Performance Nonmonotonicities: A Case Study of the UltraSPARC Processor

by

Nathaniel A. Kushman

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

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1998

Author ....
Department of Electrical Engineering and Computer Science
June 1, 1998

Certified by ....
Volker Strumpen
Postdoctoral Associate
Thesis Supervisor

Certified by ....
Charles E. Leiserson
Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Performance Nonmonotonicities:
A Case Study of the UltraSPARC Processor

by

Nathaniel A. Kushman

Submitted to the Department of Electrical Engineering and Computer Science
on June 1, 1998, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Modern microprocessor architectures are very complex designs. Consequently, they exhibit many idiosyncrasies. In fact, situations exist in which the addition or removal of a single instruction changes the performance of a program by a factor of 3 to 4. I call such situations *performance anomalies.* Avoiding these situations requires detailed understanding of the underlying architecture. Unfortunately, due to market competition, microprocessor vendors are unwilling to release the detailed implementation information necessary to understand an architecture until long after the microprocessors have been on the market. Through a case study of the SUN UltraSPARC, I show how these anomalies can be concealed, although only limited information is provided by the vendor. I explain the cause of four performance anomalies observed on the UltraSPARC, and present an algorithm to conceal each of them. I implemented these algorithms in an assembly code restructuring tool which yields speedups of 2.2% on average for the SPECint benchmark suite, and up to 8.9% on individual benchmarks.

Thesis Supervisor: Volker Strumpen
Title: Postdoctoral Associate

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

First and foremost, I am indebted to Dr. Volker Strumpen. His unending persistence in working through problems down to the last detail has allowed me to gain a more complete understanding of the UltraSPARC. Additionally, his constant desire for clarity and precision in all forms of explanation, has allowed this document to reach it's current form. Without his willingness to spend hours working through my explanations, many parts of this thesis would have remained unclear and imprecise. For all of this, I give Volker my perpetual gratitude.

I would like to thank the other members of the Cilk group and the Computer Architecture group for their insights into the problems, and their suggestions on this document. I would especially like to thank Charles for his suggestions on this thesis, and on academic life in general.

The research in this thesis was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-97-1-0270. Additionally, I would like to thank Sun Microsystems for the use of the Xolas UltraSPARC SMP cluster.

Lastly, I would like to thank my friends and family for their support through many late nights. Therefore, I thank Greg Christiana and Carissa Little for helping me survive the last few weeks spent writing this document. I would also especially like to thank my parents; my mom for putting up with me when I didn't return her phone calls because I was too busy; and my dad for his constant support throughout my four years at MIT.

Contents

List of Figures

Chapter 1

Introduction

In early computers, most instructions were executed in the same amount of time, and the overall execution time could be estimated by counting the number of instructions executed [9]. As machines became more complex, however, this situation changed quickly. Machines were pipelined, and dependencies between instructions became important. When caches were developed, the layout of instructions and data in memory became important. With the introduction of superscalar processors, evaluating processor performance has become problematic. In these processors, multiple instructions are executed at each stage of a multiple-stage pipeline. There may be multiple outstanding memory references, and the dependencies between instructions affect their execution sequence. These processors are so complex that it is difficult to predict their performance.

Because of this complexity, significant time and effort in the design of a processor is spent on performance modeling [15]. The effect of individual design decisions on the overall performance is often unclear without tools to aid the designers' intuitions. Therefore, hardware designers use performance models to assist them in making difficult design decisions. Unfortunately, the validity of these models is debatable. If these models are in fact invalid, as argued by Black and Shen [6], a huge potential for *performance anomalies* exists. These performance anomalies are the focus of this thesis.

In the remainder of this chapter, I introduce a taxonomy of performance anomalies, I give an example to show how performance anomalies can be revealed in a computer system, I discuss the difficulty in avoiding performance anomalies and I present the approach to the problem of performance anomalies that underlies this thesis.

Performance Anomalies

A ***performance anomaly*** is an unexpected runtime behavior of a program. I present three types of performance anomalies—discontinuity, nonmonotonicity, and nondeterminism—and define them according to the relationship of work (W) and execution time (T). I use the term ***work*** to refer to the total number of instructions executed to completion.

In general, as the amount of work in a program increases, the execution time increases proportionally. Performance anomalies are situations in which this is *not* the case. The following definitions provide an intuition into the understanding of performance anomalies, but are not rigorous definitions. For each anomaly, I assume that a program has work W and executes in time T. Modifying this program produces a second program that has work W' and executes in time T'.

Performance Discontinuity: A performance discontinuity is an unexpected, disproportionate jump in execution time.

If work increases from W to W', then T' increases disproportionately compared to T:

$W' > W \Rightarrow T' \gg T$

If work decreases from W to W', T' decreases disproportionately compared to T:

$W' < W \Rightarrow T' \ll T$

Performance Nonmonotonicity: If work and execution time change in opposite directions, a performance nonmonotonicity is revealed.

Although work W does not increase, execution time T increases:

$W' \leq W \Rightarrow T' > T$

Although work W does not decrease, execution time T decreases:

$W' \geq W \Rightarrow T' < T$

Performance Nondeterminism: If the same program is executed twice without changing work W, and the execution time T changes, then a performance nondeterminism is revealed:

$W' = W \Rightarrow T' \neq T$

A common type of performance anomaly in today's computer systems is performance discontinuity. Bailey [4] shows that caching causes machines to exhibit performance discontinuities. Four examples of performance discontinuities are presented in this thesis. Each of them is also an example of a performance nonmonotonicity.

This thesis focuses on avoiding *performance nonmonotonicities.* If the relationship of work and execution time of a program contains performance nonmonotonicities, it is almost impossible for a programmer to converge on a high-performing version of his code. Programming for performance is even more difficult if the relationship of work and execution time contains discontinuous performance nonmonotonicities. An example of nonmonotonic performance in the Intel Pentium is presented by Krech [10]. Krech's example of nonmonotonicity was exhibited when performing a particular sequence of memory reads. In Chapter 3, I present four examples of performance nonmonotonicities on the SUN UltraSPARC and show how to conceal them.

```c
foo(){}

main()
{
     int i;

     for (i = 0; i < 100000000; ++i)
         foo();
}
```

Program **A**

```c
foo(){}

main()
{
     int i;

     foo();

     for (i = 0; i < 100000000; ++i)
        foo();
}
```

Program **B**

Figure 1-1: Program **A** runs in between 6.3 and 8.4 seconds and Program **B** runs in 2.1 seconds.

Nondeterministic performance anomalies are the most extreme type of performance anomaly. It is nearly impossible to program for performance if the relationship of execution time and work is nondeterministic. Performance nondeterminism is often observed in timeshared environments, even if the system is used by only one user. The operating system, and specifically, the virtual memory page mapping algorithms are often identified as the cause of this nondeterminism [5]. I have found, however, that the *processor itself* also exhibits performance nondeterminism. Two of the performance nonmonotonicities identified on the UltraSPARC are also examples of performance nondeterminism. I shall present algorithms to conceal the performance nondeterminisms, but I was unable to determine their cause.

An Example

The code fragment in Figure 1-1 Program **A** exhibits all three performance anomalies to the user: performance discontinuity, performance monotonicity, and performance nondeterminism. Program **A** in Figure 1-1 iterates a hundred million times in a loop calling an empty function `foo`. Compiling this code with Sun's C-compiler (cc) using -O and executing it on a SUN UltraSPARC (143MHz) yields nondeterministic execution times between 6.3 and 8.4 seconds. If work is added to that program by inserting a call to the empty function `foo` before the loop, the program executes deterministically in 2.1 seconds. The modified program is shown in Figure 1-1 as program **B**. Surprisingly, adding a tiny amount of work reduces execution time by a factor of 3 or 4. Consequently, this code fragment is an example of all three performance anomalies: performance discontinuity, performance nonmonotonicity, and performance nondeterminism. We shall revisit this code fragment in Chapter 3.

The Dilemma

Avoiding performance anomalies seems even more difficult than avoiding correctness bugs. I hypothesize that this issue will become more significant as microprocessor complexity increases. The stiff competition in the microprocessor industry prevents vendors from releasing implementation details to the public until long after the processors have been on the market. Consequently, most performance anomalies that exist in an architecture remain a mystery to the users of the system. Users are left with little or no information about the cause of these performance problems, and no systematic way to avoid them exists to date. Additionally, since the introduction of RISC, microprocessor complexity has been increasing steadily. If this increase continues to the one-billion transistor microprocessors of the future [24], I fear that the performance problems caused by the complexity of current microprocessors will increase without bound. I propose the following resolution to this dilemma.

The Resolution

Unless processors are designed without performance anomalies, a method to prevent programmers from observing these anomalies is desired. Experience has shown us that it is difficult to avoid performance anomalies in the design of current microprocessors. We should be able to ***conceal*** these anomalies from the programmer, however, such that they are never observed in the execution of compiled code. I do this by first understanding the cause of each performance anomaly, and then developing an instruction scheduling algorithm to conceal it.

I wish to understand a microprocessor to the extent that I can characterize the situations in which it exhibits nonmonotonic performance. Yet, in general, we have very little information on the implementation details of the processor itself. This situation is comparable to that of a chemist who wishes to understand a chemical reaction without a-priori knowledge of how the reaction works. I present an approach to the problem of performance anomalies that is based on the methods of experimental science.

I employ an iterative approach that uses the data derived from microbenchmarks in conjunction with information provided by the microprocessor vendor to create a model of the machine. In contrast to the performance models created in the past [25], I create a very detailed model of particular aspects of the architecture. This model allows me to focus specifically on the architectural features which cause performance anomalies. I attempt to verify the accuracy of my model through the use of microbenchmarks. The results from these microbenchmarks are used to refine the model and create more microbenchmarks. This iterative process is repeated until the performance anomaly is understood to the extent that an instruction scheduling algorithm can be developed to conceal the anomaly from the user.

Outline of the Thesis

In order to identify and conceal processor performance anomalies I decompose the problem into three constituent parts: (1) finding, (2) understanding, and (3) concealing performance anomalies.

To begin work on the problem of performance anomalies, code sequences must be found that exhibit a performance anomaly. Figure 1-1 shows an example of such a code sequence. There appears to be no systematic way to find such code sequences, however. The performance anomalies discussed in this thesis were found either from information provided by the microprocessor vendor or circumstantially.

To conceal a performance anomaly, we must identify what aspect of the processor design is causing it. Then, we can design a code sequence to isolate the anomaly. Chapter 2 presents other methods used for identifying and concealing performance anomalies and explains the problems with each. I employed the iterative approach described in the previous section to understand four performance anomalies on the SUN UltraSPARC. This understanding is one of the main contributions of this thesis and is presented in Chapter 3.

Once an insight into the cause of a performance anomaly is gained, a general method for concealing it should be developed. Ideally, these methods should be implemented within the compiler. Generally, we are unable to modify SUN's compiler, however, so I have implemented these methods through a restructuring tool for assembly code. The algorithms for the restructuring of assembly code, are the second contribution of this thesis. In Chapter 4, I show how they were used to obtain speedups of up to 9% on the SPECint benchmarks [23]. Finally, in Chapter 5, I conclude.

Chapter 2

Related Work

Little research has been focused on the problem of processor performance anomalies. In this Chapter, I discuss five approaches to assessing hardware performance problems. The first three approaches, architecture simulation, hardware performance monitoring, and performance modeling can help to understand a performance problem. The last two approaches, architectural design and code restructuring, can be used to actually conceal known performance problems.

Architecture Simulation

A widely used technique for understanding and designing processors is architecture simulation. If not enough information can be obtained by running a program directly on the hardware, the architecture is simulated with sufficient detail to obtain the desired information.

Architecture simulation ranges from the Instruction Set Architecture (ISA) level [12] down to the transistor level [13]. Generally, the more detail the simulation provides, the worse the performance. The Stanford SIMOS project [17] provides an excellent example of the trade-off between detail and speed. This simulator provides several different modes, each with a distinct locale in the trade-off between detail and speed. In the so-called direct-execution mode, where the simulator runs the code directly on the host architecture, the simulated application runs only a factor of 2 slower than the code running natively. Alternatively, when SIMOS is running with the highest level of detail, storing information on the movement of instructions through the pipeline, the simulation runs up to a factor of 50,000 slower than the native code.

Architecture simulation has been used extensively for two purposes. First, it is the main approach used to evaluate hardware design decisions during the design phase. Second, simulators for existing architectures are used for performance tuning of programs that run on these machines. At the Swedish Institute of Computer Science, the SimICS system [12] has been used to show that architecture simulators, when used for performance tuning, can achieve up to an order of magnitude speedup on some specific applications.

To my knowledge however, architecture simulators have not been applied to the problem of identifying processor performance anomalies in processors currently on the market. I did not use such simulators, because specific information on the implementation of an architecture is required to produce the detailed simulators necessary to understand performance anomalies. In general, microprocessor vendors do not release information of sufficient detail to produce such simulators.

Hardware Performance Monitoring

Monitoring performance within the hardware avoids the problems presented by architecture simulation. First, hardware monitors are implemented as part of the architecture, so no information on the implementation of the architecture is necessary to use them. Secondly, they are designed to have no effect on the execution of the program, which helps ensure that any performance problems that occur during the normal execution of a program also occur when using hardware performance monitoring to observe the performance, and vice versa.

Hardware performance monitoring is accomplished through hardware counters. These counters accumulate the number of times that events occur within the architecture. The countable events include cache-misses, branch mispredictions, decoded instructions, retired instructions, and other events that might indicate the hardware is not performing optimally [14]. Hardware counters have been around for several years [20, 26]. Today, almost all mainstream microprocessors implement hardware counters. Additionally, many vendors have either developed special software to use the counters [2, 3] or have integrated their use into their existing profiling packages [1, 28]. Some vendors do not distribute commercial packages, but use proprietary software for tuning benchmarks and commercial software packages.

Hardware counters provide information on the number of times events occur within a section of code, but they do not allow a user to tell exactly which instruction triggered these events. For this reason, hardware counters are usually used in conjunction with other techniques such as path profiling and continuous profiling [1, 2, 3].

Work has been done to show the utility of hardware counters in improving the interaction between the hardware and the software. To my knowledge, however, none of this work has included any discussion of processor performance anomalies. Nevertheless, hardware counters can be used to help find and understand performance anomalies. I use hardware counters to monitor the performance of microbenchmarks, allowing me to isolate the particular instruction sequences that cause an anomaly.

Performance Modeling

Performance models are mathematical models used to predict the performance of an application on a computer system. The performance of a computer system is represented using a small number of measurements. The mathematical model then uses these measurements to predict the performance of an application on the given system [8, 18]. Toledo [25] presents a model where a small amount of dynamic information is used to produce a more accurate model.

Performance modeling techniques are focused on two areas: software design and hardware design. Performance models help a programmer discover what parts of the code are causing the hardware to stall. They also help a hardware designer determine the performance bottlenecks in the design of an architecture. To my knowledge, however, performance modeling has not been applied to the problem of identifying and fixing performance anomalies. Past techniques have included only ISA-level modeling of an entire computer system. I use information provided by the vendor, and results from running microbenchmarks to produce an architecture-level model of particular features of a processor. The model I produce is more than just a performance model though. My model represents a detailed understanding of how the instruction sequence interacts with the particular features of the architecture.

Architectural Design

The ideal solution to the problem of processor performance anomalies is to avoid them altogether in the design of the processor architecture. Currently, architectural designers spend a significant amount of time and energy to avoid performance bottlenecks, making use of performance modeling techniques to aid them in their design decisions [6, 7, 15]. Errors in these models, however, make it difficult to design an architecture for performance.

Performance models used by hardware designers are attempts to extract the information that affects performance from the design. Unfortunately, there is no rigorous method for finding errors in these performance models. These errors are situations where the performance model incorrectly predicts the performance of the hardware. Developers simply check the plausibility of the results the models predict. If they find surprising results, they look for possible errors in their performance model. Unfortunately, this method of validation by inspection is error-prone, and research is being done to develop better methods for validating performance models. Even more rigorous methods of validation, such as those proposed by Black and Shen [6], rely on developer-written test suites that may not test the model effectively.

Improved methods for validation of performance models may eventually reduce the number of performance problems. The current method of avoiding performance anomalies during the hardware design phase, however, is still relatively haphazard, leaving processors with many performance anomalies.

Code Restructuring

The term "code restructuring" has been used to refer to a variety of different techniques. In the following, ***code restructuring*** refers to transforming an executable code into a semantically equivalent executable code by rearranging the order of instructions.

Code restructuring is useful in circumstances where the original source code is not readily available. Often, older binaries are restructured to perform better on a new implementation of the same ISA [16]. For example, binaries compiled for the Pentium may be restructured to be optimized for the Pentium Pro [19]. Analogously, restructuring can be used to reschedule executables to conceal performance anomalies.

A disadvantage of code restructurers is that their construction involves significant implementation effort. Unfortunately, the process of restructuring code is not as simple as disassembling the code, rearranging instructions, and then reassembling it. The difficulty arises from the fact that rearranging code changes the the placement of branch targets within the code sequence. Therefore, the restructurer must ensure that the destination of each branch instruction is correct, requiring the modification of instructions rather than only reordering. Ensuring this condition becomes problematic in the presence of indirect jumps, where branch targets are extremely hard to determine, because the target of some branches is actually stored as data. Additionally, some binaries contain data stored within the text segment, making it difficult to even detect which sections of the binary can be restructured and which must remain untouched. The engineering effort required to write a code restructuring tool may outweigh its possible benefits.

Several tools have been introduced to aid in the process of developing code restructuring tools [11, 16, 21]. These tools provide a high-level abstraction of the binary code which greatly simplifies the process of writing a code restructurer. The abstraction layer presented by these tools, however, does not allow the user to control the exact layout of instructions because implementation details handled by the tools may require the insertion of additional instructions. I found that it was necessary to control the exact layout of instructions in order to conceal processor performance anomalies. Therefore, I did not use such tools to write a code restructurer.

I choose instead to conceal performance anomalies by restructuring the assembly code produced by the compiler. In this assembly code all branch targets are represented by labels. Consequently, all branch targets can be assigned by the assembler, allowing changes to be made to the assembly code without worrying about the destination of branch instructions. Concealing performance anomalies through the restructuring of assembly code requires access to the assembly code produced by the compiler, but simplifies the implementation effort considerably.

Chapter 3

UltraSPARC Case Study

To show that performance anomalies constitute a serious problem on commercial microprocessors, I performed a case study of the UltraSPARC Microprocessor, developed by Sun Microsystems [22]. The UltraSPARC is an implementation of the SPARC-V9 ISA[1] [27]. Through this study, I identified four performance anomalies caused by the design of the architecture. Each of these anomalies was caused by a specific feature of the architecture: next field predictors, fetching logic, grouping logic, and branch prediction logic. I created a set of microbenchmarks for each feature. I use information from the UltraSPARC User's Manual [14], and from the execution times of these microbenchmarks to identify the cause of each anomaly.

```
foo:
        retl
        nop
         :
.LL7:
        call foo,0
        add %l0,-1,%l0
        cmp %l0,0
        bg .LL7
        nop
```

Figure 3-1: Snippets of the assembly code generated from the C-code in Figure 1-1, used to create a microbenchmark.

The example from Chapter 1 is used to show the process of creating microbenchmarks. The process is begun with a short code sequence that exhibits a performance anomaly. For this example, the code sequence in Figure 3-1 is used. This code sequence is produced by compiling the the C code in Figure 1-1, and thus exhibits all three types of performance anomalies. A code sequence is

[1] This is an open standard developed by SPARC International, Inc.

```
.LL7:           (align 32)
        nop
        nop
        add %l0,-1,%l0
        nop
        nop
        nop
        br foo
        nop
        nop
        nop
        nop
.LL8:
        nop
        nop
        nop
        nop
        cmp %l0,0
        nop
        nop
        nop
        nop
        bg .LL7
        nop
```

```
foo:            (align 32)
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        br .LL8
        nop
```

Figure 3-2: A microbenchmark created from the assembly code in Figure 3-1.

then created that is similar to the original, yet does not exhibit any performance anomalies. Such a code sequence is shown in Figure 3-2. This code sequence was created from the code sequence in Figure 3-1 by inserting `nops` to isolate all instructions and replacing the call and return instructions with explicit branch instructions. Finally, this code sequence is carefully changed to produced a set of microbenchmarks, each of which is intended to test one possible cause of the performance anomaly. This process allowed the code sequence in Figure 3-1 to be decomposed into two microbenchmarks, each exhibiting only one performance anomaly.

I have observed four performance anomalies, on the UltraSPARC processor, each caused by a specific feature of it's design. In the remainder of this chapter, I present my understanding of the design of the UltraSPARC and explain how the specific features cause performance anomalies. In the first section I give an overview of the portion of the architecture that is the source of all four observed performance anomalies. I describe two architectural features, the I-buffer and the next field predictors, both of which are important to understanding the observed performance anomalies. The following four sections describe the features of the architecture that cause the observed performance anomalies. These features include the next field predictor table, the fetching logic, the grouping logic, and the branch prediction logic. For each of these features I provide a description of my understanding of the logic design, a characterization of the associated performance anomaly, the microbenchmarks used to identify the anomalies, and the algorithm used to conceal the anomaly.

Figure 3-3: The nine-stage pipeline of the UltraSPARC.

3.1 The Architecture of the UltraSPARC-I

This section gives an overview of the front-end of the UltraSPARC processor which is the source of the observed performance anomalies. The UltraSPARC processor produced by Sun Microsystems contains the nine stage pipeline shown in Figure 3-3. The first three stages (Fetch, Decode and Group) constitute the ***front end*** of the processor, and are shown in detail in Figure 3-4. I identified the front end to be the source of all four performance anomalies, therefore I do not describe the remaining stages. In the front end, the instructions are loaded from memory, decoded, and then grouped together to be passed to the execution stage. It is essential for optimal performance that the front end maintain a rate of instructions flow, from the I-cache to the execution units, that is not smaller than that which the execution units can handle. If this rate cannot be maintained, the execution units become underutilized, and the processor performance drops. The instruction sequence determines which instructions must be fetched from the I-cache. Consequently, this sequence can have a significant effect on the performance of the front end. The following two sections describe the instruction buffer and the next field predictor. Understanding these features aids in the understanding of the observed performance anomalies.

Instruction Buffer

The UltraSPARC executes instructions at a maximum rate of 4 instructions per cycle. To move instructions from the I-cache to the execution units at this rate, an ***instruction-buffer (I-buffer)*** that holds 12 instructions is used, as shown in Figure 3-4. I assume the I-buffer is implemented

Figure 3-4: The design of the front end of the UltraSPARC.

as a circular buffer, managed by two pointers[2]. During each cycle, up to 4 instructions are loaded from the I-cache into the I-buffer. Simultaneously up to 4 instructions, previously loaded into the I-buffer, are passed to the grouping logic.

Next Field Predictor

The UltraSPARC associates a ***Next Field Predictor (NFP)*** with each half of an I-cache line. The NFP predicts the index of the *I-cache line* of the instruction to be scheduled for execution next, which allows the throughput of 4 instructions per cycle to be maintained when loading instructions from the I-cache in the presence of ***Control Transfer Instructions (CTI's)***. Each I-cache line contains 8 instructions, and separate NFP's are associated with both words 0–3 and words 4–7 of each I-cache line. I call each of these halves of an I-cache line an ***I-cache group***. The ***NFP value*** is the index used to predict the next I-cache line to be fetched. The NFP values are stored in the separate Next Field Prediction table shown in Figure 3-4, where each NFP value has an ***NFP slot*** in which it is stored.

The NFP value is determined by the instructions in its associated I-cache group, according to the following two cases:

Code without predicted-taken CTI's: The NFP indexes the I-cache line of the next instruction

[2]The UltraSPARC-I User's Manual tells us only that the buffer is managed by two pointers.

Figure 3-5: The NFP's for code without any predicted-taken CTI's point to the succeeding I-cache group.

in the code sequence. In Figure 3-5, the instruction to be scheduled for execution after the instructions in I-cache group A is the first instruction of I-cache group B. The NFP does not index the instruction to be scheduled for execution next, but to the *I-cache line* of the instruction to be scheduled for execution next. Thus, the NFP slot associated with group A holds the index of the I-cache line containing group B, which is i. Accordingly, the NFP slot associated with group B holds the index of I-cache line j.

Code with predicted-taken CTI's: The NFP addresses the I-cache line of the target of the CTI, as shown in Figure 3-6. The SPARC-V9 ISA used by the UltraSPARC includes a ***delay slot*** after all CTI's. The design of the UltraSPARC requires the instruction in the delay slot of a CTI always to be loaded, even if it will not be executed. Therefore, the NFP of an I-cache group indexes the target of the CTI if the I-cache group contains the *delay slot* of a CTI. If an I-cache group contains more than one CTI delay slot, the NFP slot associated with that group holds the target index of the last CTI taken.

The NFP is loaded as instructions are loaded into the I-buffer. I assume that the UltraSPARC architecture includes a ***NFP register*** into which the current NFP value is loaded, as shown in Figure 3-4. I call the group of instructions loaded into the I-buffer an ***I-buffer group***. This group can be any of up to 4 contiguous instructions of a *single* I-cache line, and may contain instructions from more than one I-cache group. Instructions from multiple I-cache lines cannot be loaded in a single cycle. The I-buffer group determines the NFP slot from which the current NFP value is loaded, according to the following two cases:

Figure 3-6: The NFP's for I-cache groups containing the *delay slot* of a predicted-taken CTI, point to the target of the CTI.

I-buffer group without the delay slot of any predicted-taken CTI's: The NFP value associated with the I-cache group that contains the first instruction in the I-buffer group is loaded into the NFP register.

I-buffer group containing the delay slot of a predicted-taken CTI: The NFP value associated with the I-cache group that contains the delay slot of the CTI is loaded into the NFP register.

When the I-cache is accessed next, only instructions of the I-cache line indexed by the currently loaded NFP value can be loaded. If the NFP value is mispredicted, a 2-cycle stall is caused while the correct I-cache line is fetched, and the correct instructions are loaded. As the correct instructions are loaded, the index of the correct I-cache line is loaded into the NFP slot of the associated I-cache group.

3.2 Next Field Predictor

The next field predictor is the first of the four features that I identified as the cause of a performance anomaly in the UltraSPARC. The next field contains the index of the I-cache line and the associativity number (or way) of the I-cache line that should be fetched next [14, p. 258]. The 2-cycle stall caused when the NFP is mispredicted produces the most pronounced performance anomaly.

The effect of next field prediction on performance is studied by means of the code fragment shown in Figure 3-7, which contains a loop with a call to the empty function `foo()`. The corresponding

```
        nop     ! 32-byte aligned
        nop
L0:     nop
        call foo
        add %l0,-1,%l0
        nop
        bg L0
        cmp %l0,0
         :
        nop     ! 32-byte aligned
foo:    retl
        nop
```

Figure 3-7: Assembly code fragment demonstrating the NFP misprediction problem.

instruction layout in the I-cache is shown in Figure 3-8, assuming that the code fragment in Figure 3-7 is aligned to a 32-byte (I-cache line) boundary.

The `call` instruction to `foo` occupies word 3 of the I-cache line, and its delay slot—occupied by the `add` instruction—word 4. The NFP value associated with the `call` target is stored in NFP slot B, because NFP slot B is associated with the I-cache group (words 4–7) that contains the delay slot of the `call`. The misprediction of the NFP value in slot B causes a performance anomaly, as explained in the following.

Assume that initially, all NFP values in Figure 3-8 are undefined.[3] When entering loop `L0`, the instructions in I-cache group 0–3 of line i are loaded into the I-buffer, and the `call` instruction is scheduled for execution. After the associated next field prediction, the index of I-cache line i is assigned to NFP slot A, because the delay slot of the `call` (word 4) must be loaded next. When scheduling the delay slot for execution, the value for NFP slot B is computed. This value is the index of I-cache line j, which contains the code of function `foo`. Analogously, when executing the `retl` instruction of `foo`, the value for NFP slot C is computed, which is the index of I-cache line i. The state of the NFP's when returning from `foo` is shown in Figure 3-8.

The cause of the performance anomaly is a misprediction of the NFP values both on the return from, and the call to `foo`:

1. After returning from `foo`, the NFP values are the values shown in Figure 3-8 and instruction `bg L0` is scheduled for execution. This instruction branches to word 2 of the same I-cache line, i. Consequently, the next I-buffer group to be fetched originates from line i. The predicted I-cache line index in the associated NFP slot B is line j, however. This misprediction causes a 2-cycle stall, during which the index of the correct I-cache line, which is i, is computed and stored in NFP slot B.

[3]It is not clear to me how the NFP values are initialized in the NFP table.

Figure 3-8: State of NFP's when `foo` returns.

2. The second misprediction occurs during execution of the next call to `foo`. The just-corrected value in NFP slot B, associated with the delay slot of the call, is now incorrect, because the code of `foo` is cached in line j. Consequently, another 2-cycle stall occurs, during which the index of I-cache line i is assigned to NFP slot B.

Each execution of the call to `foo` in I-cache line i causes 2 mispredictions and 4 lost cycles due to stalls.

Performance Nonmonotonicity Bug

Next field mispredictions can be exposed as performance nonmonotonicities. In Figure 3-9, I added work in the form of 3 `add` instructions in front of loop `L0` from Figure 3-7. Although work is added, the execution of the code in Figure 3-9 can be up to 2.25 times faster than the execution of the original code in Figure 3-7, cf. Table 3.1.

Bug Fix

The performance nonmonotonicity caused by next field misprediction can be concealed by realigning the instructions such that the delay slot of the call to `foo` occupies the last word of an I-cache group, which is either word 3 or word 7 of an I-cache line.

Next Field Prediction Fix: Align all `call` instructions to word 2 or word 6 of an I-cache line.

```
        nop     ! 32-byte aligned
        nop
        add %l0,+2,%l0
        add %l0,-1,%l0
        add %l0,-1,%l0
L0:     nop
        call foo
        add %l0,-1,%l0
        nop
        bg L0
        cmp %l0,0

        ...

foo:    retl
        nop
```

Figure 3-9: Assembly code fragment similar to Figure 3-7, that does not exhibit next field misprediction.

Performance	Figure 3-7 (original)	Figure 3-9 (rescheduled)
Time (secs)	5.6 – 6.3	2.8
Cycles/Iteration	8 – 9	4

Table 3.1: Performance of the NFP misprediction microbenchmarks in Figure 3-7 and Figure 3-9.

In the example in Figure 3-7, the mispredictions are independent of the loop around the function call to `foo`. The next I-cache line is predicted correctly *only* if the function call is allocated to word 2 or 6 of an I-cache line.

Table 3.1 compares the performance of the loop shown in Figure 3-7 with its aligned version shown in Figure 3-9. The data in the table was obtained by executing one-hundred million iterations of each loop on a Ultra-SPARC-I (143 MHz) machine. Realigning the code conceals the nondeterminism observed in the original code, however, the source for the nondeterminism could not be found.

3.3 Fetching Logic

The fetching logic of the UltraSPARC determines the rate at which instructions are fetched from the I-cache. Performance degrades if the rate at which instructions are fetched is smaller than the rate at which they can be executed. The fetching logic allows instructions from only a *single* I-cache line to be fetched in each cycle. I call this restraint the ***fetching limitation***. This limitation implies that “When the fetch address mod 32 is equal to 20, 24, or 28, then three, two or one instruction(s)

```
    L1:  nop00
A        br L2:          ! 32-byte aligned
         nop01
         nop02
         nop03
         nop04
         nop05
         nop06
    L2:  nop07
B        br L3:          ! 32-byte aligned
         nop08
         nop09
         nop10
         nop11
         nop12
         nop13
    L3:  cmp %l0,0
C        bg L1:          !32-byte aligned
         add %l0,-1,%l0
```

Figure 3-10: Assembly code demonstrating the Fetching Limitation Problem.

respectively will be added to the instruction buffer" [14], rather than four.

I define an ***execution group*** to be a group of up to 4 consecutive instructions that can be scheduled for execution during a single cycle. Execution groups that cross I-cache (32-byte) boundaries expose the fetching limitation. Loading such execution groups into the I-buffer takes 2 cycles, because instructions from 2 different I-cache lines are loaded, and each load takes 1 cycle.

The code fragment in Figure 3-10 contains three execution groups, A, B, and C. Figure 3-11 shows one possible alignment of the code in the I-cache. Due to the branch instructions, only the bracketed code in Figure 3-10 (highlighted code in Figure 3-11), is actually executed. Executing 100,000,000 iterations of the loop shown in Figure 3-10 takes 2.1 seconds on an UltraSPARC-I (143MHz) machine when it is aligned in the I-cache as shown in Figure 3-11. Each iteration of the loop executes in 3 cycles[4]. Consequently, during each cycle the instructions of one of the execution groups, A, B, or C, must be loaded from the I-cache. The branch target prediction mechanism of the UltraSPARC allows it to predict the target of branches *before* they are passed to the grouping logic [14, pg. 13]. I assume the branches of the code fragment are always predicted correctly, since each iteration of the loop executes in 3 cycles.

Realigning the code in the I-cache, we produce the instruction sequence shown in Figure 3-12. Since each execution group in this alignment spans 2 I-cache lines, each requires 2 cycles to be loaded from the I-cache. Consequently, at least 6 cycles are required each iteration to load the instructions. Since the branch target prediction logic predicts all branches correctly, *exactly* 6 cycles are required.

[4] (143,000,000 cycles per sec/100,000,000 iterations) $\cdot$ 2.1 sec = 3 cycles/iteration

Figure 3-12: I-cache alignment of the code in Figure 3-10 that executes in 6 cycles per iteration.

Performance Nonmonotonicity Bug

The assembly code in Figure 3-10 reveals a performance nonmonotonicity caused by the fetching limitation. If the code is aligned as shown in Figure 3-12, inserting a single `add` instruction before `L1` will realign the loop to the alignment shown in Figure 3-11. This example reveals a performance nonmonotonicity, since as stated earlier, the code aligned as in Figure 3-11 executes in half the time of the code as aligned in Figure 3-12.

Bug Fix

Some instruction sequences that exhibit this bug can be avoided by realigning instructions using the following algorithm.

Fetching Limitation Fix: Insert `nop`s to align to 32-byte boundaries all basic blocks immediately

Number Crossing an I-cache Line Boundary	Time(secs)	Cycles/Iteration
0	2.1	3
1	2.8	4
2	3.5	5
3	4.2	6

Table 3.2: Performance of the Fetching Limitation microbenchmark shown in Figure 3-10, with four different alignments of the code in the I-cache.

following unconditional CTI's.

Execution groups in the instruction sequence that cross I-cache line boundaries must be avoided to conceal the fetching limitation performance anomaly. To avoid such execution groups, we would like to realign execution groups to I-cache line boundaries. Realigning instructions in the cache, however, requires instructions to be inserted into the code. If these instructions are executed, then we may lose the single cycle gained by avoiding an execution group that spans I-cache lines. Additionally, if the fetching of instructions is not the execution bottleneck, then inserting instructions in the code may decrease performance by requiring more instructions to be executed. Instructions inserted immediately after the delay slot of an unconditional CTI (such as a branch always or a jump), execute only if they are the target of a branch. Therefore, any instructions inserted after the delay slot of an unconditional CTI, and before the following label, can never be executed. Only at this place can we insert instructions and know they can never be executed. Consequently, the algorithm realigns only execution groups immediately following the delay slot of an unconditional CTI. Without information on the control flow of the program, it seems this is the best that can be done. By using the compiler to conceal the fetching limitation performance nonmonotonicity bug, however, information on control flow would be available and a better fix could be implemented.

The effect of aligning execution groups A, B, and C in the I-cache is shown in Table 3.2. Inserting a `nop` before `L3`, group C is aligned to a 32-byte boundary. Similarly, inserting a `nop` before `L2`, realigns groups B and C, and inserting a `nop` before `L1` realigns all three groups simultaneously. One-hundred million iterations of the loop are executed on an Ultra-SPARC-I running at 143 MHz. The results show clearly that whenever an execution group spans I-cache lines, an additional cycle of execution time is added to each iteration of the loop.

3.4 Grouping Logic

The design of the Ultra-SPARC-I includes two floating-point execution units, and two integer execution units. Each integer unit can only execute a subset of the integer instructions, however, and each floating-point unit can only execute a subset of the floating-point instructions [14]. This design restricts which instructions of an instruction stream can be scheduled for execution during the same cycle. The grouping logic ensures that none of the restrictions are violated when instructions are passed to the execution units. Other limitations, such as data dependencies, also restrict which instructions can be scheduled for execution during the same cycle. When a limitation exists between two instructions presented to the grouping logic in the same cycle, it a forces a ***group break*** between the two instructions. All instructions following the group break are scheduled for execution in later cycles.

The grouping logic itself imposes a constraint on the instruction stream, as stated in the UltraSPARC-I User's Manual [14]:

> "UltraSPARC-I can execute up to 4 instructions per cycle. The first 3 instructions in a group occupy slots that are interchangeable with respect to resources. [...] The fourth slot can only be used for PC-based branches or for floating-point instructions."

I call this the ***grouping constraint***. If this constraint is violated by the instruction stream, a group break forms before the instruction in the "fourth slot", since it cannot be scheduled for execution in the same cycle as the other instructions.

```
L1:     fmuls %f4,%f4,%f4   ┐
        nop                 │
        nop                 │
        fmovs %f7,%f7       ┘
- - - - - - - - - - - - - - -
        fmovs %f5,%f5       ┐
        nop                 │
        add %l0,-1,%l0      │
        fmuls %f6,%f6,%f6   ┘
- - - - - - - - - - - - - - -
       *nop                 ┐
        cmp %l0,0           │
        bg L1               │
       *fmovs %f9,%f9       ┘
```

Figure 3-13: Assembly code loop which is executed at the maximum execution rate of 4 instructions per cycle.

Figure 3-13 shows an instruction sequence for which instructions are executed at the maximum execution rate of 4 instructions per cycle. The brackets show the execution grouping of the instructions for which the execution rate of 4 instructions per cycle can be achieved. All of the group breaks in this instruction sequence are "normal" groups breaks, and not "forced" group breaks.

```
L1:     fmuls %f4,%f4,%f4
        nop
1 ---------------
        nop
        fmovs %f7,%f7
2 ---------------
        fmovs %f5,%f5
        nop
        add %l0,-1,%l0
        fmuls %f6,%f6,%f6
3 ---------------
        fmovs %f9,%f9
        cmp %l0,0
        bg L1
4 ---------------
        nop
```

Figure 3-14: Assembly code loop produced by exchanging two instruction in the code in Figure 3-13, which cannot execute at the rate of 4 instructions per cycle because of the grouping limitation.

The code sequence shown in Figure 3-14 can be produced by exchanging the instructions marked with a star in Figure 3-13. In Figure 3-14, each of the group breaks is numbered and marked with a dashed line. Groups breaks 1, 2, and 4 are "forced" group breaks, while group break 3 is a "normal" group break. In the following, I explain the cause of each group break:

1. Both the last instruction in the whole sequence, a `nop`, and the first 2 instructions, an `fmuls` and a `nop`, are in the execution group preceding this break. This preceding group contains 2 integer instructions—the `nops`—and only 2 integer instructions can be executed each cycle. Since the first instruction in the group following the break is also an integer instruction, it cannot be scheduled for execution in the same cycle, and a group break forms.

2. Only one of the floating-point units can execute `fmovs` instructions, so only one `fmovs` instruction can be executed each cycle. Consequently, a group break forms between the two fmovs instructions.

3. Only 4 instructions can be scheduled for execution each cycle, forming the group between group breaks 2 and 3. Group break 3 does not represent a forced group break, but rather a normal grouping with maximum utilization of the functional units.

4. The *grouping limitation* dictates that if 4 instructions are to be scheduled for execution during the same cycle, the 4th instruction must be either a floating-point instruction or a PC-based branch instruction. The instruction in the 4th slot is neither a floating-point operation nor a branch, but a `nop`. Consequently only 3 instructions can be scheduled for execution in the execution group preceding this break.

Given the grouping shown in Figure 3-14, each iteration of the loop takes 4 cycles rather than 3.

Performance Nonmonotonicity Bug

The grouping limitation causes performance nonmonotonicities. Exchanging the position of two instructions in the code sequence in Figure 3-13 to produce the code in Figure 3-14 increases the execution time by 1/3. The amount of work has not increased, although the execution time has, thereby revealing a performance nonmonotonicity.

Bug Fix

The grouping limitation performance nonmonotonicity is concealed by ensuring that, if possible, the 4th slot of every execution group contains either a floating-point operation or a CTI. Starting with the code fragment in Figure 3-14, we can conceal the bug by rescheduling the instructions to produce the code fragment in Figure 3-13. A general restructuring algorithm to conceal this performance nonmonotonicity was not developed, due to the complexity of the other limitations imposed by the architecture. It would be more appropriate to implement such an algorithm as part of a compiler, where more control flow information is accessible.

Performance	Code in Figure 3-13	Code in Figure 3-14
Time (secs)	2.1	2.8
Cycles/Iteration	3	4

Table 3.3: Performance of the grouping limitation microbenchmark in Figure 3-13 compared to the microbenchmark in Figure 3-14.

Table 3.3 compares the performance of the code in Figure 3-13 to the code in Figure 3-14 when it is run on a SUN UltraSPARC-I (143 MHz) machine. This table shows that each iteration of the loop in Figure 3-13 can be executed in 3 cycles, whereas each iteration of the code in Figure 3-14 takes 4 cycles.

3.5 Branch Prediction Logic

A 2-bit branch prediction mechanism is used on the UltraSPARC [14]. A single 2-bit predictor is associated with every two instructions. The branch prediction logic exhibits a performance anomaly called ***odd-fetch*** which is explained in the Ultra-SPARC User's Manual [14]:

> "When the target of a branch is word one or word three of an I-cache line, and the 4th instruction to be fetched is a branch, the branch prediction bits from the wrong pair of instructions are used."

```
        nop                  ! 32-byte aligned
L1:     cmp %g1, 0
        add %g2,1,%g2
        add %g5,1,%g5
        bg,a,pt %icc, L1
        sub %g1,1,%g1
```

Program **A**

```
        add %l0,0,%l0    ! 32-byte aligned
        nop
L1:     cmp %g1, 0
        add %g2,1,%g2
        add %g5,1,%g5
        bg,a,pt %icc, L1
        sub %g1,1,%g1
```

Program **B**

Figure 3-15: Assembly code fragments demonstrating the odd-fetch performance anomaly.

The microbenchmark shown in Figure 3-15 program **A** exhibits the branch misprediction caused by the odd-fetch problem. In this example, the `bg` instruction is the 3rd instruction after the branch target `L1`. Thus, when the `bg` instruction is fetched, it is the 4th instruction to be fetched. Additionally, the branch target `L1` is word 1 of an I-cache line, assuming that the `nop` before `L1` is aligned to a 32-byte boundary.

Performance Nonmonotonicity

The assembly code in Figure 3-15 exposes the odd-fetch performance nonmonotonicity. Inserting an extra `add` instruction before the loop in Figure 3-15 program **A** produces the code in Figure 3-15 program **B**. In program **B**, the loop is aligned such that the target of the branch is no longer the 1st or 3rd word in an I-cache line, but instead the 2nd word. This addition of work decreases the execution time by a factor of 3 or more, exhibiting nonmonotonic performance.

Bug Fix

To conceal the odd-fetch performance nonmonotonicity problem, we avoid all instances of the instruction sequence described in the citation from the UltraSPARC User's Manual.

Odd-Fetch Fix: Insert a single `nop` before all branch targets that are word 1 or word 3 of an I-cache line, and for which the 4th instruction, word 5 or word 7 respectively, is a branch.

By avoiding the problematic instruction sequence described in the User's Manual, we prevent the use of the wrong branch prediction bits, and conceal the odd-fetch performance nonmonotonicity.

Performance	Program **A** in Figure 3-15	Program **B** in Figure 3-15
C-code	4.2 - 4.9 seconds	1.4 seconds
Assembly Code	6 - 7 cycles/iteration	2 cycles/iteration

Table 3.4: Performance of the odd-Fetch microbenchmarks in Figure 3-15.

Table 3.4 compares the performance of the two versions of the code in Figure 3-15 when they are executed on an UltraSPARC-I (143 MHz). Realigning the code also conceals the nondeterminism exhibited by Program **A**. Again, the source for the nondeterminism in the unaligned code could not be found, however.

Chapter 4

Experimental Results

To show that performance anomalies appear not only in contrived examples, but also in real programs, I implemented an assembly-code restructuring tool. This tool implements the algorithms described in Chapter 3 for concealing performance anomalies. I show that using this tool during the compilation of the SPECint benchmarks [23] provides speedups of up 2.2% on average across compilers[1]. I first describe the experimental setup, and then present and discuss the performance gains of restructuring.

The Experiment

The instruction sequences of the SPECint benchmarks were restructured at the assembly-code level. I implemented a restructurer that uses the algorithms described in Chapter 3 for aligning instructions by means of inserting `nops`. To simplify the process of aligning instructions in the assembly file, all functions are aligned to I-cache line (32-byte) boundaries with the assembly code macro `.align`.

The two compilers used in this study are the GNU C-compiler version 2.7.2 (gcc), and the SUN Workshop Compiler version 4.2 (cc). With each compiler I created a set of fully optimized executables and a set of executables that excluded function inlining. For gcc, I used -O3 to build the fully optimized executables and -O2 to build the executables without inlining. For the GNU compiler the -O3 option also includes the loop unrolling optimization, in addition to procedure inlining. To build the fully optimized executable with cc, I used the same options used by SUN to obtain the published SPECint results for the UltraSPARC: -Xc -xarch=v8 -xchip=ultra -fast -xO4 -xdepend. The executables without function inlining were built using the option -xO3 in place of -xO4. In the following, I use the notation ***ccO4***, ***ccO3***, ***gccO3***, and ***gcc O2*** to denote the four sets of compiled executables. The instrumented executables were obtained by compiling the benchmarks with -S

[1]The 2.2% given is obtained by averaging the 2.1% achieved on executables compiled with SUN's C-compiler using full optimization and the 2.3% achieved using GNU's C-compiler with full optimization.

to produce an assembly file. Subsequently, my restructurer was used to produce an instrumented assembly file. Finally, the instrumented assembly files were assembled using the compiler they were originally created with. Fortunately, SUN's and GNU's assemblers don't treat the assembly code as another intermediate representation. Rather than performing optimizations during the assembly phase, as SGI's C-compiler does, these assemblers perform a one-to-one translation of assembly mnemonic into bit strings. This allows my restructurer to correctly align code at the assembly-code level.

All experiments were performed on SUN UltraSPARC-I machines running at 142 MHz. The machines have 64 MB memory and run Solaris 2.5.1. In order to reduce the variance of the execution times, produced by the operating system, the machine was booted in single user mode. All results were obtained by rebooting the machine, running each executable three times, rebooting the machine again, running each executable 3 additional times and selecting the smallest of the six execution times. I observed that 3 executions was enough to allow the performance to stabilize, and picking the lowest of 6 executions reduced the variance to between 1 and 2 percent.

The Results

This section contains the results obtained for each of the four optimization levels—ccO4, ccO3, gccO3, gccO4—and a brief discussion of each set of results. Table 4.1 contains the results for ccO4, Table 4.2 the results for ccO3, Table 4.3 the results for gccO3, and Table 4.4 the results for gcc O2. Each table presents execution times in the upper section, and performance increases in the lower section. Performance is represented by SPECMARKS. Therefore, *higher SPECMARK numbers represent lower execution times.* Performance increases are represented by percentages.

Five sets of SPECMARKS were obtained for each optimization level and are presented in the 5 columns of the upper part of each table. In each table, the *original (unaligned)* column shows the SPECMARKS achieved by the original instrumented executables[2]. The columns *next field misprediction, fetching limitation*, and *odd fetch* present SPECMARKS of the executables restructured to conceal the respective performance anomaly. The *all* column presents SPECMARK data for executables restructured to conceal all three performance anomalies. The lower part of each table presents the performance increases calculated from the SPECMARKS. Each column shows the difference in performance between the original executable and the executables restructured to conceal the performance anomalies.

[2]These results differ from SUN's published results because a different machine setup was used.

Sun C-compiler using -xO4

The executables built with ccO4 produce the results shown in Table 4.1. The maximum performance increase obtained by concealing performance anomalies is the 5.5% increase obtained for `li`. The SPECint rating of 4.75 is the highest rating achieved with the optimization levels I tested. Additionally, restructuring these executables produced consistent speedups, with performance increases on all benchmarks except `vortex`. Surprisingly, the `m88ksim` benchmark produced negative results for concealing individual performance anomalies, but obtains a performance gain of 3.4% when all performance anomalies are concealed.

Sun C-compiler using -xO3

The results obtained using ccO3 are shown in Table 4.2. The maximum performance increase is the 6.7% increase for `go`. The performance increases are generally higher than those obtained using ccO4. I assume that inlining function calls partially conceals the next field misprediction performance anomaly by reducing the number of function calls. Therefore, executables without inlined functions calls present a greater opportunity for performance increase due to the concealment of the next field misprediction performance anomaly. We can see this by comparing the results of individual benchmarks in Table 4.2 to those in Table 4.1. For example, concealing next field misprediction achieves only a 0.6% performance increase for the benchmark `gcc` compiled using ccO4. A performance increase of 4.1% is obtained when `gcc` is compiled using ccO3, however. It is not clear, though, why all of the benchmarks do not produce similar speedups.

GNU C-compiler using -O3

The results obtained using gccO3 are shown in Table 4.3. The largest performance gain is the 8.9% for `perl`. This performance increase is the largest I obtained for the entire SPECint suite. Additionally, a increase of 2.3% was obtained for the entire suite which was the highest SPECint rating increase that I obtained. The lowest performance increase obtained for executables compiled with the GNU C-compiler using -O3 is -1.6% for `compress`, which is surprising since a performance increase of 5.4% was obtained when this benchmark was compiled with SUN's C-compiler using -xO4. Results such as these indicate that performance anomalies depend heavily on the compiler used.

GNU C-compiler using -O2

The results obtained using the GNU C-compiler with the -O2 option are shown in Table 4.4. The highest performance increase obtained is the 2.5% increase for `m88ksim`. Surprisingly, the performance gains are the lowest of all compilations tested. Concealing individual performance anomalies

SPECMARKS					
Benchmark	original (unaligned)	next field misprediction	fetching limitation	odd fetch	all
go	5.71	5.26	5.74	5.45	5.75
m88ksim	4.14	4.13	3.67	3.98	4.28
gcc	5.06	5.09	5.07	5.00	5.08
compress	5.21	5.23	5.28	5.28	5.49
li	3.97	4.15	4.14	4.08	4.19
ijpeg	5.04	4.99	5.09	4.84	5.05
perl	4.57	4.65	4.58	4.48	4.74
vortex	4.55	4.58	4.60	4.54	4.49
SPECint Rating	4.75	4.74	4.73	4.68	4.85

PERCENTAGE INCREASE IN PERFORMANCE				
Benchmarks	next field misprediction	fetching limitation	odd fetch	all
go	-7.8	0.5	-4.6	0.7
m88ksim	-0.2	-11.3	-3.9	3.4
gcc	0.6	0.2	-1.2	0.4
compress	0.2	1.3	1.3	5.4
li	4.5	4.3	2.8	5.5
ijpeg	-1.0	1.0	-4.0	0.2
perl	1.8	0.2	-2.0	3.7
vortex	0.7	1.1	-0.2	-1.3
SPECint Rating	-0.2	-0.4	-1.5	2.1

Table 4.1: Concealing performance anomalies in executables compiled with SUN cc -xO4 provided performance increases of up to 5.5% for the SPECint benchmarks, as shown in the *all* column of the performance increase section. All execution times are given in SPECMARKS, for which higher number represent better performance. All performance increases are given as percentages.

SPECMARKS					
Benchmark	original (unaligned)	next field misprediction	fetching limitation	odd fetch	all
go	5.36	5.41	5.55	5.17	5.72
m88ksim	3.95	3.73	3.91	3.97	3.95
gcc	4.81	5.01	5.05	4.75	5.11
compress	4.94	4.74	4.80	4.75	4.83
li	3.48	3.46	3.57	3.49	3.54
ijpeg	4.62	4.68	4.65	4.68	4.67
perl	4.27	4.31	4.48	4.28	4.49
vortex	4.43	4.38	4.55	4.51	4.36
SPECint Rating	4.45	4.42	4.53	4.42	4.54

PERCENTAGE INCREASE IN PERFORMANCE				
Benchmarks	next field misprediction	fetching limitation	odd fetch	all
go	0.9	3.5	-3.5	6.7
m88ksim	-5.6	-1.0	0.5	0.0
gcc	4.1	5.0	-1.2	6.0
compress	-4.0	-2.8	3.8	-2.2
li	-0.6	2.6	0.3	1.7
ijpeg	1.3	0.6	1.3	1.1
perl	0.9	4.9	0.2	5.1
vortex	-1.1	2.7	1.8	-1.6
SPECint Rating	-0.7	1.8	-0.7	2.0

Table 4.2: Concealing performance anomalies in executables compiled with SUN cc -xO3 provided performance increases of up to 6.7% for the SPECint benchmarks, as shown in the *all* column of the performance increase section. All execution times are given in SPECMARKS, for which higher number represent better performance. All performance increases are given as percentages.

SPECMARKS					
Benchmark	original (unaligned)	next field misprediction	fetching limitation	odd fetch	all
go	5.49	5.31	5.42	5.20	5.60
m88ksim	3.86	3.86	3.99	4.08	4.07
gcc	5.01	5.03	5.02	4.99	5.02
compress	5.10	4.99	4.84	4.75	5.02
li	3.40	3.41	3.42	3.44	3.44
ijpeg	3.30	3.31	3.40	3.41	3.33
perl	4.36	4.37	4.60	4.65	4.75
vortex	4.25	4.46	4.11	4.25	4.37
SPECint Rating	4.28	4.28	4.99	4.30	4.38

PERCENTAGE INCREASE IN PERFORMANCE				
Benchmarks	next field misprediction	fetching limitation	odd fetch	all
go	-3.27	-1.3	-5.3	2.0
m88ksim	0.0	3.4	5.7	5.4
gcc	0.4	0.2	-0.4	0.2
compress	-2.2	-5.1	-6.9	-1.6
li	0.3	0.6	1.2	1.2
ijpeg	0.3	3.0	3.3	0.9
perl	0.2	5.5	6.7	8.9
vortex	4.9	-3.3	0.0	2.8
SPECint Rating	0.0	0.2	0.5	2.3

Table 4.3: Concealing performance anomalies in executables compiled with GNU gcc -O3 provided performance increases of up to 8.9% for the SPECint benchmarks, as shown in the *all* column of the performance increase section. All execution times are given in SPECMARKS, for which higher number represent better performance. All performance increases are given as percentages.

SPECMARKS					
Benchmark	original (unaligned)	next field misprediction	fetching limitation	odd fetch	all
go	5.28	5.17	5.43	5.30	5.35
m88ksim	4.06	4.16	4.00	4.11	4.09
gcc	4.95	4.98	4.96	4.97	4.96
compress	5.09	4.99	4.83	4.77	4.99
li	3.38	3.38	3.44	3.46	3.43
ijpeg	3.27	3.29	3.40	3.39	3.33
perl	4.47	4.55	4.03	4.42	4.44
vortex	4.42	4.32	3.97	4.11	4.43
SPECint Rating	4.30	4.30	4.20	4.27	4.32

PERCENTAGE INCREASE IN PERFORMANCE				
Benchmarks	next field misprediction	fetching limitation	odd fetch	all
go	-2.1	2.8	0.4	1.3
m88ksim	4.3	0.3	3.0	2.5
gcc	0.6	0.2	0.4	0.2
compress	-2.0	-5.4	-6.7	-2.0
li	0.0	1.8	2.4	1.5
ijpeg	0.6	4.0	3.7	1.8
perl	1.8	-9.8	-1.1	-0.6
vortex	-2.2	-10.2	-7.0	0.2
SPECint Rating	0.0	-2.3	-0.7	0.4

Table 4.4: Concealing performance anomalies in executables compiled with GNU gcc -O2 provided performance increases of up to 2.5% for the SPECint benchmarks, as shown in the *all* column of the performance increase section. All execution times are given in SPECMARKS, for which higher number represent better performance. All performance increases are given as percentages.

even tends to produce negative results. I assume, the reason is that the code produced by this optimization level is less optimized than the code produced by the other optimization levels. As shown in Chapter 3, concealing performance anomalies avoids situations for which the execution bottleneck is the flow of instructions from the I-cache to the execution units. If the code is compiled such that other parts of instruction execution are the bottleneck, however, then concealing performance anomalies will not produce any increase in the performance.

Discussion of the Results

The experimental results show that concealing performance anomalies through assembly code restructuring leads to performance gains of up to 8.9% compared to the original executables. The results also indicate however, that effects outside of my study make the results somewhat unpredictable. I believe, though, that this unpredictability can be explained by the combined effects of the operating system and performance anomalies.

Significant effort was spent to produce deterministic execution times. All measurements were conducted in single user mode, and the machine was rebooted between each run of 3 executions. Running executables without these precautions produced variances of the execution time that were greater than the performance increases obtained. By rebooting before each execution, and executing in single user mode, however, I was able to reduce the variance to a tolerable 1 to 2 percent. I believe this observed variance was caused by the operating system, and specifically it's management of the memory pages. This 1 to 2 percent variance helps to explain some of the smaller inconsistencies observed in the results.

It is not possible to completely isolate the effect of concealing a single performance anomaly using my restructurer. The restructuring policy of inserting `nops` affects the performance of the code in two unpredictable ways. First, it is possible that inserting a `nop` to conceal only one of the observed performance anomalies may inadvertently conceal or reveal another of the observed performance anomalies. Additionally, it seems likely that the UltraSPARC contains many performance anomalies that I have not identified. Inserting `nops` to conceal the observed performance anomalies may conceal or reveal these unknown performance anomalies as well contributing to the unpredictability of the results.

A few observations can be explained by assuming that the results presented do not completely isolate the effect of concealing individual performance anomalies. First, we observe that concealing the odd-fetch performance anomaly does not seem to improve the overall performance of executables compiled with the optimization levels tested. This may be partially caused by the fact that the odd-fetch performance anomaly occurs with a frequency of approximately 1/10 that of the other performance anomalies. Additionally, by inserting a single `nop` to conceal an odd-fetch, we realign all of the following instructions. This realignment may reveal or conceal many occurrences of next field misprediction, and of the fetching limitation. I believe the variance in the performance produced by this revealing and concealing hides the performance gains of concealing the odd-fetch performance anomaly. Additionally, we observe that the performance increases obtained by individually concealing the three observed performance anomalies do not accumulate to produce the performance gained by concealing all three performance anomalies together. This is because, with my restructurer, it is difficult to isolate the effect of concealing individual performance anomalies.

The SPECint benchmarks were chosen because of their acceptance as a benchmarking standard. I believe that similar, and possibly greater, performance can can be obtained for programs that contain tight loops, such as numerical codes.

Chapter 5

Conclusion

This thesis has presented the problem of processor performance anomalies. Current microprocessors are so complex that there exist situations for which the insertion of a single instruction can affect the execution time of a program by a factor of 3 or 4. I call situations like this performance anomalies. In this thesis I provide five contributions:

1. I have identified the *problem* of performance anomalies in commercial microprocessors.
2. I have provided a detailed *description* of the cause of four performance anomalies observed on the SUN UltraSPARC.
3. I have provided *algorithms* used to conceal each of the anomalies.
4. I have developed an assembly-code *restructuring tool* which implements the algorithms.
5. I have shown that through the use of my restructuring tool, *performance gains* of up to 9% can be obtained on some of the SPECint benchmarks.

In short, I have shown that current microprocessors contain performance anomalies and that significant performance gains can be obtained by concealing them. This work identifies what I believe to be the most important set of performance problems in the design of current microprocessors.

We take it for granted that a correct program will produce correct results when run on current microprocessors, yet we tend to accept the fact that it is near impossible to predict the execution time. Programmers who desire high performance must rely on a high degree of performance predictability, just as all programmers rely on the guarantee of correctness. The correctness of results can only be guaranteed with such a high degree of certainty, however, because processors are being designed for testability. I propose that an analogous philosophy be applied to the problem of performance anomalies. I believe that the only way to fully *avoid* performance anomalies is for microprocessor designers to *design for performance monotonicity.*

Until microprocessors are designed with monotonic performance, however, we can use the approach presented in this thesis of *concealing* performance anomalies. The use of an assembly-code restructuring tool has shown that appropriate instruction scheduling can conceal performance anomalies. The restructuring method of realigning code through the insertion of `nop`s has many problems, however. The addition of `nop`s increases both the total size of the executable file and the number of instructions that are executed. Additionally, the insertion of a single `nop` changes the scheduling of all following instructions. This makes it difficult to isolate the effect of concealing individual performance anomalies. I believe these problems can be avoided by implementing algorithms for concealing performance anomalies as part of the instruction scheduling phase of a compiler. The availability of control flow information would allow for proper alignment of instructions possibly without the insertion of `nop`s. The implementation of algorithms to conceal performance anomalies within the compiler would avoid both the increase in executable size and the increase in the number of executed instructions, as well as isolate the effect of concealing individual performance anomalies. This implementation must be done by the compiler writers, however, and I hope that this thesis provides sufficient evidence to motivate the inclusion of such algorithms within the compiler.

Until we can rely on the compiler writers, however, performance anomalies can be concealed through code restructuring. Unfortunately, my method of restructuring is limited by its reliance on access to the assembly code output by the compiler. I chose to restructure assembly code rather than binaries because of the reduced implementation effort. Implementing instruction rescheduling algorithms as part of a *binary restructurer*, however, would allow the rescheduling of code for which the assembly code is not available. This type of restructuring would allow anyone using an UltraSPARC machine to obtain performance increases such as the ones presented in this thesis. I believe the performance gains presented in this thesis justify the additional effort for implementing a binary restructurer.

The algorithms presented in this thesis conceal performance anomalies of the SUN UltraSPARC, and are presumably not applicable to other processors. Krech [10] has shown that similar performance anomalies exist on the Pentium, however, and I believe that the design of *all* modern microprocessors includes performance nonmonotonicities. I hope that my explanation of the performance anomalies on the UltraSPARC sparks the discovery of performance anomalies on other processors. If, as I suspect, all modern microprocessors do include such anomalies, this lends even more support to my belief that future microprocessors must be *designed for performance monotonicity*.

Bibliography

[1] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffery Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St. Malo. France, October 1997.

[3] Mark Atkins and Ramesh Subramaniam. PC software performance tuning. *IEEE Computer*, pages 47–54, August 1996.

[4] David H. Bailey. RISC microprocessors and scientific computing. In *Proceedings of the Conference on High Performance Computing and Communications*, pages 645–654, November 1993.

[5] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain program running in a paging machine. *Communications of the ACM*, 12(6):349–353, June 1969.

[6] Bryan Black and John Paul Shen. Calibration of microprocessor performance models. *IEEE Computer*, pages 59–65, May 1998.

[7] Pradip Bose and Thomas M. Conte. Performance analysis and its impact on design. *IEEE Computer*, pages 41–49, May 1998.

[8] Philip Heidelberger and Stephen Lavenberg. Computer performance evaluation methodology. *IEEE Transactions on Computers*, C-33(12):1195–1220, December 1984.

[9] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[10] Michael Krech, February 1998. `http://www.intelligentfirm.com`.

[11] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[12] Peter S. Magnusson and Johan Montelius. Performance debugging and tuning using an instruction-set simulator. Technical Report SICS-T–97/02-SE, Swedish Institute of Computer Science, June 1997.

[13] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[14] Sun Microelectronics. *STP1030BGA UltraSPARC-I User's Manual.* Sun Microsystems, Mountainview, CA, 1996.

[15] Matt Reilly and John Edmondson. Performance simulation of an Alpha microprocessor. *IEEE Computer*, pages 50–58, May 1998.

[16] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brad Chen, and Brian Bershad. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, August 1997.

[17] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, Fall 1995.

[18] Rafael Saavedra-Barrera, Alan Smith, and Eugene Miya. Machine characterization based on an abstract high-level language machine. *IEEE Transactions on Computers*, 38(12):1659–1679, December 1989.

[19] Eric Schnarr and James R. Larus. Instruction scheduling and executable editing. In *Proceedings of the Workshop on Compiler Support for System Software*, February 1996.

[20] Ashok Singhal and Aaron J. Goldberg. Architectural support for performance tuning: A case study on the SPARCcenter 2000. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[21] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of SIGPLAN '94*, Orlando, Florida, June 1994.

[22] Sun Microsystems. The UltraSPARC processor – technology white paper. `http://www.sun.com/microelectronics/whitepapers/UltraSPARCtechnology/`.

[23] The Standard Performance Evaluation Corporation. `http://www.specbench.org/osg/spec95`.

[24] The Standard Performance Evaluation Corporation, September 1997. Theme Feature: One-billion transistor microprocessors.

[25] Sivan Avraham Toledo. *Quantitative Performance Modeling of Scientific Computations.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1995.

[26] G. Michael Uhler, Debra Bernstein, Larry L. Biro, John F. Brown III, John H. Edmondson, Jeffrey D. Pickholtz, and Rebecca L. Stamm. The NVAX and NVAX+ high-performance VAX microprocessors. *Digital Technical Journal*, 4(3):1–19, summer 1992.

[27] David L. Weaver and Tom Germond. *The SPARC Architecture Manual.* PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[28] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of Supercomputing '96*, November 1996.