block, while the other set of top-down lists holds the instructions for all of the later basic blocks in the trace. The separation of these sets manages the priority distinction between native instructions and non-native instructions. For both the top-down and bottom-up lists, the scheduler updates each instruction list during the scheduling of a cycle (e.g. to mark scheduled instructions, to simulate hardware interlocks, and to move instructions from the anti-dependent list to the ready list), and it updates each instruction list after the scheduling of a cycle (e.g. to remove scheduled instructions and to insert newly-ready instructions). Finally, the scheduler redistributes the instructions between the native and non-native sets when it completes the scheduling of a basic block (to prepare for the scheduling of the next, newly current, basic block).

A.2 Scheduling a packet

The scheduling of a cycle consists of matching the available resources with a set of the ready instructions. For this purpose, the basic block scheduling algorithm maintains a resource reservation table which describes the resources available at each cycle. From a compiler's point of view, this table represents a *packet* of instruction slots which it must fill with ready instructions. The filling of this packet is constrained both by the execution resources available in the machine (e.g. only one memory port so only one load or store per cycle) and by the issue resources available in the machine (e.g. the processor is asymmetric so instruction placement is constrained). Figure A-2 lists the steps required in the cycle scheduling of an instruction packet. This figure represents the core set of steps in both the top-down scheduler and the bottom-up prescheduler. The top-down scheduler though invokes this packet scheduling algorithm twice: first for the lists containing the native instructions, and second for the lists containing the non-native instructions. This ordering is how we ensure that native instructions always have priority over non-native instructions.

```
if (empty READY list && hardware interlocks available) {
   increase cycle count by smallest waiting time;
   update READY list with instructions from WAITING list;
}

/* fill packet with cycle ready instructions */
while (unchecked instructions in READY list
       && empty instruction slots in packet) {
   ip = highest priority, unchecked instruction;
   if (ip available via global code motion) {
      if (resource available for ip) {
         pick issue slot and reserve resources;
         mark ip as scheduled;
         if (newly ready instruction in ANTI list)
            update ANTI and READY lists;
      } else if (busy resource non-pipelined)
         move ip from READY to WAITING list;
   }
}
```

Figure A-2: Basic algorithm for cycle scheduling an instruction packet.

The first step in the scheduling of a packet is to make sure that the ready list contains at least one instruction. If the ready list is empty but there are unscheduled instructions in the current basic block, the waiting list must contain instructions from the current basic block that are simply waiting for their operands to become available. By increasing the cycle count by the smallest waiting time in the waiting list and by appropriately updating the ready and waiting lists, the scheduler can mimic a hardware interlock. If the hardware does not support a full set of interlocks, the scheduler fills out the schedule with the appropriate number of NOPs. By only forcing hardware interlocks when the ready set is empty at the beginning of the scheduling of a cycle, this approach ensures that the schedule is never extended unnecessarily by a hardware interlock.

The next step scans the non-empty ready list looking for instructions to schedule in the current cycle. This scan continues as long as there are unchecked instructions and unreserved resources in the current instruction packet. The scheduler pulls instructions from the ready list in the order of highest priority to lowest priority. As each ready instruction is selected, the scheduler checks the earliest availability of this instruction to determine if the instruction is actually available for scheduling at this point. As we mentioned in Chapter 3, our global scheduler summarizes availability so this check is easy and compile-time efficient (e.g. check two integer values for equality). For certain special instructions (e.g. CTIs, non-interlockable instructions, special system instructions, etc.), the scheduler performs other specific checks to determine if the instruction is really ready. If the instruction passes the ready and available checks, the scheduler then checks the resource reservation table to determine if current packet contains enough free resources to execute the instruction. If the required resources are free, the scheduler picks an issue slot for the instruction, marks the required resources as busy, and records the instruction as scheduled. If the instruction is not available or all required resources are not free, the instruction is left in the ready list and the next instruction in priority order is checked. If the instruction is available but the required resource is currently busy and non-pipelined, the instruction is moved from the ready list to the waiting list; thus, the waiting list is used for both data and resource interlocks.

Whenever the scheduler schedules another ready instruction, it checks the anti-dependence list to determine if the scheduling of this instruction permits the scheduling of any anti-dependent partner instructions. If an anti-dependent instruction becomes ready because of the scheduling of a ready instruction, the scheduler updates the ready and anti-dependent lists. By updating the ready list in the middle of the scan, the scheduler might need to adjust its scan data structures so that it scans the newly inserted instructions at the appropriate time (i.e. insert in priority order).

Delayed CTIs add a number of interesting twists to the basic packet scheduling algorithm. During top-down scheduling of a packet, a data-ready CTI encounters an additional check to determine whether the CTI is really ready for scheduling. The check examines the count of unscheduled, unmarked instructions in the current basic block. If this count is zero, then the CTI is really a candidate for scheduling. If the count is greater than zero, the CTI remains in the ready list. To guarantee that the CTI is scheduled in the earliest cycle possible, the CTI acts like it has an anti-dependent constraint on the last unmarked instruction. In other words, as soon as the last unmarked instruction is scheduled, the CTI is considered for scheduling in the current packet.

Delayed CTIs also complicate the enforcement of the requirement that the scheduler not move any instructions down out of their original basic block. Even though all unmarked instructions are scheduled, the operands of a marked instructions may still be unavailable, and because of global code motions, this marked instruction might get pushed out of the CTI delay slots. The reason that this happens is that an interlock only occurs if absolutely no instructions are ready. For top-down scheduling outside of the CTI delay slots, an interlock occurs if both the current basic block ready list and the later basic block ready lists are empty. To avoid downward code motion, the top-down scheduler should force an interlock if it is scheduling a CTI delay slot and the current basic block ready list is empty.

The immediate reservation of resources is a simple approach to mapping scheduled instructions onto the machine, but it is also an inefficient approach for microarchitectures that are asymmetric. For instance, if a machine can execute either an ALU or memory operation from the first issue slot in the packet but can only execute an ALU in the second issue slot, a non-backtracking scheduler can produce a poor schedule if it searches and assigns resources in a simple first issue slot to last issue slot manner. Under this simple scheme, the scheduling of a ready list containing a high-priority ALU operation and a low-priority load operation results in a two cycle schedule, even though both instructions can issue in the same cycle (by placing the load in the first issue slot). To avoid this problem without backtracking, the actual picking of an issue slot and the reserving of resources is postponed for a scheduled instruction if that instruction has multiple choices among the available resources. The actual algorithm maintains an indication of the number of multiple resources reserved, but it postpones the mapping of a scheduled instruction to a machine resource until a specific choice is forced or the packet schedule is complete. A choice is forced when there is only one available issue slot or resource for the scheduled instruction. Of course, the forced mapping of one instruction might cause earlier postponed instructions to map.

A general algorithm for this delayed-mapping approach is difficult. The basic problem is that overlapping resource classes can cause a cycle in a simple reservation checking mechanism. Consequently, the algorithm that we implement only handles resource classes that are perfectly nested or totally disjoint with respect to the instruction issue locations. For example, suppose we are given a microarchitecture which issues five instructions per cycle and which contains three ALUs and two memory ports. If any issue location in this machine can accept both an ALU and a memory operation, then the other memory port must also be grouped with an ALU (perfect nesting). The other choice (totally disjoint resource sets) has the two memory operations in the two issue slots that cannot accept an ALU operation. This restriction is relatively minor from a machine microarchitecture point of view, especially since we focus on machine models with few parallel resources and small issue widths.

A.3 Overlapping branch delays

The pipelining of CTIs improves performance, and the execution of a CTI in another CTI's delay slot is fairly simple to support in hardware [Chow 1989]. Though our global scheduling algorithm does not reorder CTIs (to minimize the potential of code explosion), the global scheduling algorithm can support the scheduling of a CTI in another CTI's delay slot. The cost of this performance feature is a small increase in the complexity of the global scheduler. That is, the scheduling of a CTI in another CTI's delay slot involves a global code motion, and this global motion might result in the duplication of the second CTI. Figure A-3 shows an example where the delay slots of the copies of the conditional branch instruction `cBra3` are no longer adjacent to each copy of the instruction. This separation is not a problem for the hardware, but the scheduler must remember to schedule this duplicate CTI with fewer delay slots because some of the delay slots are already scheduled elsewhere.

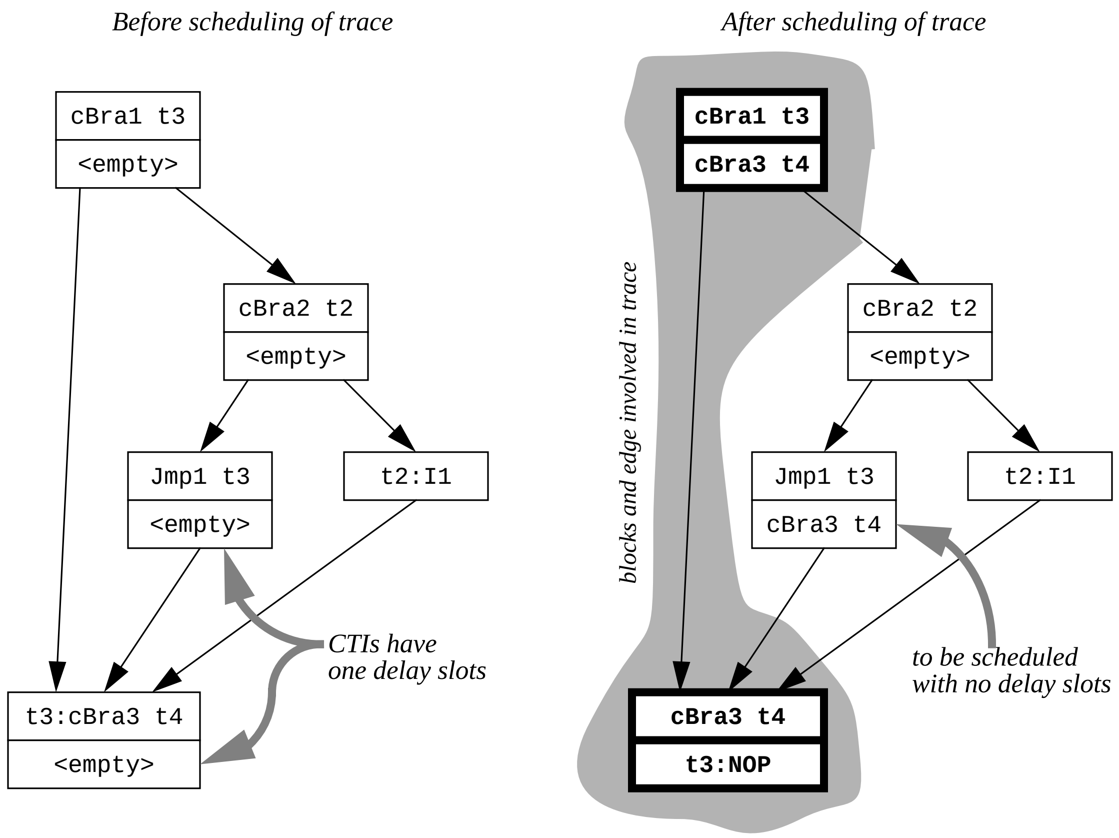


Figure A-3: Example of overlapped CTI scheduling.

Appendix B

Secondary Issues

The major conclusion of this thesis implies that static scheduling with boosting is an acceptable solution to the problem of instruction scheduling in superscalar processors. Yet, this thesis primarily focuses on how static scheduling with boosting affects the execution engine, and the discussion so far largely ignores how static scheduling with boosting affects other secondary issues that are extremely important when determining the true feasibility of an idea. This section focuses on two of those secondary issues. The first issue is a performance question that specifically deals with the effect of static scheduling on the instruction memory subsystem. The second issue is a commercial concern that specifically deals with the effect of boosting on code compatibility. These brief discussions follow our philosophy of integrated approach to high-performance processor design, and they uphold the conclusions of this thesis.

B.1 Code compaction

Our global scheduling algorithm of Section 3.3 contains a compaction routine, and though compaction is not an actual part of the scheduling routine, compaction is an integral part of instruction scheduling for superscalar machines with limited hardware for dynamic dependence checking. Compaction involves the removal of the static NOP instructions from the software schedule. Compaction of some sort is necessary since the global scheduler cannot always fill every instruction issue slot. Currently, the global scheduler fills each empty issue slot with a NOP instruction. These static NOPs increase the size of the object file and therefore decrease the effectiveness of a fixed-size instruction cache. By increasing the complexity of the hardware issue and execution mechanisms, one can remove more and more of these static NOPs. Of course, one could remove all of the static NOPs by implementing a machine that checks for dependences between instructions in issue and by implementing a compiler that schedules anti-dependences with a dependence edge delay of one cycle. Yet, this solution ignores the fact that the compiler already determined what the hardware could issue in parallel. Why throw all of this information away and make the hardware regenerate it? A solution somewhere in the middle seems much more appealing. As usual though, the question remains of how much complexity one should put in the hardware and how much help can the compiler provide.

The instruction fetch and decode logic is an important component of the overall system design. A superscalar processor would like to maintain the simplicity of the fetch and decode subsystem found in today's RISC processors. The fixed-length, fixed-format style of RISC instructions decouples instruction fetch from instruction decode so that fetch and decode can operate in parallel. To achieve a decoupling of the fetch and decode units in most superscalar processors today, designers include prefetch and buffering logic. This logic is necessary in superscalar machines that perform dependence checking in the fetch unit and thus allow for variable-sized issue packets. The prefetching and dependence-checking logic can add a significant amount of hardware to the processor, and this logic can often increase the number of stages in the pipeline (e.g. see the description of the SUN SuperSparc [Case 1991]). Additional techniques are then required to ensure that the branch delay does not increase because of the additional pipe stages.

Our approach to this problem minimizes the fetch and dependence-analysis hardware while still removing a large number of the static NOP instructions. In this approach, the software does what it is best at doing—determining the grouping of independent instructions; and the hardware does what it is best at doing—nullifying instruction issue slots. In other words, the hardware dynamically inserts NOPs into the instruction execution stream where the software requests, without the software having to insert an entire 32-bit NOP instruction. The approach allows the compiler to condense a set of multiple execution packets into a single fetch packet. This approach is achieved by including some extra information with each fetch packet that tells the hardware how to reconstruction the original set of execution packets. Currently, this extra information is encoded as *timing bits* in each instruction word. The timing bits tell the hardware how long to delay from the packet fetch till that instruction's issue. The timing bits succinctly inform the hardware of dependences between the instruction in the fetch packet, and they greatly reduce the complexity of the instruction issue unit. By never splitting an execution packets to finish out the fill of a fetch packet, the complexity of the instruction fetch unit is also greatly reduced (prefetching is unnecessary). Figure B-1 contains three examples from a two-issue machine with a single timing bit per instruction; the compiler indicates a delay of one cycle by prefixing a "D." specifier to the front of the opcode. In the first two examples, the compiler compacts the execution sequences, but in the last example, the compiler does not compact the sequence because compaction would split the second execution packet (and ultimately lengthen the execution of this sequence because the hardware does not prefetch and merge instructions from different packets). Figure B-2 outlines the compiler algorithm that compacts a software schedule for this hardware configuration.

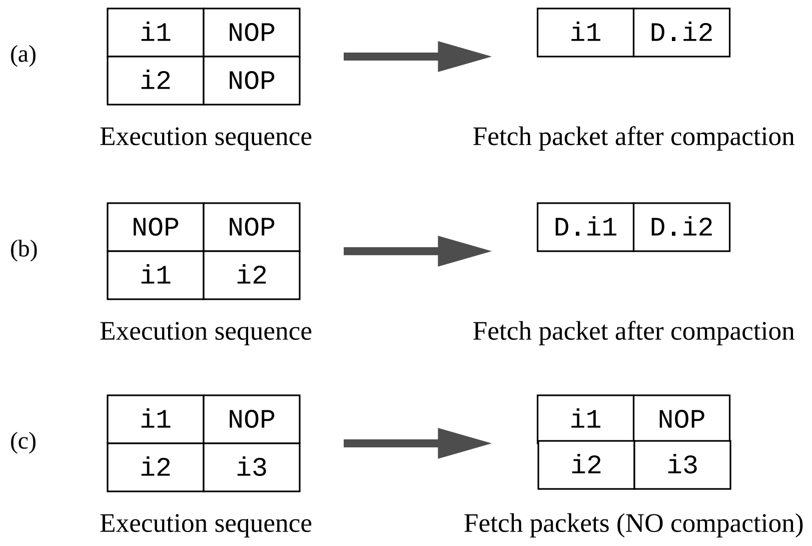


Figure B-1: Examples of compaction with a single timing bit per instruction.

```
d = 0;
cpkt = first packet in software schedule;
npkt = next packet in software schedule;
if (no npkt) done;
while (not done) {
   if (number of NOPs in cpkt >= number of instrs in npkt) {
      d++;
      put instructions from npkt into cpkt with delay d;
   } else {
      d = 0;
      output cpkt;
      cpkt = npkt;
   }
   npkt = next packet in software schedule;
   if (no npkt) done;
}
output cpkt;
```

Figure B-2: Algorithm for compacting a software schedule for hardware with timing bits.

The purpose of this section is not to completely explain and evaluate this mechanism, but to demonstrate that global instruction scheduling can improve more than just the efficiency of the functional units. Of course, a full implementation of our technique requires an understanding of how the mechanism interacts with scheduler changes to branch and jump targets (e.g. scheduling from already-scheduled basic blocks), with limited-issue architectures, with hardware interlocks, and with exceptions. The utility of this mechanism does not show up in CPU cycle counts. The mechanism simplifies fetch and decode hardware which affects the cycle time of the machine, and the mechanism removes static NOPs which reduces the number of cycles spent in the instruction memory subsystem. So, with all of these caveats aside, this mechanism reduces the number of static NOP instructions in the applications of Chapter 4 from around 30-35% of the total instructions to around 15-20% of the total instructions. The machine model in this experiment is a two-issue, limited-issue superscalar machine with no load interlocks and one-level of boosting (one shadow register file and one shadow store buffer).

B.2 Code compatibility

In addition to performance, code compatibility is also an important issue in the commercial world. Architectural extensions such as boosting require more instruction encoding space than exists in the current 32-bit RISC architectures. Architectural extensions possibly jeopardize compatibility between different extensions of a given architecture because the extensions could differ in instruction length and encoding. Furthermore, strict static scheduling also jeopardizes compatibility between different implementations of the same architecture because the implementations could differ in issue width and functional unit organization.

This subsection briefly addresses the issue of code compatibility in our TORCH implementation. The purpose of this subsection, like the last subsection, is not to completely explain and evaluate some solution to the code compatibility issue, but to demonstrate that architectural extensions and static scheduling do not destroy code compatibility. As always, a full evaluation of the solutions requires an understanding of how each mechanism interacts with instruction addressing, virtual memory, etc.

For architectural extensions, the simplest method of increasing the instruction encoding space is to use prefix bytes, such as was done in the Intel 80386 [Crawford 1986]. Prefix bytes ensure compatibility by leaving the bits that compose the original instruction format unchanged and encode the additional functionality required by the architectural extensions in the new instruction bits. In TORCH, we use a prefix byte to encode the architectural extension information for each instruction. Specifically, TORCH instructions are composed of a 32-bit base component that is identical to a MIPS instruction and an 8-bit extension byte that contains the boosting and timing information. Figure B-3 illustrates a sequential and boosted encoding for a typical MIPS instruction.

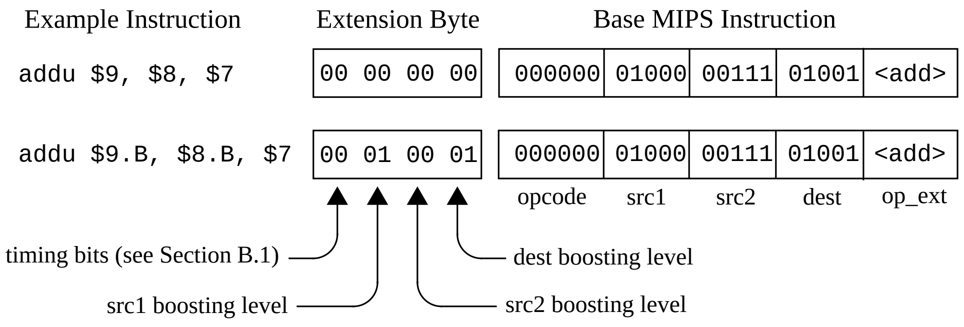


Figure B-3: Example of instruction encodings in TORCH.

To effectively satisfy the high instruction bandwidth of the superscalar processor, we organize the primary instruction cache in TORCH so that it caches 40-bit objects. That is, the primary instruction cache contains a binary number of 40-bit instructions per cache line, and the program counter directly indexes the primary cache (the program counter is an instruction number and not the byte address of the instruction). To minimize the effect of the 40-bit instructions on the later levels of the memory hierarchy, a translation takes place during a primary instruction cache miss that translates an instruction number into a byte address (the translation is a multiplication by 5/4 which corresponds to a shift-and-add operation on the program counter). The advantages of this scheme are that the implementation of the CPU and primary instruction cache (which is most-likely on the same chip with the CPU) completely hide the complexity of the 40-bit instructions and that the implementation optimizes the performance of the time-critical path between the CPU and the primary instruction cache. Preliminary studies show that this solution typically incurs less than a 2% degradation in instruction memory system performance. The memory system in this preliminary study is typical of those found in today's high-performance machines: 8 kilobyte primary instruction cache with a 6–8 cycle miss penalty, and a 512 kilobyte combined secondary cache with a 20–25 cycle miss penalty.

Now, the execution of a 32-bit MIPS object file on the 40-bit TORCH implementation simply requires the cache miss handler to correctly fill the 40-bit instruction cache with

32-bit instructions. Specifically, the primary instruction cache miss handler does not translate the program counter (an instruction number for a 32-bit instruction is its byte address), and the handler defaults the prefix bytes to some set pattern. The simplest fill pattern is one that forces single instruction issue. Yet, better performance results from the inclusion of some dependence analysis hardware in the miss handling logic. This augmented miss handler analyzes each packet of instructions, and it appropriately sets the timing bits to take advantage of any possible parallel issue. A similar scenario handles the execution of an object file compiled for a two-issue machine on a four-issue machine. This approach uses dynamic scheduling to achieve efficient code compatibility because dynamic scheduling best satisfies the requirements of code compatibility. Still, this approach limits the complexity of the dynamic scheduling hardware by analyzing instructions only within a packet, and it limits the cost of the dynamic scheduling hardware by analyzing the instructions during the filling of the primary instruction cache (outside the critical path of the processor). This approach demonstrates another tradeoff between functionality, cost, and performance.

References

[Aho et al. 1986]

A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Bernstein and Rodeh 1991]

David Bernstein and Michael Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 241–255, June 1991.

[Bernstein et al. 1991]

David Bernstein, Doron Cohen, and Hugo Krawczyk. Code Duplication: An Assist for Global Instruction Scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 103–113, November 1991.

[Bradlee et al. 1991]

David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. In the *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 122–131, April, 1991.

[Case 1991]

Brian Case. Superscalar Techniques: SuperSPARC vs. 88110. *Microprocessor Report*, 5(22):1–11, December 1991.

[Chang and Hwu 1991]

Pohua Chang and Wen-mei Hwu. Profile-Guided Automatic Inline Expansion for C Programs. Center for Reliable and High-Performance Computing Report CRHC-91-13, Univ. of Illinois at Urbana-Champaign, Urbana, IL, April, 1991.

[Chang et al. 1991a]

Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. Using Profile Information to Assist Classic Code Optimizations. Center for Reliable and High-Performance Computing Report CRHC-91-12, University of Illinois at Urbana-Champaign, Urbana, IL, April, 1991.

[Chang et al. 1991b]

Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen-mei W. Hwu. IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors. In the *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[Chow 1989]

Paul Chow, editor. *The MIPS-X RISC Microprocessor.* Kluwer Academic Publishers, Boston, MA, 1989.

[Chow and Hennessy 1990]

F.C. Chow and J.L. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems,* 12(4), October 1990.

[Colwell et al. 1987]

R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, P. K. Rodman. A VLIW Architecture for a Trace Scheduling Compiler. In the *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 180–192, October, 1987.

[Crawford 1986]

J. Crawford. Architecture of the Intel 80386. In the *Proceedings of the IEEE International Conference on Computer Design*, pp. 155–160, October 1986.

[Davidson et al. 1981]

Scott Davidson, David Landskov, Bruce D. Shriver, and Patrick W. Mallett. Some Experiments in Local Microcode Compaction for Horizontal Machines. *IEEE Transactions on Computers*, C-30(7):460–477, July 1981.

[DEC 1992]

Digital Equipment Corporation. *DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet*. Digital Equipment Corporation, Maynard, MA, April 1992.

[Ebcioğlu 1988]

Kemal Ebcioğlu. Some Design Ideas for a VLIW Architecture for Sequential-Natured Software. In *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing),* edited by M. Cosnard et al., North Holland, pp. 3–21, April 1988.

[Ebcioğlu and Nakatani 1989]

K. Ebcioğlu and T. Nakatani. A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture. In *Languages and Compilers for Parallel Computing*, edited by D. Gelernter et al., Research Monographs in Parallel and Distributed Computing, MIT Press, pp. 213–229, 1988.

[Ebcioğlu and Nicolau 1989]

K. Ebcioğlu and A. Nicolau. A Global Resource-Constrained Parallelization Technique. In *Proceedings of the Third International Conference on Supercomputing*, pp. 154–163, June 1989.

[Ellis 1985]

John R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. Ph.D. thesis, Yale University, Department of Computer Science, February 1985. (Technical Report No. YALEU/DCS/RR-364).

[Fisher 1980]

Joseph A. Fisher. 2^n-Way Jump Microinstruction Hardware and an Effective Instruction Binding Method. In *The 13th Annual Microprogramming Workshop*, pp. 64–75, November 1980.

[Fisher 1981]

Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478-490, July 1981.

[Fisher 1983]

Joseph A. Fisher. Very Long Instruction Word Architectures and ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 140–150, June 1983.

[Fisher and Freudenberger 1992]

Joseph A. Fisher and Stefan M. Freudenberger. Predicting Conditional Branch Directions From Previous Runs of a Program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85–95, October 1992.

[Fu et al. 1987]

John Fu, James B. Keller, and Kenneth J. Haduch. Aspects of the VAX 8800 C Box Design. *Digital Technical Journal*, No. 4, pp. 41–51, February 1987.

[Gibbons and Muchnick 1986]

Phillip B. Gibbons and Steven S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, pp. 11–16, June 1986.

[Goodman and Hsu 1988]

James R. Goodman and Wei-Chung Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of the 1988 International Conference on Supercomputing*, pp. 442–452, July 1988.

[Gross 1983]

Thomas Gross. Code Optimization of Pipeline Constraints. Technical Report No. 83-255, Stanford University, Stanford, California 94305, December 1983.

[Gross and Ward 1991]

T. Gross and M. Ward. The Suppression of Compensation Code. In *Advances in Languages and Compilers for Parallel Processing*, The MIT Press, Cambridge, MA, pp. 260–273, 1991.

[Hennessy and Patterson 1990]

J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[Hsu and Davidson 1986]

Peter Y. T. Hsu and Edward S. Davidson. Highly Concurrent Scalar Processing. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[Hwu and Patt 1987]

Wen-mei W. Hwu and Yale N. Patt. Checkpoint Repair for Out-of-order Execution Machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 18–26, June 1987.

[Johnson 1990]

Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[Jouppi and Wall 1989]

Norman P. Jouppi and David W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, April 1989.

[Kane 1987]

Gerry Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[Keller 1975]

Robert M. Keller. Look-Ahead Processors. *Computing Surveys*, 7(4):177–195, December 1975.

[Lam 1988]

Monica S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.

[Lam 1990]

Monica S. Lam. Instruction Scheduling for Superscalar Architectures. *Annual Review of Computer Science*, Vol. 4, pp. 173–201, 1990.

[Lam and Wilson 1992]

Monica S. Lam and Robert P. Wilson. Limits of Control Flow on Parallelism. In the *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 46–57, May 1992.

[Mahlke et al. 1992]

Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael 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*, pp. 238–247, October 1992.

[Melvin and Patt 1991]

Stephen Melvin and Yale Patt. Exploiting Fine-Grained Parallelism Through a Combination of Hardware and Software Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 287–296, May 1991.

[Moon and Ebcioğlu 1988]

Soo-Mook Moon and Kemal Ebcioğlu. An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors. Computer Science Reseach Report RC 17962 (#78691), IBM Research Division, Yorktown Heights, NY, April 1992.

[Nicolau and Fisher 1984]

Alexandru Nicolau and Joseph A. Fisher. Measuring the Parallelism Available for Very Long Instruction Word Architectures. *IEEE Transactions on Computers*, C-33(11):968–976, November 1984.

[Nicolau 1985]

A. Nicolau. Percolation Scheduling: A Parallel Compilation Technique. Computer Sciences Technical Report 85-678, Cornell University, May 1985.

[Nicolau 1989]

A. Nicolau. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions on Computers*, C-38(5):663–678, May 1989.

[JESmith 1982]

James E. Smith. Decoupled Access/Execute Computer Architectures. In P*roceedings of the 9th Annual International Symposium on Computer Architecture*, pp. 112-119, April 1982.

[JESmith and Pleszkun 1985]

James E. Smith and Andrew R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 36–44, June 1985.

[MDSmith et al. 1989]

Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on Multiple Instruction Issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 290–302, April 1989.

[MDSmith et al. 1990]

Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting Beyond Static Scheduling in a Superscalar Processor. In the *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 344–354, May 1990.

[MDSmith 1991]

Michael D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, Stanford, California 94305, November 1991.

[MDSmith et al. 1992]

Michael D. Smith, Mark Horowitz, and Monica S. Lam. Efficient Superscalar Performance through Boosting. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 248–259, October 1992.

[Smotherman et al. 1991]

Mark Smotherman, Sanjay Krishnamurthy, P.S. Aravind, and David Hunnicutt. Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 93–102, November 1991.

[Sohi and Vajapeyam 1987]

Gurindar S. Sohi and Sriram Vajapeyam. Instruction Issue Logic for High-Performance, Interruptible Pipelined Processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 27-34, July 1987.

[SPEC 1990]

SPEC. *The SPEC Benchmark Report*. Waterside Associates, Fremont, CA, January 1990.

[Thornton 1964]

J.E. Thornton. Parallel Operation in the Control Data 6600. In *Proc. AFIPS FJCC*, 26(2):33-40, 1964.

[Tjiang et al. 1991]

S. Tjiang, M.E. Wolf, M.S. Lam, K.L. Pieper, and J.L. Hennessy. Integrating Scalar Optimization and Parallelization. In *4th Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[Tjiang and Hennessy 1992]

S.W.K. Tjiang and J.L. Hennessy. Sharlit—A Tool for Building Optimizers. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 82–93, June 1992.

[Tokoro et al. 1981]

M. Tokoro, E. Tamura, and T. Takizuka. Optimization of Microprograms. *IEEE Transactions on Computers*, C-30(7):491–504, July 1981.

[Tomasulo 1967]

R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal*, 11(1):25–33, January 1967.

[Uvieghara et al. 1992]

Gregory A. Uvieghara et al. An Experimental Single-Chip Data Flow CPU. *IEEE Journal of Solid-State Circuits*, 27(1):17-28, January 1992.

[Wall 1991]

David W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, April 1991.