# Cache Behavior in the Presence of Speculative Execution - The Benefits of Misprediction

by

**James E. Pierce**

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
1995

Doctoral Committee:

Professor Trevor Mudge, Chair
Professor Edward Davidson
Professor John Hayes
Professor Jeffrey Rauch

# DEDICATION

To my parents, my wife, Pam, and my sons, Jimmy and Austin.

# ACKNOWLEDGEMENTS

I am indebted to many people for making it possible for me to write this dissertation. First and foremost, my wife, Pam, deserves much credit for the completion of my degree. Without her unfailing encouragement, support, and family maintenance, this dissertation would have never been written. I am deeply indebted to her for the time and effort that she has contributed toward making my academic life successful.

I have been blessed with parents who have provided unreserved support and guidance throughout my life. From stressing the importance of education as a child, to providing substantial financial support during college, they have in every way possible assisted me in my academic pursuits.

I want to thank my advisor, Trevor Mudge, for the many hours spent with me during my graduate years. His ideas, insight, and support have contributed greatly to the quality of my work. I especially thank him for his patience during both my periods of low productivity and last-minute bursts of activity. Konrad Lai should receive credit for the initial conception of the some of the ideas presented in this work and I thank him and Intel Corporation for Intel's financial support during much of my course of study. Dave Nagle and Rich Uhlig deserve credit for developing the tools to gather the system traces I used in my prefetching studies and I thank them for sharing the traces with me. I would also like to acknowledge Michael D. Smith's assistance in writing the survey chapter on instrumentation tools. Finally, I want to thank my thesis committee for their time spent reading and listening to my ideas and their helpful comments and suggestions.

During my undergraduate and graduate years, there have been a few people who, while not directly involved in this research, have been both instrumental in my academic

studies and have had a lasting impact on my (and Pam's) life. These people deserve sincere thanks for their guidance and friendship. They are Dennis Peters, Alberto Torchinsky, and Robert Glassey at Indiana University, and Jeffrey Rauch at the University of Michigan.

Finally, I thank God for giving me the ability and the opportunity to complete my graduate work. I realize that it is because of God's grace and unlimited generosity that I am able to accomplish all things in my life. My faith in Him has helped me persevere through the challenges of graduate school and I take great comfort in knowing that faith will continue to help me in all my future endeavors.

I hope that I will someday be able to offer the same kind of support and encouragement to others that I have been fortunate enough to receive from the many people mentioned above.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1
# Introduction

Cache performance analysis is an increasingly important area of computer architecture research. Cache misses cause significant performance degradation in today's microprocessors and, because of increasing instruction issue rates and higher disparities between CPU clock speed and memory access times, misses will have an even greater effect on the performance in future designs. Thus, both the means of simulating cache behavior as well as methods for improving cache performance are imperative in the development of new architectures. This dissertation is a collection of several related studies which examine the cache behavior in next-generation microprocessors. While each study focuses on different aspects of cache simulation and design, the results of one study provided the questions, inspiration, or experimental method for the next study.

The major contribution of the work is the creation of a new instruction cache prefetching algorithm which was found to decrease the cycle time of current-generation processor/memory architectures by as much as 16%. An instruction cache prefetching algorithm fetches memory lines into the cache before they are needed by the CPU's fetch unit. If the algorithm can both correctly predict which lines will be needed, and bring them into the cache before they are needed, many instruction miss cycles can be reduced or avoided entirely. Compared to other prefetching schemes, the proposed algorithm, called wrong-path prefetching, offers better performance in terms of cycle time reduction at a reduced or equivalent hardware cost. The result is surprising because the speedup is achieved by prefetching lines down the not-taken direction of conditional branches. More importantly, my results show that wrong-path prefetching, as well as instruction

prefetching in general, will become even more effective in future architectures. The details of the method and experimental results are described in Chapter 4.

The possible benefits of memory references encountered in the not-taken paths of conditional branches was discovered while studying the effects of speculative execution on cache performance. This study is discussed in Chapter 3. Future superscalar processors will speculatively execute instructions past unresolved conditional branches in order to achieve a high instruction issue rate. Unfortunately, infrequent but unavoidable branch misprediction will cause incorrect instructions to be executed. Even when cache writes are held until branch resolution, incorrect path execution will alter the cache state since memory reads will allocate and possibly displace cache lines. Recovery mechanisms can return the processor to a correct state but the changes in the cache cannot be undone. It was thought that these wrong-path cache changes increase the cache miss rate since the execution of wrong paths generates additional, unnecessary memory references. Consequently, these references would displace needed data, thereby increasing cache pollution, cache misses, and bus traffic. However, the results of the study were not as expected. The dominant effect of wrong path memory references was to prefetch instructions and data for later correct path execution. Additional misses due to cache pollution were minimal even with deep speculation.

Since cache behavior has a large impact on processor performance, it is important that methods are available to study the effects of different cache configurations or prefetch algorithms. This is usually done by simulation using runtime memory reference traces as input. Chapter 2 is a overview of the methods available for reference trace collection. It focuses on the principles of developing instrumentation tools which modify a program in order to obtain a reference trace during program execution. In particular, it describes the operation and design of two instrumentation tools, IDtrace and spex, which were written to obtain memory reference traces on the Intel Architecture platform. These and other

similar tools are utilized to provide input to trace-driven cache simulators and to provide empirical data describing program execution patterns for use in cache model equations.

Chapter 5 discusses efforts made to model the behavior of a prefetched cache. A previously developed cache model called the gap model was chosen to extend to include prefetching effects. The gap model miss prediction is based upon computing the probability that cache lines are displaced between basic block executions. The lengths of the gaps between block executions are determined from the program's execution trace. The longer the gap, the greater the probability that the block's cache lines will have been displaced. The fact that the model is based upon block behavior makes it conducive to extend to prefetched caches.

Derivation of next-line and wrong-path prefetched cache models as well as a non-prefetched cache model are given along with predicted verses measured miss comparison results using SPEC benchmark programs. Unfortunately, the number of misses only partially determines the performance of a prefetched cache since prefetching significantly alters miss latencies. The chapter outlines basic equations which describe the cycle time performance of both non-prefetched and prefetched caches. The equations contain variables which describe the interaction between instruction fetch, cache miss, and cache line refill. When possible formulas were derived for these variables based upon cache configuration parameters. Trends and dependencies were noted when a formula was not evident.

To reduced the chance of confusion with the terms used throughout this paper, several terms will be defined. A program can be partitioned into basic blocks. A basic block is a sequence of instructions which have one execution entry point and one exit point. Thus, if one instruction in the basic block is executed, they all are executed. Basic blocks are delimited by control instructions such as conditional and unconditional branches, subroutine calls, and returns.

Branch prediction is used to guess the control flow direction of a conditional branch before the outcome of the branch condition is known. Branch prediction is implemented in speculative processors to direct instruction fetch beyond conditional branches. The correct path of the conditional branch is the proper execution direction of the branch based upon the outcome of the condition. The wrong path is opposite path of the correct one. The taken path is always the correct path in a non-speculative processor and is the predicted path in a speculative processor. If the branch prediction scheme predicts incorrectly, execution will proceed down the wrong path. A correct-path reference is a memory reference occurring during correct-path execution. Similarly, a wrong-path references occurs during wrong-path execution.

# CHAPTER 2
# Instrumentation Tools

## 2.1 Introduction

This chapter outlines the methods available for reference trace collection. It contains general, user-manual, information concerning the applicability and use of many trace gathering tools implemented on various platforms. In addition, it describes the tools used to gather the memory reference traces upon which the cache performance studies of Chapters 3 and 4 are based. This chapter does not directly pertain to the main emphasis of the thesis and skipping it will not effect the readability of the cache performance analysis chapters.

The instrumentation of applications to generate run-time information and statistics is an important enabling technology for the development of tools that support the fast and accurate simulation of computer architectures. In addition, instrumentation tools play an equally important role in the optimization of applications, in the evaluation of new compilation algorithms, and in the analysis of operating system overhead. An instrumentation tool is capable of modifying a program under study so that essential dynamic information of interest is recorded while the program executes. The instrumentation process does not affect the original logical operation of the test program. In a typical situation, a computer architect uses an instrumentation tool to produce an instruction or data trace of an application. The architect then feeds that trace to a trace-driven simulation program. The usefulness of instrumentation tools is obvious from a quick glance at current research publications in the area, where a significant number of

authors use traces generated by two of the most popular instrumentation tools: pixie [58] and spixtools [13]. These tools are popular because of their applicability to many architectures and programs, their relatively low overhead, and their simplicity of use. If the use of such tools is unfamiliar, several examples are given in Section 2.7.

This chapter's focus is the design of instrumentation tools. Particular emphasis will be given to two tools I wrote to gather memory references on Intel architecture platforms. Both tools, IDtrace and spex, take a binary file as input and create a new, instrumented executable which performs as the original while also outputting a memory reference or full execution trace. In addition, spex incorporates branch prediction, speculative execution, and misprediction recovery in order to produce a memory reference traces approximating a trace obtained when running the application on a speculatively executing microprocessor.

Section 2.2 describes how instrumentation tools fit into the broad range of techniques available for the collection of run-time information. Section 2.3 lists the points in the compilation process at which one can instrument an application. It goes on to discuss the advantages and disadvantages of performing instrumentation at these points, noting that the basic structure of an instrumentation tool and the problems faced are common to all of the approaches. Section 2.4 then discusses the specifics of instrumentation tool design, and Section 2.5 presents the important characteristics of some existing instrumentation tools including IDtrace and spex. Finally, the chapter concludes with an appendix which demonstrates the use of two common instrumentation tools.

## 2.2 Run-Time Information Collection Methods

Before discussing the design of instrumentation tools in detail, other approaches will be described that provide a functionality (i.e., the ability to collect run-time information) similar to that provided by instrumentation tools. In general, a run-time data

collection method can be classified as either a hardware-assisted or a software-only collection scheme. Each type of approach has advantages and disadvantages to consider.

A hardware-assisted collection scheme involves the use of hardware devices that are added to a system solely for the purpose of data collection. These monitoring devices are not necessary for the proper functioning of the computer system under test. Many different hardware methods exist for unobtrusively monitoring system-wide events. They include off-computer logic analyzers such as the University of Michigan's Monster system [46] that monitor the activity of the system bus, specially designed hardware boards such as the BACH system [17] which observe and record bus activity, and special on-chip logic such as the performance monitoring counters on the DEC ALPHA 21064 microprocessor chip which summarize specific run-time events [14].

The two main advantages of a hardware-assisted collection scheme are that one can build hardware to capture almost any type of event and that a hardware monitor can theoretically collect dynamic information without slowing down the application under test. Unfortunately, there are a number of disadvantages to these schemes too. First, since a huge amount of data can be gathered in a short time, the monitoring hardware is built either to summarize events (e.g., a counter that only counts the number of cache misses and not their addresses) or to record disjoint segments of program operation (e.g., a hardware monitor with a large memory that accepts the run-time information at the full execution rate and then later dumps this data to a backing store). In either case, less than the full amount of information is gathered which could lead to distortions in the data. To minimize the amount of unwanted data collected, researchers have combined hardware-assisted approaches with software instrumentation of applications to signal when the hardware should start and stop monitoring [46] — another compelling reason to understand software instrumentation methods. Finally, hardware-assisted collection schemes are costly and highly dependent upon the characteristics of the monitored machine; thus, they are not applicable to the general computer user.

Software-only collection schemes, on the other hand, are relatively inexpensive and more portable than hardware-assisted collection schemes because the software schemes use only the existing hardware to gather the desired run-time information. In general, software-only schemes can be divided into two approaches: 1) those which simulate, emulate, or translate the application code and 2) those which instrument the application code. Briefly, a code emulation tool is a program that simulates the hardware execution of the test program by fetching, decoding, and emulating the operation of each instruction in the test program. SPIM [22] and Shade [12] are examples of tools in this category. One of the major advantages of emulation tools is that they support cross-simulation and the ability to execute code on hardware that may not yet exist. Compared to instrumentation tools though, an emulated binary, even with sophisticated techniques such as dynamic cross-compilation [12], is noticeably slower than an instrumented binary when capturing the same run-time information.

An instrumentation tool works by rewriting the program that is the target of the study so that the desired run-time information is collected during its execution. The logical behavior of the target program is the same as it was without instrumentation, and the native hardware of the original application still executes the program, but data collection routines are invoked at the appropriate points in the target program's execution to record run-time information. It is possible for a instrumentation tool to provide traces for an architectural model different than the machine on which the new binary is run, see Section 2.5.2. Overall, researchers have proposed the following three distinct mechanisms to invoke the run-time data collection routines: microcode instrumentation, operating system (OS) trapping, and code instrumentation.

Agarwal, Sites, and Horowitz [4] describe a microcode-instrumentation technique called ATUM (Address Tracing Using Microcode) that supports the capture of application, operating system, interrupt routine, and multiprogramming address activity. Instead of instrumenting the individual applications, their technique instruments the microcode of

the underlying machine so that the microcode routines record, in a reserved portion of main memory, each memory address touched by the processor. This approach is effective because, typically, only a small number of the microcode routines are responsible for the generation of all memory references. This approach is general because it is independent of the compiler, object code format, and operating system—as Agarwal states, ATUM is "tracing below the operating system [3]." In fact, any information visible to the microcode can be instrumented. Agarwal, Sites, and Horowitz report that the overhead of this approach causes applications to run about ten times slower than normal when used to collect address traces [4]. Of course, microcode instrumentation is only applicable to hardware platforms using microcode and even then, the user must have the ability to modify the code. Furthermore, since most processors today have hardwired control, this approach has limited applicability.

A more widely applicable approach is to collect run-time information using OS traps. For instance, data address traces can be collected by replacing each memory operation in the target program with a breakpoint instruction which traps to a routine that records the effective address. A disadvantage of using OS traps is that, if many events must be recorded, the cumulative OS overhead of handling all the traps is significant. However, there are a number of exception mechanisms in operating systems that can be utilized to improve the efficiency of this method. Tapeworm II [70] is an example of an efficient software-based tool that drives cache and TLB simulations using information from kernel traps. It utilizes low-overhead exceptions and traps of relatively few events. The applicability and efficiency of the OS-trap approach dependents upon the accessibility of certain OS primitives. With proprietary operating systems, this can be a problem.

The most generally applicable approach is the direct modification of the program's code. This approach, called instrumentation, inserts extra instructions into the target program to collect the desired run-time information. Data collection occurs with minimal overhead because the application runs in native mode with, at most, the overhead of a

procedure call to invoke a data collection routine. Most instrumentation tools can create instrumented binaries that run at less than a ten-times slowdown in execution time when collecting an address trace. Some instrumentation tools such as QPT [37] rely on sophisticated analysis routines and post-processing tools to reduce this overhead even more. This approach is generally applicable because it is independent of the operating system and underlying hardware, it has been implemented on systems ranging from Intel architectures [48][49] to the DEC ALPHA architecture [37][62]. Furthermore, most code instrumentation tools require only the executables, not the sources files, so a user can instrument a wide range of programs.

However, there are a number of shortcomings to code instrumentation. It is most suited to the instrumentation of application programs. Furthermore, most code instrumentation tools only instrument single-process programs; kernel code references and multiple process interactions are not typically included. Therefore, address traces generated by these tools are often incomplete and of limited utility for TLB or cache simulations that require the monitoring of system-wide events. Recently however, there have been tools written that do instrument kernel code and multitasking applications [8][15][38].

Overall, software-only collection schemes are less expensive to implement and easier to port from system to system than hardware-assisted schemes. Software-only schemes, however, do impose some overhead on the system under test and often are restricted in the type of run-time information that they can gather. Even so, the robustness and simplicity of code instrumentation tools makes them a popular choice of today's computer architects. The remainder of this chapter focuses on the design of code instrumentation tools.

## 2.3 When to Instrument Code

Code instrumentation can be performed at any one of three points in the compilation process: after the executable is generated, during object linking, or during some stage of the source compilation process. Although different problems arise depending upon when the code is instrumented, the general procedure of instrumentation is the same at all levels. In general, code instrumentation involves four steps:

- preparing the code for instrumentation - code extraction, disassembly, and/or structure analysis,
- adding instrumentation code - selecting instrumentation points and inserting code to perform the run-time data collection,
- updating original code to reflect new code addition - reassembly, relocation information update, or control instruction target translation,
- constructing the new executable.

I will discuss the issues involved in instrumenting code at each of the different stages of application generation.

### 2.3.1 Executable Instrumentation

Instrumenting the executable or late code modification is of the greatest utility to the user. However, it is also the most difficult for the instrumentation tool since a great deal of code structure information is no longer available. The tool is responsible for recognizing and disassembling the code sections, instrumenting the code, and then relocating the code while rebuilding the executable. This missing information affects the tool's ability to perform all three actions. Without the structure information, the tool must invoke compiler knowledge or code structure heuristics to accomplish the tasks which can result in both performance and reliability problems. When the tool cannot accurately predict code behavior statically, runtime overhead is incurred to adapt to the behavior

during execution. In addition, instrumentation can fail or worse, produce incorrect code, due to invalid code structure assumptions. These issues will be discussed more fully in the next section.

Sophisticated tools which can overcome these obstacles present many advantages to the user such as the following:

- Source code independence - This makes to a wide range of programs available for tracing.
- Program generation independence - Most tools can instrument binaries produced by different compilers of various languages.
- Automatic library module instrumentation - Full tracing of user-level execution is easy since the library code is included in the executable (if statically linked.)
- Fast and efficient - No source code recompilation or assembly is required. The user is not required to maintain instrumented library modules.
- Code creation details hidden - The user need not be familiar with the compile-assembly-link process needed to create the application. In particular, details such as the necessary library modules or flags, non-standard linking directives, or intermediate assembly code generation are of no concern.

Late code modification tools have various requirements for the information necessary in the binary file. The most general tools can instrument a stripped binary, a binary without a symbol table. At the other extreme are tools which require the compiler to include additional symbol table information. These tools usually require the source to have been compiled with the `-g` option which includes profile and debugging information in the symbol table. Late code modification tools exist for most microprocessors and many of them are discussed in Section 2.5.

### 2.3.2 Link-Time Instrumentation

If one is willing to give up source code independence, a convenient time to instrument a program is after the objects have been compiled but before the single executable has been created and the relocation and module information has been removed. Instrumentation can be done by a sophisticated linker which includes an object rewriter. During the linking process each object is passed to the rewriter which performs the necessary code modifications. It handles code and data relocation by just noting location changes in the object's relocation dictionary and symbol table. The modified objects are then passed back to the linker proper and are combined into one executable in the normal manner. Recompilation of the source code is unnecessary. The presence of the relocation information and symbol table make relocation straightforward. Postponing modification until the executable stage when this information is missing makes relocation much more difficult and sometimes impossible.

There are several tools which perform link-time modification. Mahler is a back-end code generator and linker for Titan, a DECWRL experimental workstation [72]. The module rewrite linker can perform intermodule register allocation, basic block counting and address trace generation, and instruction pipeline scheduling. Code and data relocation is done as described above. Another tool, epoxie, relies on incremental linking which produces an executable containing a combined relocation dictionary and symbol table [73]. Its advantages over Mahler are that the standard linker can be used and data sections remain fixed so data relocation is not necessary. Epoxie produces address traces and block statistics. An extension of epoxie has been created by Chen which can instrument kernel-level code [8]. It is described in Section 2.5.8.

Link-time instrumentation is not automatic like late code modification and requires input from the user. The user must have the application object files and know the application's linking requirements. In addition, the source files are probably necessary to

generate the object files. While it is not actually necessary to have the source files, it would be unusual to have access to the object but not the source files.

### 2.3.3 Source Code Modification

The earliest time to instrument the code is while it is being compiled. This is also perhaps the most straightforward time since the tool has maximal knowledge about the code. Unfortunately, it has several drawbacks from the user perspective:

- Source files are required.
- Compiler limited - Most tools are either incorporated into one compiler or based upon a particular language or intermediate level generated by one compiler. This further restricts the traceable applications.
- Instrumentation speed - Each time the application is instrumented the source must be recompiled. This also implies that the user must be familiar with the application's compilation procedure.
- Limited code instrumentation - Library modules are not instrumented automatically because they are not included in the source files. It is possible to create separate instrumented copies of all library modules and link them to the instrumented source objects but this requires obtaining the module source code and maintaining multiple versions of modules. Kernel code is difficult if not impossible to instrument with this method.

A major advantage of source-level instrumentation is that the binary creation phase of the instrumentation is greatly simplified. Often the unmodified system assembler and linker can be used to create the binary. Furthermore, the large amount of information available at this stage permits types of instrumentation to be done which are not feasible at later times. For instance, most source-level tools take advantage of compiler control-flow knowledge to reduce the amount of instrumentation code[1]. This reduces both the execution time and resulting trace size.

AE (Abstract Execution) is a tracing system developed by Larus and Ball which is incorporated as part of the Gnu C compiler [6]. Its goal is to generate very small traces which can be saved and then reused for multiple simulation runs. The modified compiler actually produces two executable programs. The first is the modified application. In addition to normal compilation, the compiler uses the notion of abstract execution to insert tracing code in the application code. Abstract execution is based upon control-flow tracing to reduce the amount of trace code necessary. The resulting trace produced by the modified application is only a tiny part of the full trace. This allows traces representing long execution runs to be saved on disk. The compiler also produces an application specific trace regeneration program. The regeneration program is a post-processing tool which accepts the compacted trace and outputs the full execution trace. The tracing overhead, including the cost of saving the compacted trace to disk, is 1-12 times the unmodified program's execution time [36].

MPtrace is a source-level instrumentation tool developed by Eggers et al. to generate shared-memory multiprocessor traces [15]. Their goals were to develop a tool which was highly portable, caused minimal trace dilation, and generated accurate traces, i.e. complete traces which closely resemble those gathered using non-intrusive techniques. Minimizing program dilation is critical in multiprocessor tracing since a change in execution time effects the coordination of multiple processes and thus the overall execution behavior of the program. Source-level instrumentation allows MPtrace to achieve those goals. MPtrace is more closely tied to a parallel C compiler than to an architecture. Thus, its portability dependents upon the compiler's portability. MPtrace was initially created for Sequent ix86-based, shared-memory systems and only twenty five percent of the tracing system was machine dependent most of that being a description of the instruction set.

---

1. QPT also does this but it is much more difficult at the executable stage.

MPtrace attempts to limit execution time dilation by employing compiler flow analysis techniques to reduce the amount of added instrumentation code. It instruments the code by adding assembly instructions to the assembly-level output of the compiler which will produce a skeletal trace. At the same time, program details are encoded in a roadmap file used for later trace expansion. The modified assembly-level sources are assembled and linked using the respective unmodified system tools. A compacted trace is produced upon the execution of the instrumented application. Using a post-processing program and the roadmap file, the full multiprocessor trace can later be generated. MPtrace can achieve a time dilation of less than a factor of 3 but the usual execution time increase is around a factor of 10 [15]. Library module code is not traced.

In summary, there are three times at which code instrumentation can take place. Late code modification does not require source files, library code is automatically instrumented, and the binary creation details are hidden from the user. However, due to the lack of information available in the binary file, late code modification tools are the most complex and the resulting binaries can suffer performance and reliability problems. Link-time modification takes advantage of remaining code information to simplify binary creation. It retains use of the system linker, can instrument module code, but the application source is likely to be required. Finally, source-level instrumentation utilizes substantial code information to simplify the code instrumentation process and to produce complex traces. It requires application sources and usually more information from the user. Library module code is not easily instrumented. The remainder of this chapter will focus on late code modification tools.

## 2.4 How Late Code Modification Tools are Built

An instrumentation tool must insert tracing instructions into the executable without altering the logical behavior of the program. At no point can the added instructions change the program state. For trace generation, the events which need to be

recorded are the execution of basic blocks and all data memory references. With this information, an execution profile, memory reference, or full execution trace can efficiently be produced. The usual way these events are recorded is by adding code segments prior to each event. The code stores the information in a trace buffer which is periodically checked during program execution and flushed to backing store when full. The four tasks of the instrumentation tool are to:

- Find the section(s) of the executable file which contain code and disassemble them to obtain program structure information,
- Insert instructions to record events thereby expanding the original code section,
- Translate all addresses which were changed because of the code expansion,
- Put parts back together to make new executable.

The next four subsections describe the problems faced and the specific actions required of the tool during each of the above stages. The final subsection discusses some architectural properties which facilitate or frustrate late code instrumentation. To assist in describing problems and the methods used to overcome them, we use several existing instrumentation tools as examples: IDtrace for the Intel architecture, pixie for the MIPS architecture, and QPT for both MIPS and SPARC architectures. These tools will be discussed in detail in Section 2.5. IDtrace is used most often as an example due to the authors' familiarity with the tool. However, it should be noted that all late code instrumentation tools encounter similar instrumentation problems and implement similar solutions.

### 2.4.1 Code Extraction and Disassembly

The first steps of the instrumentation tool are to locate and then disassemble the code sections of the executable. Unix executables come in a variety of flavors: ELF, COFF, ECOFF and the BSD a.out format [19][28], but their structure is basically the same. They all begin with tables containing information such as the number, type and

location of sections in the file, if and where the sections are to be loaded into memory, and where to begin program execution. Most executables contain one text section, one data section, and one BSS section. The text section contains code. The BSS section allocates space for uninitiated data and is actually empty in the file. Once the text section is located, it must be disassembled. During disassembly the code is split into basic blocks and a relocation table is created which stores the locations of basic blocks. This table will be needed later to instrument the code and update the target addresses of control instructions. Since instructions will be inserted into the code, almost all instructions will have their location shifted in memory and the branch and jump instruction targets must be translated to reflect this. For most instructions this is straightforward since the targets are known at instrumentation time. For data objects, however, address translation is difficult, and without the symbol table, impossible. It is important that all data locations remain unchanged during instrumentation. Therefore, data sections are not modified and are loaded into memory in their original positions.

In some cases, data can be found within the code segment and this can present several problems for disassembly. There are two reasons a compiler might put non-instruction bytes in the text section. One is to insure constant data cannot be written and to allow the data to be shared by multiple processes. The other source of non-instruction bytes are in-lined indirect jump tables which are created by the compiler for switch or case statements. The obvious problem associated with data in the text section is that, without additional information, the disassembler treats the data words as instructions and tries to disassemble them. These "non-instructions" could mistakenly define basic blocks, be instrumented, or even be modified. Even if the data were not mistakenly modified by instrumentation, earlier code expansion would cause it to be moved within the section. As stated before, data addresses cannot be relocated so this cannot be allowed to happen. The solution is to create a new text section which contains the instrumented code and to treat the entire original text section as a data section. It might be thought that modifying or

adding erroneous instructions would lead to incorrect execution. This will not happen because these "bogus" instructions will never be executed. Since control was never passed to data in the text section in the original program, control will not pass to the instrumented data in the new program.

Another, more subtle, problem is more serious and affects ISAs with variable-length instruction. It is highly likely that after a disassembler blindly disassembles through non-instruction bytes, it will be out of alignment with the following real instruction bytes. For instance, suppose a disassembler creates meaningless instructions from a block of constant data and it needs one byte past the end of the data block to complete the last instruction. Again, these bogus instructions are of no concern because they will never get executed. However, because of the one byte used earlier, disassembly will be out of alignment with the beginning of the true instruction bytes after the constant data and will continue to generate meaningless instructions. To combat this problem, the disassembler must know where non-instruction bytes are located in the text section and skip over them. Constant data locations can be found in the symbol table but locations and sizes of jump tables can only be deduced by knowing compiler code generation behavior. Thus, instrumentation tools like IDtrace which run on ISAs with variable-length instruction must be compiler dependent and could require the executable to contain the symbol table to assist in disassembly. Fortunately for IDtrace, most compilers for the Intel architecture put constant data in the data section and the symbol table is not necessary. However, IDtrace's disassembler is compiler dependent and will not properly instrument programs with unrecognizable jump table code.

### 2.4.2 Code Insertion

Once the code is disassembled, the instrumentation code is added in binary form since there is no later assembly phase. Actual code insertion is not difficult. The only requirement is that the added code cannot alter the current state of the program. Most

instrumentation tools add short code sequences at the beginning of each basic block. If a memory reference trace is required, instruction sequences are also added prior to each memory referencing instruction.

For instance, during profile instrumentation, IDtrace labels each basic block with a unique number. Instrumentation produces two new files: the new executable and a `.blk` file. The latter holds information about each block such as its size, beginning address, and label number. During runtime, an array exists in memory which holds the execution count of each block. A code sequence is inserted before each basic block which will increment the proper array position for that block. When the program exits, this array is dumped to the `.cnt` file. Figure 2.1 is an example of IDtrace basic block instrumentation code. The block count array variable incremented and the trace buffer is checked and emptied if close to full. Even thought each count array entry is a 32-bit unsigned integer value, it could still overflow if the program were sufficiently long. Using a command line option, IDtrace will add code to check for overflow and do sequential saves to the `.cnt` file. This adds extra instructions to each basic block sequence and will slow execution.

```
      push status_flag_reg          ; save status flag register
      push temp_reg                 ; save temp register
      temp_reg <- block_number      ; put block label in register
      M[ctab+(4*temp_reg)]          ; update basic block execution
        <- M[ctab+(4*temp_reg)] + 1 ;  count table
      temp_reg <- tbuf_ptr
      (temp_reg > tbuf_near_full)   ; check if trace buffer is full
      if not goto END
      call flush_buffer             ; if full, flush trace buffer
END:  pop temp_reg                  ; restore temp register
      pop status_flag_reg           ; restore status flag register
```

**Figure 2.1 Basic Block Instrumentation Code - Code inserted before each basic block by IDtrace in profile mode.**

Memory reference code is similar. It calculates the effective address of the data reference and sends it to a trace buffer. Figure 2.2 shows the code added by IDtrace to record a data reference.

### 2.4.3 Address Translation

As the new code is added to the instrumented text section, the control instruction targets must be translated. This is easy for conditional branches and most jump and call instructions because they contain either the absolute target address or its relative offset. Most tools create a relocation table to perform address translations during instrumentation. The table holds the original and corresponding new addresses of all control instructions and their targets. IDtrace accomplishes address translation using two code passes. During the first pass through the code, the original locations of all control instructions and their targets are added to the table. During the second pass, instrumentation instructions are inserted in the code and the new addresses of the targets are added in the table. When a control instruction is encountered and the new location of target is already in the table (this would occur for a backward branch), the new relative distance can be calculated and entered in the instrumented code immediately. When a forward branch is encountered the new location of the target will not be in the table and the new location of the branch must

be noted in the table. Later, when the target instruction is instrumented and its new location is known, the relative offset in the earlier branch instruction is adjusted.

**Original Instruction**

```
reg1 <- reg1 + M[reg2+100]
```

**Instrumented Instruction**

```
push status_flag_reg            ; save status flag register
push temp_reg1                  ; save temp registers
push temp_reg2
temp_reg1 <- reg2+100           ; compute effective address
temp_reg2 <- trc_buf_ptr        ; load trace buffer pointer
M[temp_reg2] <- load_tag        ; record reference type tag
M[temp_reg2+1] <- temp_reg1     ; record reference address
trc_buf_ptr <- trc_buf_ptr + 5; step trace buffer pointer
pop temp_reg2                   ; restore registers
pop temp_reg1
pop status_flag_reg             ; restore status flag register
reg1 <- reg1 + M[reg2+100]      ; original instruction
```

**Figure 2.2 Data Reference Instrumentation Code - Code inserted before a data reference instruction by IDtrace in memory reference mode.**

Unfortunately, there are some control instructions for which the target cannot be calculated at instrumentation time. The most difficult ones to handle are indirect call instructions where the target address is found in a register or memory location. Since the data values are unknown during instrumentation, the target cannot be calculated. Furthermore, instrumentation does not affect data values so execution of the unaltered instruction will produce the original target address rather than the new address. To maintain correct program behavior the address translation must be performed at runtime. As the code is being instrumented, a translation table is created which is a list of original and new address pairs corresponding to the beginning of each procedure. This table is included in the instrumented file and is loaded into memory at runtime. Each indirect call instruction is replaced by a group of instructions which computes the original target address and then passes this address to a call-handling routine. This routine performs a table lookup using the original target address to find the associated new address. If a target

translation is found, control is passed to the translated address and the indirect call works as intended. If, however, the target is not found, an error message is reported and execution halts. Without the use of the symbol table, some heuristic is necessary to detect procedure beginnings. For example, IDtrace marks all instructions following a `return` or `nop` instruction as potential procedure beginnings. If the code contains procedures with other instructions endings or if the target of an indirect call is the middle of a procedure, the table lookup scheme will fail. Even if execution progresses correctly, this method incurs substantial runtime overhead for each indirect call executed and significant memory space is required to hold the table.

Indirect jump instructions also pose a translation problem but can be handled in a similar manner to indirect calls. The jump instruction is replaced by code which computes the original target address and passes the address to the runtime lookup routine. This scheme has two drawbacks however. One disadvantage is the increase in overhead due to more runtime translations. The other is that the translation table requires more entries. Not only procedure beginning addresses but all basic block beginning addresses must be included in the table. This increased table size requires more space and increases address lookup time.

If instrumentation can be based upon compiler code generation knowledge, indirect jumps can be handled in a more efficient manner. In compiled code, indirect jumps are used in two situations. One is in conjunction with a jump table produced for switch or case statements. A jump table is a list of absolute addresses and the target of the indirect jump is found by using a register value as an index into the table. If the jump table can be identified, the absolute addresses can be translated at instrumentation time and the unaltered indirect jump instruction will work correctly at runtime. IDtrace translates the jump table addresses during instrumentation since it has already found the location and size of the jump tables during disassembly. The other use of indirect jumps is for procedure returns in many RISC processors, such as the MIPS and ALPHA architectures.

These too can be translated during instrumentation if assumptions about the compiler are utilized. The discussion of nixie in Section 2.5.3 describes the method.

QPT cleverly stores the translation table in the location of the original text section [36]. Instead of being an opcode, the word at the original instruction address is the translated address. This allows QPT to load a complete translation table, one which holds the translation for every original instruction address, without using any additional memory or file space. This succeeds only because 1) there is not constant data in the text section, and 2) instructions are a fixed 4-byte length.

A final issue in branch translation is branch target distances. Some ISAs such as the Intel architecture, include both short and long target length branch instructions. Usually, code expansion moves the targets out of range of the original short branch instructions. The simplest solution is to convert all short branches to long branches. In MIPS code, all branches targets are 24 bit long but it is still possible for code expansion to push target distances beyond this distance. Pixie can adjust for this if `-branchcounts` is given as a command-line option.

### 2.4.4 Rebuilding the Executable

After the code has been instrumented and target translation is completed, the file sections must be combined to make a new executable. There are now the original text, data, and BSS sections, a new text section, and some tables and buffer space. The original sections must be loaded into memory in their original locations since they contain data. The optimal solution would be to either extend the text section to include the new text and translation table or to create a new text section. Space would be added to the BSS section to include the trace and block count buffers. The executable file format tables would be updated to reflect these changes and to point to the new text section as the location to begin execution. For various reasons, the optimal solution is not possible on many platforms.

The main problem encountered is that many OS loaders do not make full use of the information found in the load format tables. Most formats allow the user to specify of number of text and data sections, the location of where they are to be loaded into memory, and at what address execution is to begin. Unfortunately, to facilitate faster loading, most OS loaders load an application's sections into memory in the same positions in which they reside in the file, ignoring the position information in the format tables. Furthermore, SysV loaders only accept one file structure. It must contain one text section, one data section, and one BSS section in that order. Execution must begin at a fixed address in the text section. The data section must immediately follow the text section. Obviously, special tricks are required to create the new, instrumented binary.

IDtrace does the following as show in Figure 2.3. It combines the original data and the zero-filled BSS sections along with the new text section, trace buffer, and other tables into one big data section. Execution must begin in the original text section so the first few instructions there are modified to transfer control to the beginning of the new code found in the middle of the expanded data section. Another dummy BSS section is added to the end to satisfy the loader's requirement of one BSS section. Note that if the first instructions of the text section were not changed the program would run exactly as before since the original text and data sections are unmodified.

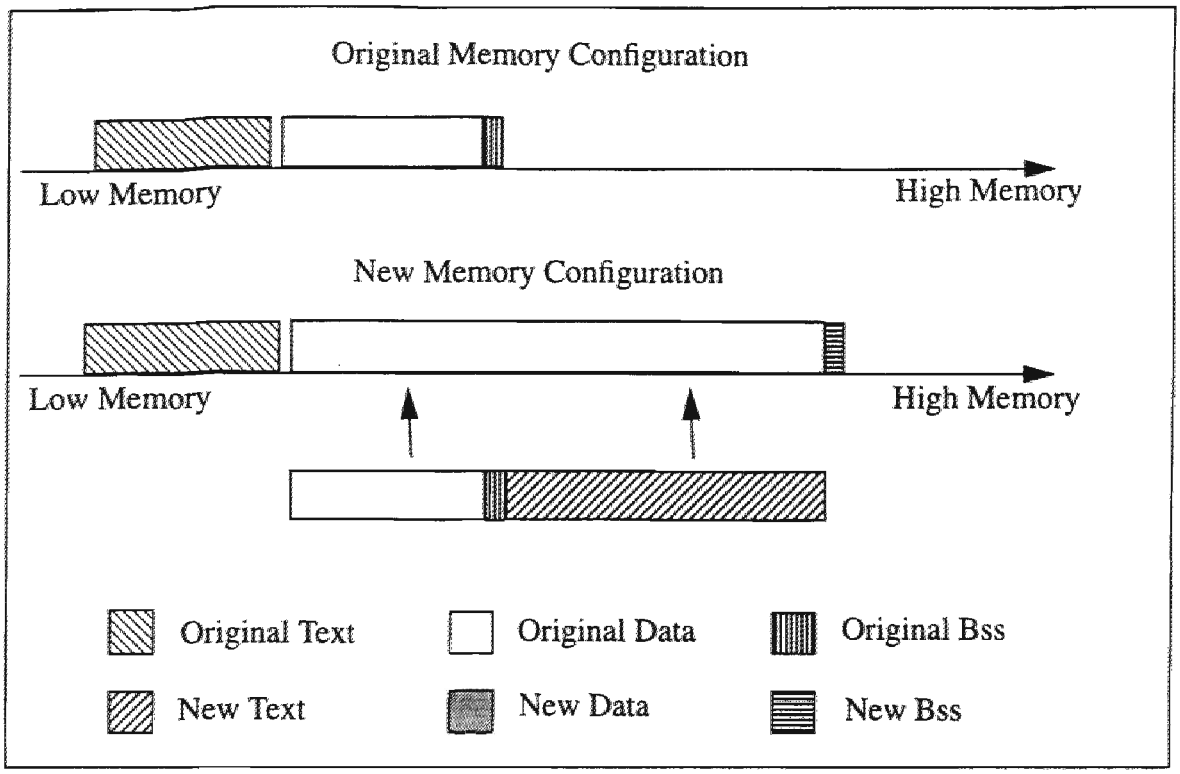


**Figure 2.3 Original and New Binary File Configuration. The new data section contains the original data section, the original BSS section, and the new text section.**

QPT has similar problems on SPARC processors because the text and data sections abut one another leaving no room to expand the text section. In this case, the QPT designers had two choices: add a new text section after the BSS section which would require explicitly represent zero-filled data in the binary file or add a new text section between the data and BSS sections which would create relocation problems with BSS data since the addresses of BSS data would then point to new text code. They compromised. The new text section is added between the data and BSS sections. Then, immediately upon execution, the new text copies itself to a location above the BSS data and zero fills the uninitialized memory space.

Rebuilding methods which expand the data space must allow for correct dynamic memory allocation. For example, on Intel platforms, the last address of the data space is stored in the `_curbrk` variable found in the application program. It is accessed by `sbrk`,

a routine called by C's malloc function to position dynamically allocated memory. The `_curbrk` value must be updated with the last address of the expanded data space so that memory will not be allocated over top of the new code. IDtrace must know the location of `_curbrk` to make this change. Since IDtrace does not depend upon the symbol table, it finds the location of `_curbrk` by pattern matching disassembled instructions with the known sbrk instruction sequence. From those instructions, it extracts the location and updates `_curbrk` to reflect the data section's expanded size. If IDtrace cannot find `_curbrk` a warning message is produced. This is not always an error, however, since `_curbrk` is not included in all programs.

There are several other small issues which must be handled before the new binary will run correctly. First, the exit call must be modified so that the trace and basic block count buffers can be dumped to a file before control is returned to the OS. Most instrumentation tools modify the exit routine to call a new routine which performs these cleanup functions and then exits. The address of the exit procedure can be found in several ways:

- Look up the address in the symbol table. This method, of course, requires the binary to contain the symbol table.
- Pattern matching the disassembled code for the known sequence of exit procedure instructions. This method relies upon code knowledge.
- Knowing the location of a call to the exit procedure in the program and extracting the address from the instruction bytes. This is not too difficult because the initialization library code, crt0.o, contains an exit call and this code is always positioned at the beginning of the text section. This method also relies upon code knowledge.

The start code must also be modified to initialize instrumentation buffers and perhaps open trace files. If the OS loader cannot be told to begin execution at a non-default location, the original start code must also jump to the beginning of the new code section.

As the above sections have described, many problems are encountered when trying to modify an application at the executable stage. Actually inserting the trace code is not nearly as difficult as translating control instruction targets and rebuilding the binary. Some tools rely on compiler-based assumptions to overcome these problems. Others require significant information in the symbol table. Still other tools, such as pixie, sacrifice execution efficiency in order to be almost compiler independent.

### 2.4.5 ISA Properties

Some inherent architectural features simplify instrumentation. Others pose difficulties or add complexity to the resulting code. Some of these properties are discussed below. In general, RISC[1] processor code is more easily instrumented and the resulting code is shorter and faster. However, some instrumentation problems are unique to RISC code.

#### 2.4.5.1 Load-Store vs. Memory-to-Memory Architectures

The major factor in the size and consequently the execution time of a program instrumented to trace memory references is the number of instructions requiring tracing code. Thus a memory-to-memory instruction set such as the Intel architecture which frequently performs operations using memory operands will have many more instructions to instrument than does a load-store architecture which usually retrieves operands from the register file. Memory-to-memory architectures often have a smaller register set which forces local variables to be stored in memory locations. Furthermore, memory operands can often be used as a source and destination in the same instruction thereby generating two trace entries from one instruction. All of these properties of memory-to-memory architectures contribute to the large size and runtime dilation of instrumented code. The

1. By this I mean ISA's with properties such as fixed-length instructions, a large register file, and few memory referencing instructions.

i486 has approximately 180 instructions which can address memory. In addition, many of these instructions can perform both a load and a store and some non-string instructions reference two different addresses [28]. In contrast, the MIPS R3000 has only 14 instructions which can reference memory. Each can only perform a read or a write and no instruction can access more than one memory address [31].

#### 2.4.5.2 Multi-reference Instructions

Some processor instruction sets such as the i486 and the RS/6000 include string operations which can perform an indeterminate number of references per instruction. One example, in the i486 ISA, is the rep instruction prefix which can cause one string instruction to repeatedly access sequential memory addresses until a condition is satisfied. It is impossible to ascertain the number of iterations at instrumentation time. To record an accurate reference trace, the single instruction must be replaced by a sequence of instructions which output the reference, perform the string operation, check the condition, and loop back if the condition is not satisfied. This emulation code adds to the size and execution time of the instrumented binary.

#### 2.4.5.3 Register Allocation

As seen in the sample code in Figure 2.1 and Figure 2.2, registers used in the trace code segments must be first saved and then restored so that the inserted trace code will not alter the current state of the application. If the processor has a large register set, tricks can be performed to eliminate these time consuming operations. For instance, pixie scans the original code prior to instrumentation and utilizes the three least referenced registers as dedicated instrumentation registers. The original code instructions which referenced these registers are replaced with memory referencing instructions. Pixie then uses the registers exclusively as instrumentation registers holding buffer pointers and effective address calculations. They are used in instrumentation segments throughout the program without

having to continually save and restore their values [73]. QPT relies on the caller-save procedure register convention to scavenge instrumentation registers. QPT finds registers which were saved by the calling procedure but unused in the current procedure. This assumes that the program obeys the calling convention, and QPT tries to use symbol table information and optional command-line arguments to verify obeyance. If it cannot be assured, the register values are saved and restored as described earlier. Because their target processors have 32 registers, pixie and QPT are able to contain code expansion.

#### 2.4.5.4 Condition Codes

Condition code values are part of the state of the computer and so cannot be altered by actions in the tracing code. The Intel architecture has special instructions which push and pop the status flag register and these instructions are used by IDtrace hide any affect the tracing code might have on the flags. The SPARC processor has four condition code registers. While the processor does not have user mode instructions which save and restore the registers, two types of arithmetic instructions are implemented: one which affects condition codes and one which does not. QPT's tracing code uses the non-affecting arithmetic instructions in all places except for the trace buffer overflow check. In this case, it either inserts the check instructions where the condition codes are not live or performs the check with a more expensive code sequence which does not affect the codes.

#### 2.4.5.5 Variable Instruction Lengths

Variable length instructions in combination with data located within the text section can wreak havoc with code disassembly. The disassembler must use information in the symbol table to skip constant data and use compiler specific knowledge to recognize and pass over jump tables. This was an unexpected and serious problem with IDtrace. Instruction length also affects the length of the output trace. When instructions are of uniform length, the trace need not contain the address of each instruction in order to

quickly derive an execution trace. It is sufficient only to output each executed basic block beginning and data reference addresses. The position of data references relative to instruction references can be denoted using only a small integer offset. The offset represents the number of instructions executed since the last basic block beginning or data reference.

#### 2.4.5.6 Delayed Branches

Delayed branches in some RISC processors necessitate careful instrumentation. An instruction in a delayed branch slot succeeds a branch instruction in assembly code order but will get executed regardless of the branch direction. It is important that no instrumentation code get inserted between the branch and the delay slot instruction. The easiest way to handle this situation is to move any delay slot instruction which requires instrumentation to a location prior to the branch. It must be verified that this movement does not affect the outcome of the branch.

#### 2.4.5.7 Indirect Addressing

Finally, ISAs with heavy dependance upon indirect addressing will suffer from the overhead caused by the runtime address translation. In the MIPS architecture for instance, procedure returns are done with the jump register instruction (`jr`). The call instruction stores the return address in a general purpose register (usually r31) and `jr` indirectly finds the return address in that register. Thus, every return causes an address table lookup thereby adding to the execution time of the instrumented program. A method to avoid this overhead which is based upon compiler knowledge is described in Section 2.5.3.

## 2.5 Current Instrumentation Tools

Late code instrumentation tools can be found for most of the popular current microprocessors. The following is a description of a selection of tools for use on various platforms.

### 2.5.1 IDtrace

IDtrace is an instrumentation tool for the Intel Architecture Unix platforms [48]. It instruments SysV R4 ELF binaries compiled using the Intel/AT&T C, USL CCS C, and gcc compilers. Currently, it cannot automatically process code compiled by Intel's Proton compiler developed for the Pentium. IDtrace can produce a variety of trace types including profile, memory reference, and full execution traces. Primitive post-processing tools which read output files, view traces, and compute basic profile data are included in the IDtrace package. IDtrace can instrument stripped binaries, i.e., the symbol table is not needed. However, the executable must be statically linked and kernel code references are not included in the trace. Using full execution trace instrumentation, IDtrace will produce a executable which is about 5 times larger and runs 10-12 times slower than the original.

Primarily due to the need to recognize jump table code for disassembly purposes, IDtrace is compiler-dependent. To help alleviate problems due to non-compiler generated code, IDtrace can accept hints from the user on how to instrument a binary. The location or size of a jump table or the location of the beginning of a procedure are examples of such hints. IDtrace reads the hint information from an input file and uses it to assist in disassembling the code and translating addresses. As an example, execution of an instrumented program might abort with a message stating that a particular indirect call target address could not be translated at runtime. This could occur if IDtrace not recognize the address as a procedure beginning and add it to the runtime transition table. The user could add this address to the hint file and reinstrument the program. IDtrace will then

include the address and its translation in the translation table so that runtime lookup can occur during re-execution. While this process is tedious, it does allow the execution of handwritten or other non-compiled assembly code.

### 2.5.2 spex

Spex is a binary instrumentation tool which allows the user to gather speculative traces on a non-speculative processor. Like IDtrace, it was built for ix86-based computers running Unix SysV R4 and currently will instrument only statically-linked code. Regular trace generation instrumentation tools add additional instructions to a program to record application code runtime information and memory references while preserving the original logical operation of the program. Spex not only instruments the benchmark code to output memory references but it also incorporates branch prediction and wrong path recovery code to allow the execution of mispredicted paths. Spex is similar to IDtrace in use, except that spex requires two additional inputs: a wrong path execution depth number and a choice of branch prediction algorithm. A depth argument of zero corresponds to no speculation while a positive argument represents the number of instructions to be executed down the mispredicted path before recovery. During execution of the instrumented binary, a conditional branch which is mispredicted by the prediction algorithm will cause execution to proceed down the wrong path to the specified depth. When that depth is reached, a recovery procedure restores the state to that prior to the conditional branch and execution resumes in the proper direction. Conditional branches encountered during wrong path execution are predicted using the same algorithm. Since all mispredictions take the same number of instructions to resolve, the case of a later conditional branch being resolved before an earlier one cannot occur.

Currently six prediction algorithms have been implemented. Three are static algorithms: always taken, prediction based on branch opcode, and prediction based on opcode and direction. Two are dynamic predictors using a variable sized history table. The

table is indexed by the address of the conditional branch. One algorithm keeps a bit in each table entry corresponding to the previous direction of the branch. The other has two or three bit saturating counters as table entries which maintain a weighted history for each branch [56]. The last algorithm uses a profile file. The predicted direction of the branch is based upon the entry for that branch in the file. If no entry exists, one of the above algorithms is used to predict the branch.

As an example of use, spex can instrument a program called `bench` by typing:

```
spex -d 10 -b 5 bench
```

where bench is a statically linked executable. The new executable, `bench.spex`, will execute 10 instructions down a wrong path and use branch prediction algorithm number 5 (dynamic counter with a default counter size of 2 bits). Figure 2.4 shows that when the new executable is run two more files are created, a `.spst` statistics file containing runtime information and a `.sptr` trace file. The trace file can then be piped into a cache or memory simulator. The trace format is a list of 5-byte entries (one byte tag, four byte

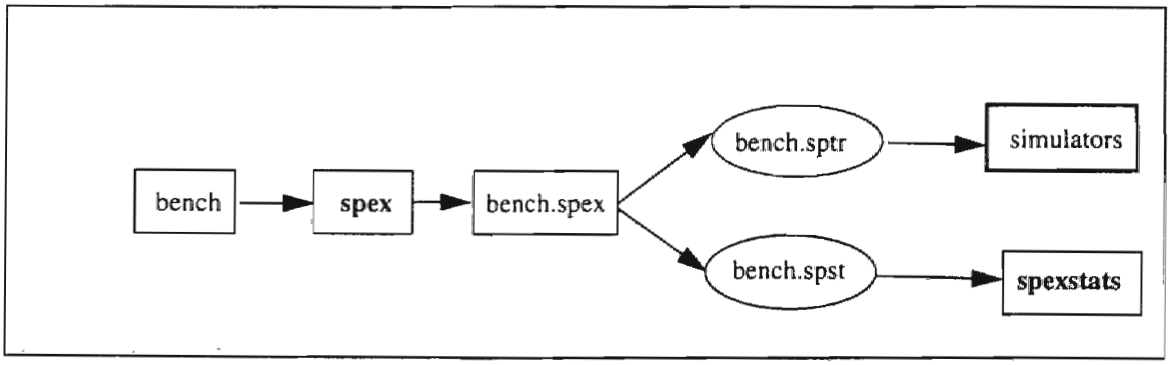


**Figure 2.4 `spex` Programs and Files - Rectangles are executables, ovals are data files produced by `spex`, boldface names are spex tools.**

address); one for every instruction and memory reference. Different tags are used to differentiate between correct path and wrong path references.

Spex adds code prior to each conditional branch to call the prediction algorithm and possibly begin wrong path execution. Code must also be added before each instruction

to possibly exit from wrong path execution. Finally, guard instructions must be added prior to any instruction that can cause an exception. This prevents an exception from occurring during the execution of a mispredicted path. For instance, suppose during execution of an instrumented program a pointer has yet to be initialized and it has the value zero. The original code might be of the form

if (*pointer initialized*)
then *reference pointer*
else *initialize pointer*

and the conditional branch corresponding to the *if* statement could be mispredicted. Then an attempt would be made to reference memory location zero and a segmentation fault would halt execution. Boundary checking code is added before all reads and writes to prevent this error. Unfortunately, there are many more exceptions such as divide by zero and floating point errors which could be triggered during wrong path execution due to incorrect data. Adding code to check for all possible exceptions would greatly reduce execution speed and since segmentation faults were the only errors occurring in any of the tested code, they were the only ones handled.[1]

The instrumented code action is straightforward. Assume a correct path is being executed. At each branch, the prediction algorithm is called and the result is compared with the known actual direction of the branch. If they are the same execution continues down the correct path. If the branch is mispredicted, the register state and correct next instruction location are saved, a counter is set to the wrong path depth argument, and execution begins down the wrong path. The following actions are taken for each wrong path instruction:

- The counter is decremented and when it is zero wrong path execution is terminated.

1. The C signal library functions could have been used to handle all exceptions efficiently but its use would have interfered with breakpoint exceptions and made debugging the binary code extremely difficult.

- If the instruction performs a read or write the address is first checked against segment boundaries to prevent segment violations caused by incorrect data.
- Memory reference entries are output with special wrong path tags.
- If the instruction performs a write the address and original value are stored in a write restore buffer.

When wrong path execution terminates the writes are undone using the data stored in the write restore buffer, the register state is restored, the prediction algorithm is updated, and execution resumes down the saved correct branch path. Sometimes wrong path execution can terminate without going down the full wrong path depth. This can happen in the following cases:

- Exit call - Cannot exit during wrong path execution.
- System call - This is a call to the OS and cannot be traced.
- Data segment fault - An invalid read or write outside the data segment would cause a fault if allowed to proceed.
- Indirect jump - An attempt to index a jump table with an invalid index will usually cause execution to continue outside the text section so wrong path execution is always halted.
- Indirect call - Like in IDtrace, indirect calls in instrumented code are handled by a runtime lookup matching the original target address with the new target address in the instrumented code. If the original address is computed from incorrect data the lookup will fail and wrong path execution halted.
- Execution fault - This could be caused by a floating point or divide by zero exception. As stated earlier, code cannot be added before every instruction to test for all possible exceptions so in this case execution terminates.

While this appears to only roughly approximate actual execution behavior, only the system call termination differs from the actual behavior of a speculative processor. Moreover, the above termination conditions occur very infrequently. In the test running

several benchmarks for a combined total of about 175 million mispredicted paths, 90% of the mispredicted paths executed to a depth of at least 25 instructions, see Table 2.1.

| Termination Type | Occurrences | Percentage |
|---|---|---|
| System Call | 347K | < 1% |
| Data Segment Fault | 14.1M | 8% |
| Indirect Jump | 947 | 0% |
| Indirect Call | 13K | 0% |

**Table 2.1 Termination Statistics - This shows the number and percentage of mispredicted paths which were terminated before executing down 25 instructions. The total number of mispredicted paths was 174.8 million.**

The speculative instrumentation adds significantly to the size and runtime of the new binary. Code expansion is roughly 25 times. Runtime is increased by factors of 25-45 for zero depth (no speculation) and 35-65 for wrong path depth of 10.

### 2.5.3 pixie and nixie

Pixie was the first binary instrumentation tool which received widespread use. Pixie is a full execution trace generation tool which runs on MIPS R2000, R3000 and R40x0 based systems [58]. The tool is included in the performance/debugging software package of most systems based upon the MIPS architecture. Versions are available which instrument ECOFF and ELF file formats. With newer versions of pixie, if pixified dynamic libraries exist, they can be linked into the instrumented application to generate traces of dynamically-linked as well as statically linked code. Pixie does not, however, record kernel activity.

The default instrumentation option is to record only basic block execution counts. An informative post-processing tool, pixstats, can interpret the output to present a wide-array of runtime statistics. Using command line arguments, pixie will also instrument the application to produce an instruction and/or data trace. The reference trace output is written to a file descriptor. Using another tool called makepipe, the trace can be piped

directory to a trace consumer program such as a memory simulator. Program expansion and time dilation depend upon the type of instrumentation used. When tracing both instruction and data references, the new executable is roughly 3 times larger and 4 to 5 times slower. The time dilation does not count the time required to save or pipe the trace.

Pixie is virtually compiler-independent. Constant data in the text section does not cause disassembly problems because the MIPS architecture has fixed-length instructions. It avoids having to recognize and decipher jump tables by performing all indirect jump address translations at runtime. Thus, switch generated indirect jumps, procedure returns effected by jump-to-register-value instructions, and indirect calls, all incur the overhead of a runtime table lookup to perform the target address translation. While pixie is not as restrictive as IDtrace, it does have some limitations. Like, IDtrace, it must use some heuristic to decide upon basic block separation. These heuristics are based upon MIPS compiler generated code. Hand assembled code could cause errors in separation and lead to inaccurate results. In addition, pixie cannot trace past fork calls and will fail on some special library routines.

In an attempt to lower the runtime overhead of pixie, another tool called nixie was created [73]. At the cost of becoming compiler-dependent and operating on a smaller set of application binaries, it makes assumptions about the binary code structure in order reduce runtime address translations. One of the main sources of these translations is the use of indirect jump instruction, `jr`, to perform procedure returns in MIPS code. The compiler convention for a procedure call is to use `jal` or `jalr` and put the return address in `r31`. The return code convention is to use `jr r31`. Nixie avoids the runtime translation for the return by translating during instrumentation the return address found in the `jal` instruction. Then, nixie assumes that `jr` via `r31` is a return and the value in `r31` has already been translated. `jalr` instructions are treated as indirect calls and are translated using the runtime lookup table as before. When the new address is found, the new return address is put in `r31`. The remaining `jr` instructions (the ones not using `r31`)

are assumed to be indirect jumps produced by case or switch statements. Nixie recognizes the code patterns the compiler uses to begin a jump table and deciphers the size and memory location of the jump table. The entries in the table are translated at instrumentation time so they do not require runtime translation. The developers found about two dozen places in standard library code where the above assumptions were incorrect. Fixes for these exceptions were built into nixie so that most code can be instrumented without error.

Because nixie makes compiler-based assumptions about code structure, it can only instrument a subset of the pixie-instrumentable applications. However, results from benchmark tests showed that the runtime of nixie instrumented binaries were up to 30% faster than pixie-instrumented ones [73].

### 2.5.4 Goblin

Goblin is a trace generation tool which instruments IBM RS/6000 applications [64]. It annotates code on the basic block level, i.e., code is added prior to each basic block to report block execution. Goblin has characteristics of both a late code and link-time modification tool. It accepts as input an executable with a detailed symbol table yet performs instrumentation separately on each object. The instrumented objects are reassembled and linked into a new executable by the system's assembler and linker programs. Goblin's first step is to use the descriptive symbol table to separate and disassemble the executable into assembly code objects. It then annotates the assembly code, records static data about the blocks in the objects, and updates the symbol table to reflect the instrumentation changes in each object. The regular system assembler and linker are then used to create an instrumented executable from the instrumented objects. The profile routines are introduced at the link stage as a profile library to be included in the image. The user can select different kinds output traces by linking in different trace libraries. Several libraries exist. One generates a complete basic block trace. Another

allows the generation of a full memory reference trace. Finally, since storage of large traces is difficult, there is library which performs on-the-fly basic block statistic calculations so that the whole trace need not be saved.

### 2.5.5 SpixTools

SpixTools comprises several programs that implement late-code modification of SPARC application binaries to produce instruction-level statistics [13]. The two main tools in the SpixTools distribution are spix and spixstats. Spix accepts an executable program and generates an instrumented executable. When run, this instrumented executable produces, in addition to its normal output, information indicating the number of times that each basic block in the original program was executed. By default, this information is directed to file descriptor 3, but the user can change this default through the use of the `-fd` option in spix. Unlike pixie, spix does not generate instruction or data traces; it only generates basic block counts.[1]

Spixstats uses the basic block counts to summarize the behavior of the instrumented program. This tool creates tables of (static and dynamic) opcode usage, branch and delay slot statistics, register and addressing mode usage, distribution of constants in immediate and displacement fields, and function execution information. The ranking of functions is based on the total number of instructions executed in that function and not on the total number of cycles spent in that function. Exact cycle counts would require specific pipeline and memory system information which not available to spixstats.

Spix handles the problems with executable instrumentation in similar fashion to the tools already discussed. For instance, when spix cannot correctly identify the targets of a register-indirect jump instruction, it simply has the instrumented executable print a

1. Older versions of spix were capable of generating instruction and data traces. These capabilities were removed since other SPARC tools such as shade have made these capabilities unnecessary.

diagnostic message indicating the address of the undiscovered target instruction and then terminate abnormally. Through the use of the `-jaddr` option in spix, the user then re-instruments the executable with this extra piece of information. This method is not unlike the hint information in the IDtrace approach. Furthermore, like the previous tools, spix works only with static code (no support for self-modifying code or dynamic libraries), and it is not capable of instrumenting the kernel.

For the SPEC89 benchmarks, spix roughly quadruples the size of the executables. For the integer benchmarks where the average basic block size is small, the spix-instrumented executables run approximately 2.5-times slower. On the floating-point intensive benchmarks where instrumentation code execution can be overlapped with long latency floating-point operations and the basic block size is larger, the spix-instrumented executables run anywhere from 5% to 50% slower [13].

### 2.5.6 QPT

Like its predecessor AE, the design goal of QPT is to produce compact traces which can be stored for later simulations [37]. The difference between the two tools is that QPT instruments the executable while AE is part of a C compiler. This allows QPT to be applicable to many applications created by various compilers. As noted in the last section, QPT must overcome the disassembly and relocation obstacles common to all late code modification tools. In addition, QPT performs control flow analysis to reduce the amount of inserted tracing code. Therefore, it must rely heavily on symbol table information and code structure knowledge in order to reconstruct the exact code structure. QPT processes the code on a procedure basis. The address of each procedure is found in the symbol table and a control flow graph (CFG) is constructed with a basic block at each node. Using heuristics to decide the likeliest execution path, optimal code insertion points are located on CFG edges rather than nodes (blocks) and trace instructions are added to the original code.

The trace regeneration process is another unique feature of QPT. The trace output by the instrumented program is a compact trace which needs expansion before it can be used by a trace consumer program. Most tools supply statically created information files which can be read by a post-processor program to expand the trace. The AE system creates an application-dependent trace regeneration tool for each instrumented application. In both these cases the expanded trace would then be piped to the consumer program. QPT instead creates a regeneration program object file which can be linked into the compiled consumer program. Thus, the consumer program can read the compacted trace directly from disk [36].

The performance of the abstract execution instrumentation depends upon the regularity of the program's control flow and memory reference patterns. Numeric programs with sequential access patterns and few conditional branches require less instrumentation and therefore produce a more compact trace than do non-numeric programs with more irregular behavior. Statistics reported by Larus in [36] show that the runtime of traced programs ranges from 1.4 to 12.3 times that of the non-traced program. These numbers include the time to store the trace to disk. The compact traces are between 13 and 250 times smaller than the expanded full execution trace. Larus states that regeneration costs are insignificant since the regeneration routine can produce the full trace at a rate of 200,000 to 500,000 addresses per second while most memory simulators consume addresses at the rate of tens of thousands per second. QPT does not currently instrument dynamically-linked shared libraries but could be modified to do so.

### 2.5.7 ATOM

Unlike the other instrumentation tools discussed previously, ATOM [62] is a tool that allows the user to build his/her own customized instrumentation and analysis tools. For example, using ATOM, a few small C routines can be written to emulate the functionality of pixie and pixstats on a DEC ALPHA machine. On the other hand, if the

trace information generated by pixie is not adequate, ATOM can be directed to gather and analyze a customized set of trace information.

Within ATOM, the authors have defined a set of instrumentation primitives common to all instrumentation programs. These primitives separate the tool-specific part of an instrumentation program from the common infrastructure required by all instrumentation tools. As a user, you write C routines using ATOM's instrumentation library that indicate the parts of the application program that interest you. For instance, ATOM provides library routines that allow you to have access to each procedure in an application, each basic block in that procedure, and each instruction in that basic block. By appropriately indicating where instrumentation code should go (e.g., before or after a particular set of program structures) and by indicating the particular information to be gathered at this instrumentation point, you can use ATOM to access to all of the dynamic information in an application.

In addition to instrumentation routines, an ATOM user can also write analysis routines (e.g., cache simulation routines that use the instrumentation data) that become part of instrumented program. In this way, both the instrumented code and the analysis code run in the same address space, and thus experience the lower communication overhead of a simple procedure call rather than that of context switching, file piping, or inter-process communication. The ATOM system guarantees correct operation by ensuring that the instrumented routines and the analysis routines do not share library procedures or data. Still, the incorporation of the analysis routines into a single executable with the instrumented application program can cause perturb the output trace. For instance, if an analysis routine dynamically allocates memory, the trace of the heap addresses in an instrumented application will be different from the addresses used in the uninstrumented version of that application. ATOM employs several techniques and urges the user to avoid certain programming constructs to make certain that the behavior of the application is unchanged by the instrumentation and analysis routines.

ATOM is implemented on top of a link-time modification system called OM [63]. ATOM works by translating an ALPHA executable[1] into OM's RISC-like symbolic intermediate representation. Through some extensions to OM, ATOM inserts instrumentation procedure calls at the appropriate points in the application code, optimizes the instrumentation interface, and translates the symbolic intermediate representation back into an ALPHA executable.

Since ATOM starts with an executable file, it can be considered to be a late-code modification tool. It, however, is not as robust an approach as a system like pixie since ATOM requires relocation information in the executable image in order to work. This relocation information does simplify the work required to adjust branch targets due to the insertion of instrumentation code.

Another advantage of the ATOM approach is that the underlying OM system can efficiently support an approach that does not steal registers from the application program. ATOM, like QPT and unlike pixie, uses the typical register save and restore mechanisms of a procedure call at each instrumentation site. This approach is desirable because it means that ATOM works on programs that use signals and setjmp - program features which are difficult to correctly handle under an approach that steals registers. The downside of a procedure call approach is that it incurs a greater overhead for each instrumentation action, especially if one does not have exact information on the register requirements of the instrumentation routines. Since the instrumentation routines can be quite complex in the ATOM system (remember that ATOM allows the user to use the instrumentation information immediately in an analysis routine), ATOM relies on sophisticated heuristics and techniques to reduce the procedure call overhead.

The performance of ATOM is related to the granularity of instrumentation and the complexity of the analysis routines. Srivastava and Eustace [62] report performance

1. ATOM is currently available for the DEC ALPHA architecture though it is designed to be easily ported to other machine architectures.

numbers for several different analysis tools built with ATOM. To summarize, for an analysis tool that instruments each memory reference and simulates a direct-mapped 8 kilobyte cache, Srivastava and Eustace found that it took an average of approximately 120 seconds[1] to instrument each program in the SPEC92 benchmark suite and that each instrumented program ran an average of nearly 12-times slower than the uninstrumented version. On the other hand, for an analysis tool that simply instrumented each system call site and summarized this information, they found that it still took only 120 seconds on average to instrument the SPEC92 suite but each instrumented program now ran only 1.01-times slower. Overall, ATOM is a powerful tool for building customized analysis programs.

### 2.5.8 Multitasking and Kernel Tracing Tools

Most code instrumentation tools simply record user-level events within a single thread of control. Recently though, researchers have implemented tracing systems that extend existing code instrumentation tools so that they are able to capture multitasking traces and kernel actions. We briefly describe two such systems that illustrate the key issues related to the gathering of an accurate interleaving of application and operating system reference traces within a multitasking environment. Obviously, one could further extend these tools so that they could record other types of dynamic information.

The basic action of any multitasking tool is the sequenced collection of trace data from each instrumented application into a single global trace buffer. Recall that the act of instrumenting an individual application involves the placement of instrumentation code around the points of interest in the program and the inclusion of extra support routines which provide initialization, trace buffer management, and other support functions. In general, the instrumentation of each program in a multitasking workload is identical to the

1. Srivastava and Eustace [62] report that all measurements were performed on a DEC ALPHA AXP 300 Model 400 with 128 megabytes of main memory.

instrumentation of a single program except that the support routines change to reflect the management of the shared trace buffer. On the other hand, the trace of a multitasking workload is slightly different than the trace produced by a single application because the multitasking trace must include extra process information to disambiguate the trace items of one process from the trace items of another process. For efficiency and practicality reasons, the existing multitasking tracing tools add extra support code into the operating system kernel to help gather this process information and ensure the consistent writing of the global trace buffer.

For the most part, the operating system is just another instrumented application. However, the portions of the operating system that are required to support the tracing system must be runnable with tracing turned off. The dumping of the global trace buffer to disk, for instance, is not part of the normal operation of the system and thus should not be traced. Furthermore, several portions of the operating system are too delicate to instrument automatically. For example, standard basic block instrumentation techniques will fail to instrument properly an operating system routine which flushes the CPU write buffer.

Chen [8] describes one such multitasking tracing tool based on the epoxie instrumentation tool [73] that modifies executables prior to linking. Chen's modified epoxie tool instruments code written for the MIPS instruction set architecture and thus, like pixie [58], uses register scavenging to select registers for use by the instrumentation code. Ideally, one would like to share the pointer into the global trace buffer indicating where the last trace item was written among all of the instrumented applications. Unfortunately, register scavenging precludes the direct mapping of a single global buffer into each application since we cannot guarantee that one single register is available in all instrumented applications at all times. As a result, Chen's system maintains a trace buffer for each traced process, and at every entry into the kernel, the kernel copies the contents of the current process's trace buffer into the global trace buffer.

The tracing of system activity is more sensitive to software trace distortion than the user-level tracing of a single application. Chen's tool illustrates how one can minimize the problems of memory and time dilation. Even though epoxie creates instrumented executables with very little code expansion due to its link-time optimizations, these instrumented executables are approximately 2-times larger and run approximately 15-times slower than the uninstrumented versions of the executables [8]. Chen compensates for the memory dilation in two ways. First, the traces are gathered on a system with a large amount of physical memory so that page misses due to limited memory capacity do not occur, and second, he uses the trace to simulate the TLB behavior of an uninstrumented system. Chen only partially compensates for the time dilation since the focus of his experiments is not influenced by some of the effects of time dilation. In particular, he reduces the rate of the system clock interrupt by 1/15, and he scales the idle activity - the time spent in the operating system idle loop - by a factor of 15. These rough compensations are adequate since his research focuses on memory system behavior, and Chen claims that memory system behavior is largely unaffected by errors in these areas. The other operating system entity affected by time dilation is the process scheduler. Finally, Chen ignores the effects of time dilation on scheduler policy by focusing on single-process and client-server workloads where context switches are driven by the applications and not by the scheduler policy.

Mazieres and Smith [38] describe another multitasking tracing tool based on the QPT instrumentation tool [37] that performs late code modification. Unlike Chen [8], their research is interested in the analysis and evaluation of I/O-bound applications such as network applications. Therefore, they organized their multitasking tool to reduce the effects of time dilation. Essentially, Mazieres and Smith attack the problem of time dilation in two ways. First, they chose QPT as their base instrumentation tools since it uses abstract execution [6] to minimize the amount of instrumentation overhead that occurs the execution of an instrumented application. Secondly, they implemented their tool on a

SPARC architecture where they could take advantage of several unused registers that are reserved by the SPARC ABI [67]. They use one of these reserved registers as the single, global, register-based, trace-buffer pointer that is shared by all instrumented executables. This decision removes the need for the copying of the per-process trace buffers into the global trace buffer as seen in Chen's system. They also describe a few other optimizations that have the potential to further reduce instrumentation overhead. Overall, the systems by Chen and by Mazieres and Smith prove that it is possible to gather useful multitasking traces using code instrumentation techniques. However, there are several problems that make the gathering of accurate multitasking traces significantly more difficult that the gathering of a single application trace.

## 2.6 Summary

Instrumentation programs are powerful tools used by architects and system designers to obtain runtime information for use in performance simulation. While there are many ways that this information can be collected, instrumentation tools allow the general user to obtain accurate traces in a quick and efficient manner. IDtrace was the first such tool written for the commonplace ix86 architecture. Both it and its successor, spex, were used to gather memory reference traces for performance studies detailed in the following chapters.

## 2.7 Appendix

This appendix gives two examples of how late code modification tools can be used to gather dynamic information. We assume that the user is familiar with Unix and can create a statically-linked executable on a Unix system.

### 2.7.1 Runtime Statistics

Suppose one wanted to compare to frequency of usage of certain instructions between several architectures. In particular, suppose one wanted to compare the most frequently used instructions in a typical RISC processor (R3000) with that of a CISC-like processor (i486). This could easily be done using two instrumentation tools: pixie on a MIPS R3000-based DECstation running Ultrix and IDtrace on a i486-based SysV Unix system. Suppose `cc1`, the major part of the C compiler `gcc`, is used as a benchmark program. The program `cc1` must be statically linked but neither the symbol table in the binary nor the sources are necessary. The steps required to use IDtrace are show in Figure 2.5. The use of pixie is similar. First, we instrument the i486 version of `cc1` by typing

```
idt cc1
```

The will produce the instrumented binary `cc1.idt` and the basic block information file `cc1.blk`. Then typing

```
cc1.idt stmt.i
```

will execute the instrumented version of `cc1` and also produce the basic block execution count file `cc1.cnt`. The post-processing tool vcount can then be run,

```
vcount cc1
```

to produce some basic runtime statistics. Part of the statistics is shown in Figure 2.6. Pixie

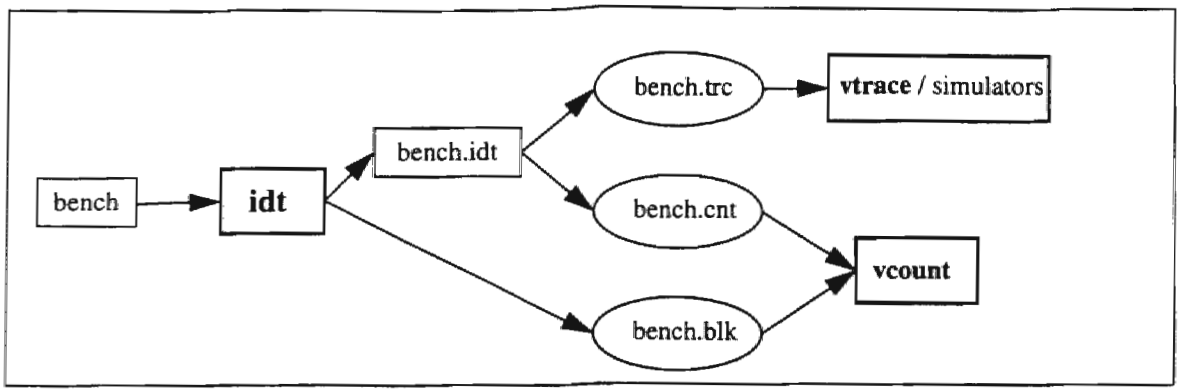


**Figure 2.5 IDtrace Programs and Files - Rectangles are executables, ovals are data files produced by IDtrace, boldface names are IDtrace tools.**

| Instruction Usage | | Percentage |
|---|---|---|
| mov | 19306218 | 29.7% |
| cmp | 9642978 | 14.8% |
| push | 4211418 | 6.5% |
| je | 4166772 | 6.4% |
| jne | 3309404 | 5.1% |

| Other Information | |
|---|---|
| Dynamic instruction count: | 65081680 |
| Dynamic block count: | 17257218 |
| Average instructions per block: | 3.8 |
| Static block count: | 41807 |
| Largest block (# of instructions): | 95 |

**Figure 2.6 i486 Profile Information - Obtained using IDtrace and vcount.**

works in a similar manner. First the executable is instrumented by typing

```
pixie cc1
```

which creates the files `cc1.pixie` and `cc1.Addrs`. Then the new program is run,

```
cc1.pixie stmt.i
```

to produce the `cc1.Counts` file. Finally, `pixstats` reads the output files to calculate an extensive list of runtime information part of which is shown in Figure 2.7.

| Instruction Usage | | Percentage | Other Information |
|---|---|---|---|
| spec | 27615307 | 33.19% | 84450624 (1.015) cycles (3.38s @ 25.0MHz) |
| lw | 13027613 | 15.66% | 83199619 (1.000) instructions |
| addu | 7676940 | 9.23% | 17272839 (0.208) basic blocks |
| addiu | 7363426 | 8.85% | 13217812 (0.159) branches |
| sw | 7357767 | 8.84% | 4.8 instructions per basic block |
| | | | 6.3 instructions per branch |

**Figure 2.7 MIPS R3000 Profile Information - Obtained using pixie and pixstats.**

### 2.7.2 Memory Simulation Trace

Now suppose one needs memory reference traces for some type of memory system simulation. The method to generate the trace is similar to that explained above. To create a reference trace using pixie, type

```
pixie -idtrace cc1
```

which modifies the binary to record both instruction and data references. Using `-itrace` or `-dtrace` will give just instructions or just data respectively. Typing

```
idt -c cc1
```

will instrument an Intel architecture binary to record a cache line trace. In this trace, all data references will be output, but only one instruction reference will be output per cache line. This reduces the number of instruction reference entries which must be recorded. The cache line size can be adjusted using the `-l` option. When `cc1.pixie` is executed, the trace is sent to a file descriptor. Using a program called `makepipe`, the trace can be piped directly to the simulator. IDtrace will send the output trace to a file, in this case `cc1.trc`. The trace can be send directly to the simulator by using standard csh pipe commands.

Technical reports for both tools give trace format descriptions as well as complete descriptions of command-line options and trace piping methods [48][58].

# CHAPTER 3

# Speculative Execution and Cache Performance

## 3.1 Speculative Execution

Parallel computation is an increasingly important research area in the quest for faster general-purpose microprocessors. Superscalar processor architectures exploit instruction-level parallelism to concurrently run multiple instructions per cycle. Current implementations can issue from two instructions to six instructions per cycle under certain conditions [5][25][39][74]. More aggressive designs have been announced and will be available soon [25][44][44][44][74]. Although the amount is hotly debated, research has shown that more instruction-level parallelism exists in frequently executed, non-numeric code [many papers]. However, architecture designers face many problems in harnessing this potential concurrency. One problem is that the length of an unbroken stream of instructions, a basic block, only averages between 3.0 - 6.5 instructions in applications such as those found in the SPEC benchmark suite [60]. Thus, to achieve high issue rates, instructions must be fetched beyond the basic block ending conditional branches. This can be accomplished by speculative execution which involves guessing the direction of the branch before the branch condition is computed and continuing execution in the predicted direction until the branch is resolved. If the prediction is incorrect, the processor state must be restored to the state prior to the predicted branch and execution resumed down the correct path.

## 3.2 Problem to Address

Due to their deep pipeline and high issue rates, future processors will speculatively execute many instructions prior to branch resolution. While this speculation enables instruction issue to continue during branch computation, it produces some unavoidable side effects. One effect is an increase in the number of instruction and data references. These extra, wrong references will increase memory bus traffic and could cause cache pollution. Cache pollution occurs when unnecessary lines are brought into the cache which displace needed data, i.e., the cache becomes polluted with non-referenced data. On the other hand, they could act as prefetching references, bringing data or instruction lines into the cache for later, correctly predicted path execution. Speculative execution requires that data writes be handled carefully. Usually cache writes are held until the branch is resolved and the write can be committed. Although the added recover complexity would prohibit allowing a speculative write to modify an existing cache line, it might be advantageous to allocate a cache line for a speculative write miss regardless of whether the write is committed. The changes in the execution model due to speculation warrant a re-examination of memory system behavior. In this chapter, the effect of deep speculation on cache performance will be examined.

## 3.3 Experimental Procedure

To study these effects, an instrumentation tool called spex was developed to generate speculative execution traces. The traces were fed into a trace-driven memory simulator to observe the speculation effects. This section describes the trace generation and simulation procedure and defines the processor and memory model for which the next section's results are valid.

### 3.3.1 Trace Generation

A new instrumentation tool, spex, was created to generate speculative reference traces and was described in detail in Section 2.5.2. To reiterate, spex takes as input an application binary, a speculation depth, and a choice of branch prediction algorithm. It produces a modified executable which, when run, will output a memory reference trace resembling the trace produced by a speculative processor. Inputs to spex along with the binary are a branch prediction scheme choice and a fixed value, $n$, which is the number of cycles required to resolve each conditional branch. In reality, branch resolution times in pipelined and superscalar machines will vary depending upon the type of conditional branch, the data dependencies between instructions currently in the pipeline, the number, type, and data dependencies of instructions waiting to be issued, and processor exceptions. Thus, the results should be interpreted as bounds for machines that speculate no more than $n$ instructions past a conditional branch.

Since the spex model and a true speculative processor have different viewpoints of speculative paths, it is important to clarify some terms used in this chapter. A speculative path is the execution path taken after a conditional branch prediction. Spex can immediately classify the speculative path as a correct path (correctly predicted direction) or a wrong path (mispredicted direction) by comparing the predicted direction with the actual result of the branch. Thus, it can tag references generated on these paths as correct path references or wrong path references. It must be remembered, however, that a real speculative processor cannot foresee the accuracy of the prediction until the branch is resolved some distance down the speculative path. Therefore, an immediate cache action such as line allocation cannot be based upon the correctness of the reference.

### 3.3.2 Platform and Benchmarks

The applications used in this study are the SPEC92 C benchmarks, see Table 3.1 [60]. The applications were compiled using gcc with the highest level of optimization. The platform for the simulations is an Intel i486 50MHz computer running USL Unix SYSV R4.

It is not my intent to specify an optimal cache configuration for the modeled architecture based only upon application generated references. Several studies have shown traces generated only by single process application code, specifically SPEC benchmark applications, underestimate the cache load of the actual system and will thus lead to incorrect predictions of optimal cache configurations [9][18]. Instead, the differences in cache performance between the speculative and non-speculative execution models for various cache configurations were examined. While the magnitude may vary, these differences will occur both in application and system code. It is suspected that the problems detected using application-only traces will be amplified when full system activity is considered.

| Program | Description | i486 Instructions |
|---|---|---|
| compress | Unix compression utility | 60 million |
| eqntott | Boolean equation to truth table translator | 1666 million |
| espresso | Logic minimization tool | 531 million |
| sc | spreadsheet program | 848 million |
| xlisp | XLISP interpreter solving 8 queens problem | 991 million |

**Table 3.1 The SPEC92 C benchmarks used in the following studies.**

### 3.3.3 Processor and Memory Model

Two trace-driven cache simulators were used to gather results: a modified version of Tynero and a multicache simulator designed to monitor prefetching and pollution

effects [50]. Both simulators distinguish between correct path references and misses, and wrong path references and misses.

The processor/memory model is a multi-issue, pipelined processor with out-of-order execution requiring deep speculation to maintain a high instruction issue rate. All mispredicted paths are assumed to execute down a constant, predefined depth before branch resolution. The processor has a first level set-associative, non-blocking cache with an LRU replacement policy. Cache line size is fixed at 32 bytes for all experiments. The cache completes all outstanding memory requests. This means that if a wrong path reference causes a read miss, the cache line is updated even if the branch is resolved before the request to memory is completed. Speculative path writes are held until the branch is resolved so that the cache always contains valid data. Finally, the cache write policy is copy-back with write allocation for correct writes.

Two branch prediction algorithms are used in this study. Algorithm 1 is a two-level adaptive scheme with a 512 entry, 4-way associative register table and a 4096 entry pattern table containing 2-bit saturating counters [75]. This algorithm is expensive in terms of hardware but it achieves excellent accuracy for the benchmarks in our study, see Table 3.2. It is the algorithm used unless otherwise specified. Algorithm 2 uses a simpler history table of 1024, 2-bit saturating counters [56]. No tags are recorded so multiple addresses which map to the same table entry will share the counter.

| Program | Alg. 1 | Alg. 2 | Program | Alg. 1 | Alg. 2 |
|---|---|---|---|---|---|
| cc1 | 88 | 85 | espresso | 93 | 86 |
| compress | 89 | 87 | sc | 96 | 92 |
| ear | 96 | 94 | xlisp | 95 | 85 |
| eqntott | 96 | 82 | | | |

**Table 3.2 Branch Prediction Accuracies.**

## 3.4 Results

### 3.4.1 Total Data Traffic

The main result of the study is that, in most benchmarks, the total data memory traffic is not significantly increased by deep speculation. Figure 3.1 shows the percent increase in total (correct and wrong path) misses of speculative execution over that of non-speculative execution.

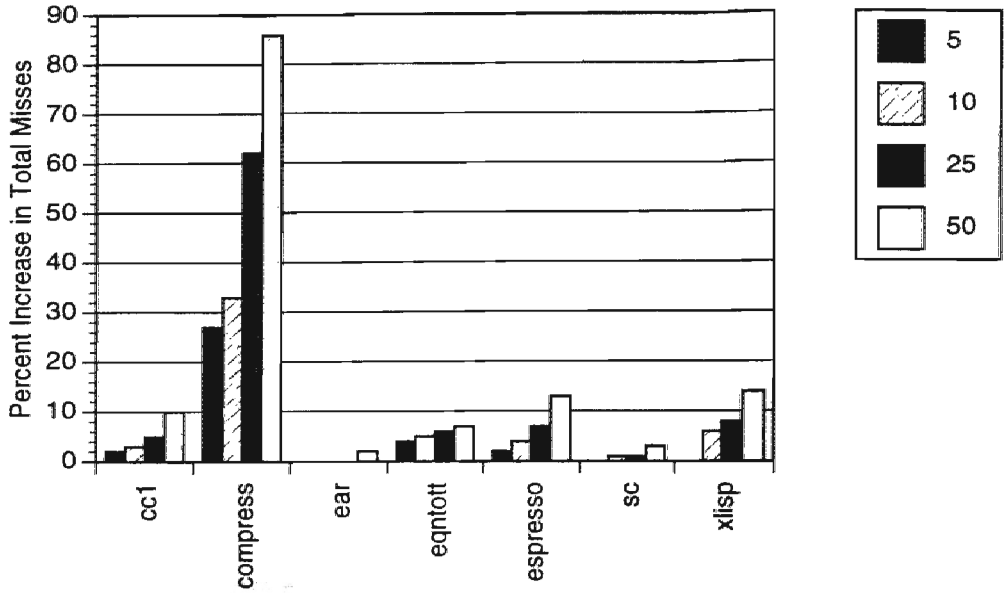


**Figure 3.1 Data Miss Increase vs. Speculation Depth - Speculative data miss percentage increase over that of non-speculative execution. The legend show the speculation depth. The cache size is 32 Kbytes with 4-way set associativity.**

Figure 3.2 shows that there is a substantial increase in the number of data references due to speculation (up to 75% for 50 deep speculation) yet, from Figure 3.1, for most benchmarks deep speculation increases the total number of data cache misses by only around 15% over that of non-speculation.

The traffic ratio, defined as

$$\text{traffic ratio} = \frac{(\text{cache misses} + \text{copy-backs}) * (\text{cache line size})}{(\text{memory references without a cache}) * (\text{word size})},$$

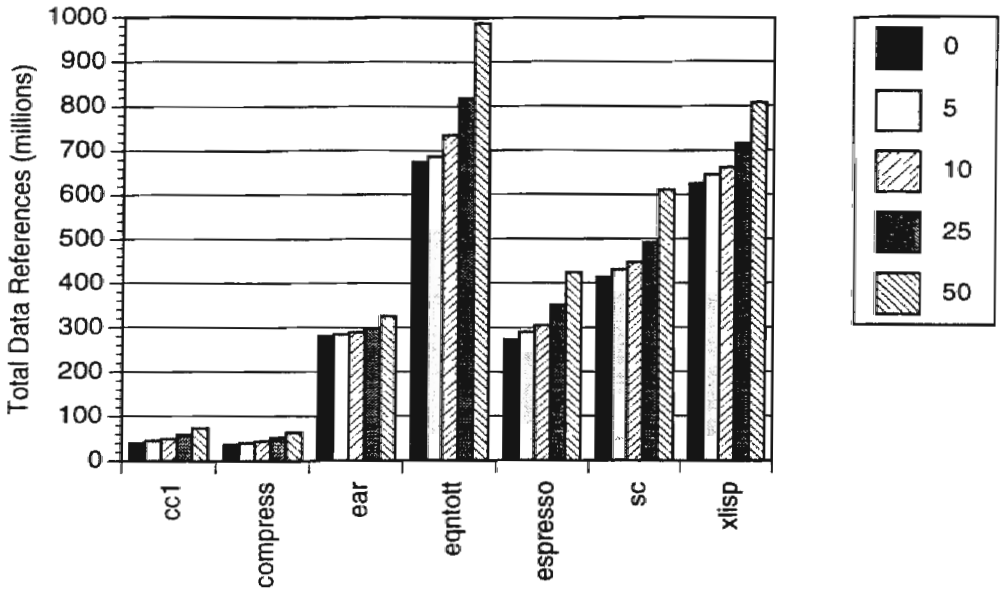


**Figure 3.2 Total Data References. The zero depth speculation represents no speculation.**

is a measure of the efficiency of the cache. Figure 3.3 shows that the cache usually becomes more efficient, traffic ratio decreases, as speculation increases. The exception to

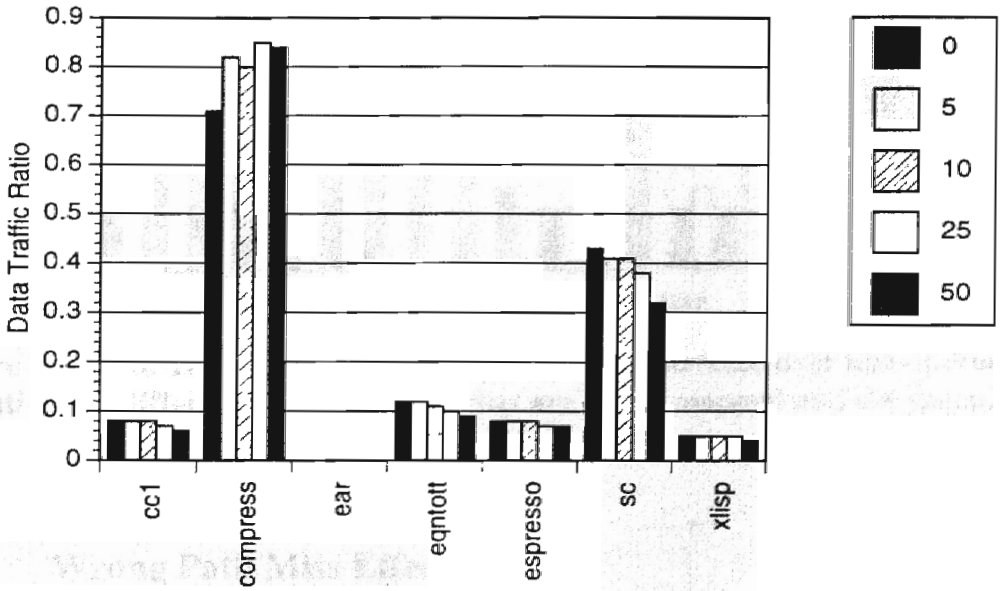


**Figure 3.3 Traffic Ratio - The cache size is 32K and is 4-way set associative.**

this is compress. A 0.71 traffic ratio without speculation from Figure 3.3 reveals that the cache is not well suited for this application. Repeating the experiment using a smaller line size resulted in a lower miss ratio. Furthermore, 8% of the non-speculative references missed the cache when speculating 25 instructions deep. Compress has poor locality down several mispredicted paths which repeatedly get executed. The large line size magnifies the traffic problem.

The next two figures display how the bandwidth increase changes with increasing cache size and increasing associativity. As would be expected, configuration changes which reduce pollution generally reduce the additional bandwidth required for speculative execution. The inconsistent data for large associativities is probably due to the small number of misses.

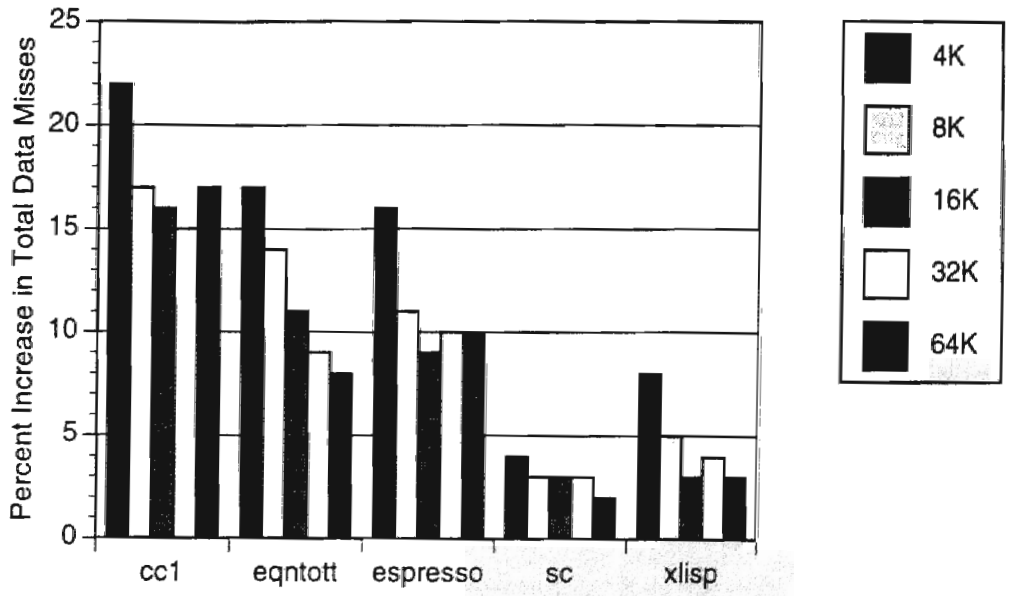


**Figure 3.4 Data Traffic for Different Cache Sizes - Increase over non-speculative execution for different cache sizes. Caches are direct mapped and the speculation**

### 3.4.2 Wrong Path Miss Effects

There are several ways that the additional wrong path references can affect the cache. First, they can prefetch data which will later be used during correct path execution.

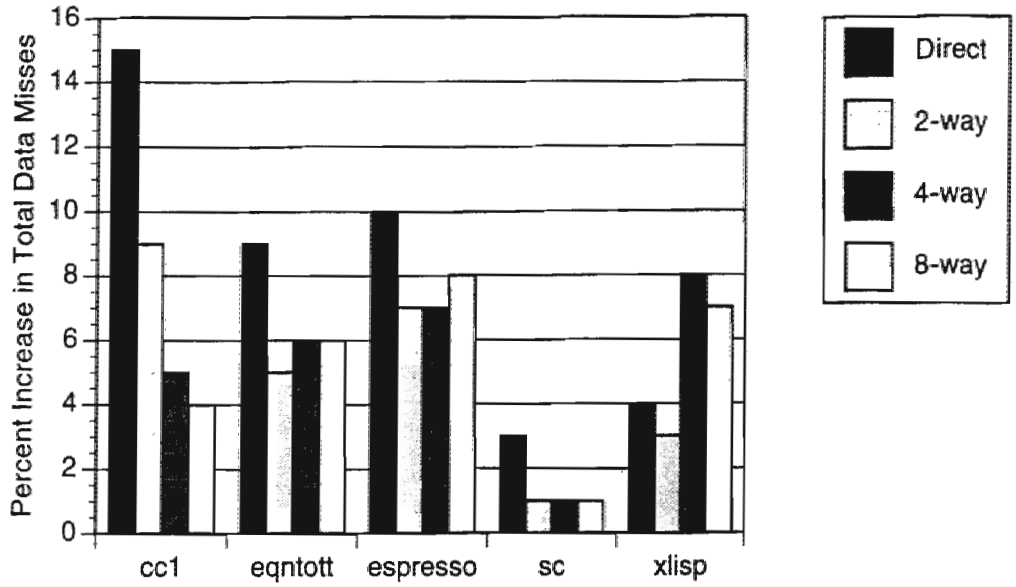


**Figure 3.5 Data Traffic for Different Associativities - Increase over non-speculative execution for different cache associativities. Caches are 32K and the speculation**

A miss during correct path execution can then be avoided if this data is accessed before being displaced in the cache. These wrong path misses are called prefetch misses and they reduce the number of correct path misses. On the other hand, pollution misses increase the number of correct path misses. They are caused by wrong path read misses allocating lines which displace lines needed for later correct path execution. To discuss these effects it is helpful to define a ratio $P_n$, where

$$P_n = \frac{\text{\# of correct path misses for a given cache with speculation depth } n}{\text{\# of non-speculative misses for a given cache}}$$

Figure 3.7 gives $P_n$ for speculation down 25 instructions using the higher accuracy prediction algorithm. It shows that speculation reduces the number of correct path misses so prefetching must dominate the pollution caused by wrong path reads. Deeper speculation increases the number of wrong path references, thereby increasing the prefetch misses and further reducing $P_n$. Another way to increase the number of wrong path references is to use a less accurate prediction algorithm to execute more wrong paths.

Figure 3.7 shows a more dramatic reduction in $P_n$ due to using the less accurate Algorithm 2. Notice in compress, $P_n$ increases with depth signaling that pollution, rather than prefetch, is the dominant effect.

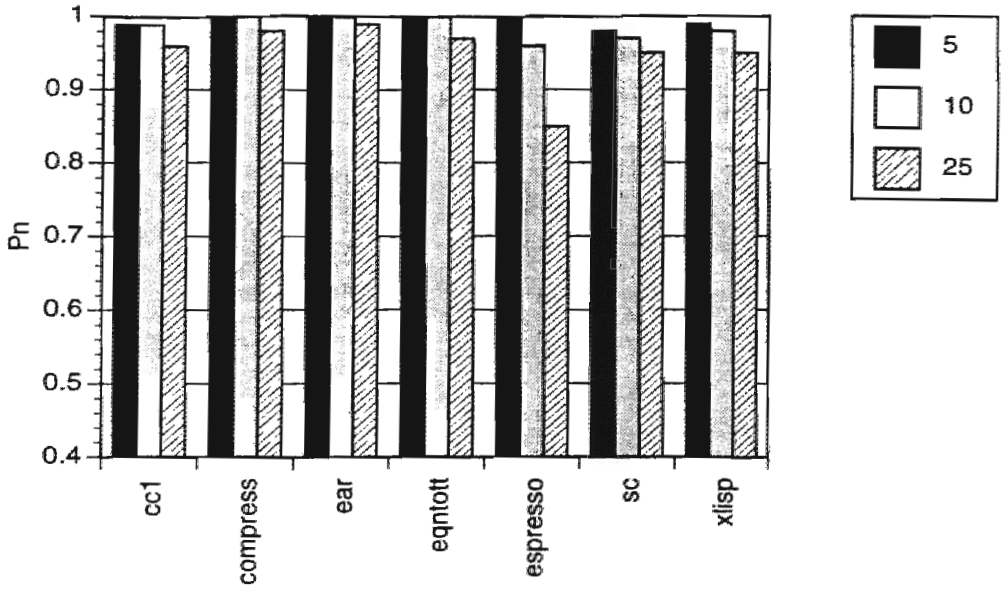


**Figure 3.6 $P_n$ Using Branch Prediction Algorithm 1 - The cache is 16K with 4-way set associativity.**

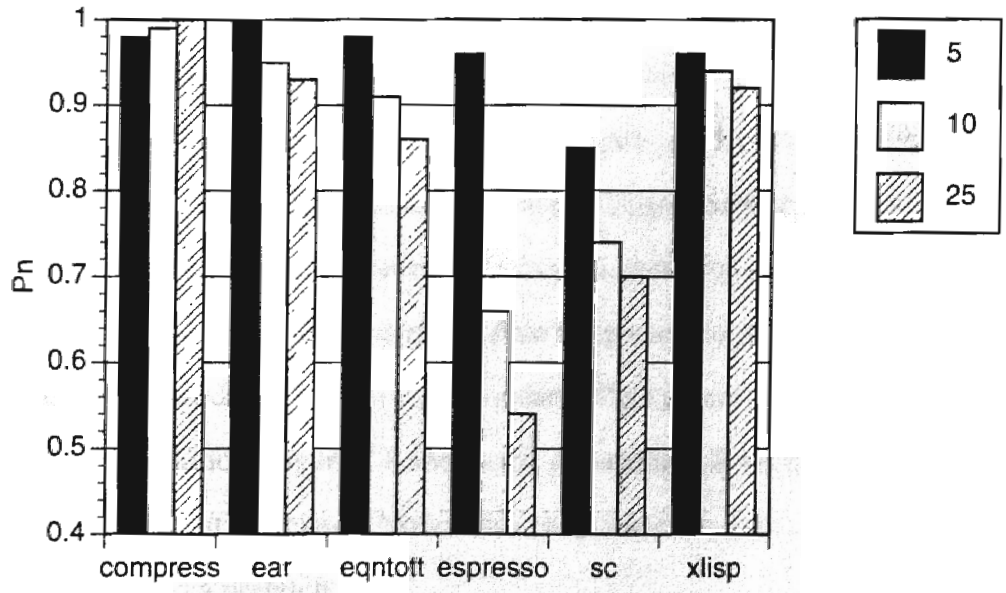


**Figure 3.7 $P_n$ Using Branch Prediction Algorithm 2 - The cache is 16K with 4-way set associativity.**

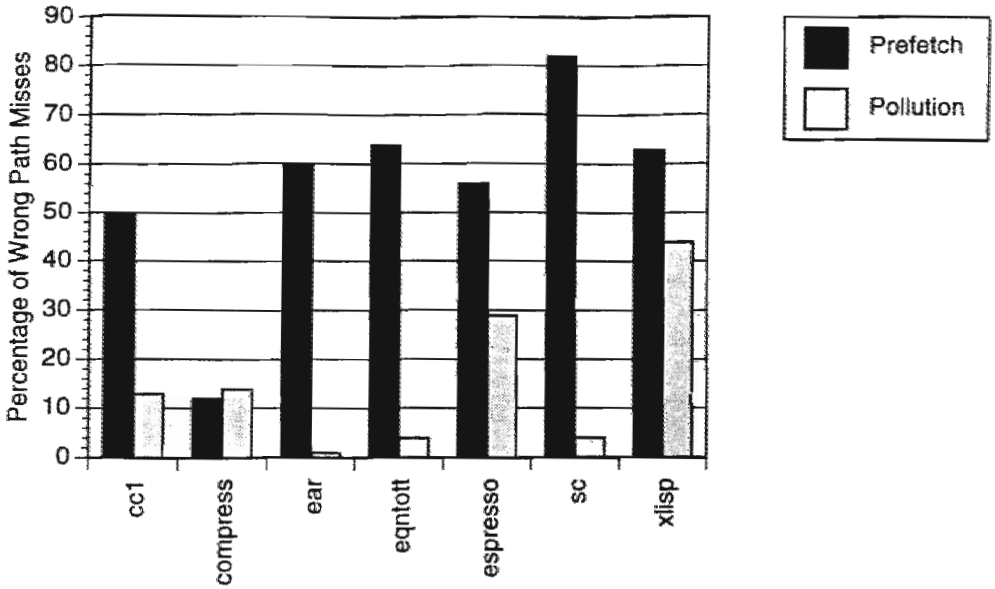


**Figure 3.8 Breakdown of Prefetch and Pollution Effects - The cache size is 16K with 4-way set associativity.**

The cache simulator was modified to directly count the number of prefetch and pollution misses caused by wrong path references and it was found that they alone did not completely account for the change in $P_n$. It was observed that wrong path cache hits can also reduce (or increase) correct path misses. They do so by reordering the lines in a LRU set associative cache. For example, suppose that a wrong path reference hits a least recently used line and thus promotes it to most recently used. Then suppose a cache read miss occurs in the same set. A different line will be displaced than if execution were non-speculative. If the line is needed during correct path execution this action will have avoided a miss. If the displaced line rather than the promoted one is needed an additional miss is incurred. Reordering has only a secondary effect on cache misses compared to that of prefetch and pollution. Figure 3.8 shows the percentage of wrong path misses which were prefetch or pollution misses. Notice that, with the exception of compress, over 50% of misses performed a prefetch.

### 3.4.3 Wrong Path Writes

During speculative execution our processor model must delay all write references occurring down a speculative path until the branch is resolved. This requires write buffers between the processor and cache to hold the address and value of these writes until branch resolution. If the branch was predicted correctly the writes are released to the memory system. Otherwise, the suspended writes are squashed.

Figure 3.9 shows the number of instructions issued for a particular number of unresolved conditional branches. It is surprising to notice that often many conditional branches are outstanding and that seldom are instructions issued non-speculatively. Even with a speculation depth of 10, 85% of instructions are issued speculatively. When the depth increases to 30, the percentage increases to over 95%. Since most instructions are issued speculatively, most writes will be temporarily suspended. This leads to several questions. How many buffers are necessary to hold most wrong path writes and what write allocation policy for the cache is appropriate?

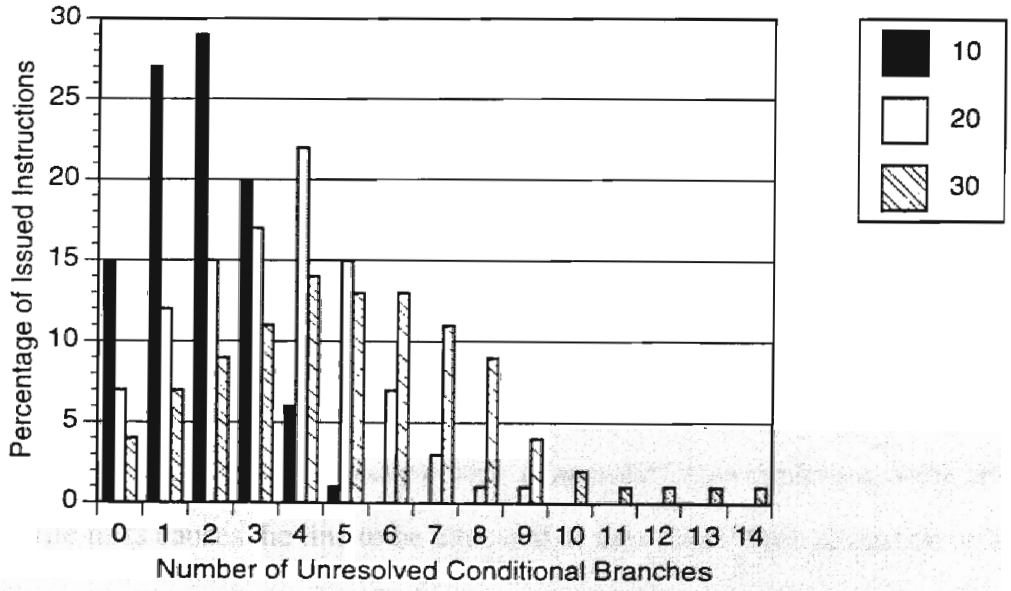


**Figure 3.9 Outstanding Conditional Jumps - Percentage of instructions issued per number of outstanding conditional jumps.**

### 3.4.4 Speculative Write Buffers

The instrumented benchmarks were run and the number of writes in speculative paths were counted to estimate the number of needed buffers. Figure 3.10 shows the results. To fully execute 90% of the speculative paths, 4, 6, and 10 write buffers are required for depths of 10, 20, and 30 respectively. It must be remembered that paths with more write references than available buffers can still partially complete before instruction issue must be halted.

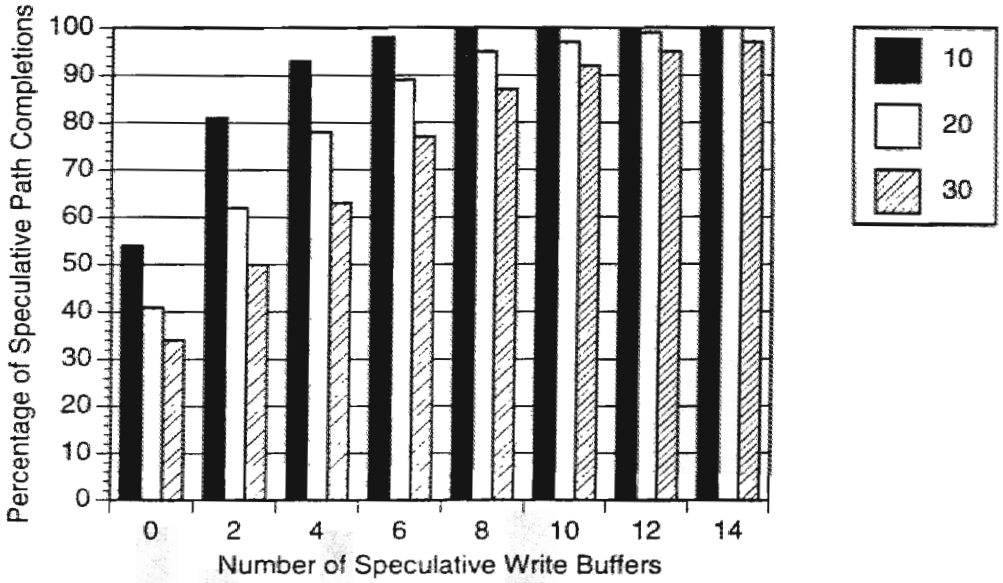


**Figure 3.10 Number of Speculative Write Buffers - Percentage of speculative paths fully executed for various numbers of speculative write buffers.**

### 3.4.5 Wrong Path Write Allocate

Most caches which use a copy-back write policy also implement write allocate, i.e., a write miss causes the line to be allocated in the cache. Write allocation on a speculative processor can be handled in several ways. One way is to wait until the branch is resolved and allocate lines for the correct writes in the cache write buffers. Another way would be to allocate cache lines for writes as they are being suspended during speculative execution. This would have the desirable effect of prefetching the lines so that some lines

could be in the cache at resolution time. But writes produced by mispredicted path will also cause write allocations and this has an unclear overall effect on cache performance. Allocations produced by mispredicted paths which are never taken during correct execution will increase memory traffic and cache pollution. However, wrong path write allocation might also prefetch lines for later correct path execution. Figure 3.11 compares the effects of allocating cache lines for all speculative writes or just for committed writes after branch resolution. The data shows allocating all writes increases the total memory traffic by a negligible amount. Furthermore, allocating wrong path writes performs a small amount of prefetching for later correct path memory references. Therefore, it is beneficial to allocate cache lines on all speculative write references prior to branch resolution.

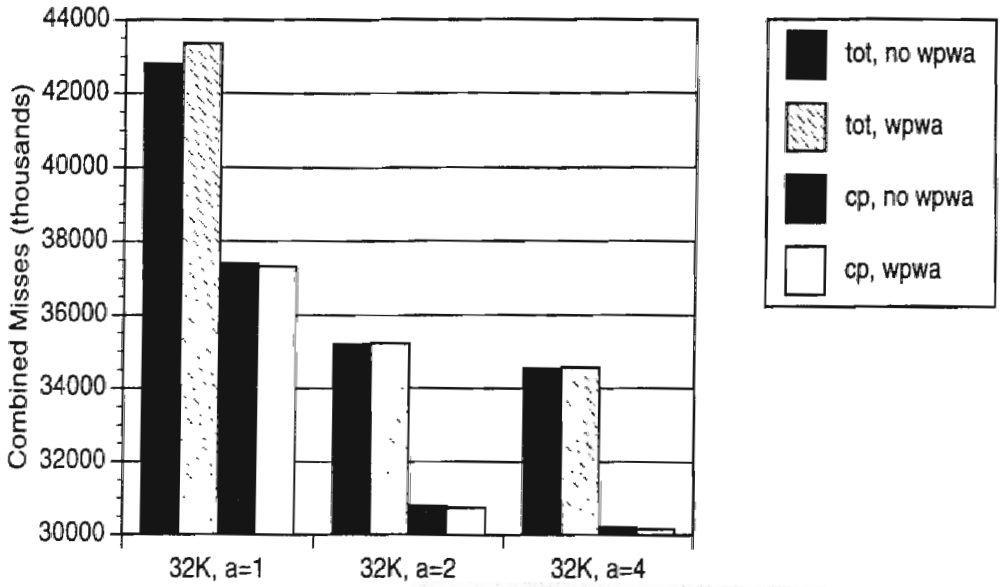


**Figure 3.11 Wrong Path Write Allocation - Effect on memory traffic when wrong path writes are allocated in the cache.**

### 3.4.6 Instruction Prefetching

This study has focused on data cache behavior because the SPEC benchmarks do a poor job of exercising even small instruction caches as described in [18] and as further shown in Section 4.4.3. Pollution and prefetch effects cannot be adequately observed

when the full working set fits into the cache. However, it is likely that instruction references produced during mispredicted paths could perform prefetches for later instruction references. Table 3.3 shows that over 50% of the lines which were allocated in the instruction cache during mispredicted paths were then accessed during correct paths executed later in the program. Thus, half of all wrong path instruction misses could prefetch useful instruction cache lines. Unfortunately, because misprediction is infrequent, few additional lines are referenced. Thus, the prefetch effect is small and will get smaller as prediction accuracy increases. An area of further study is the potential benefit of aggressively prefetching lines down wrong paths. Since accurate prediction limits the number of wrong path executions, this would entail allocating instruction cache lines from both the taken and not taken execution paths.

| Program | Reuse % |
|---|---|
| cc1 | 60 |
| compress | 52 |
| ear | 66 |
| eqntott | 60 |
| espresso | 68 |
| sc | 56 |
| xlisp | 70 |

**Table 3.3 Wrong Path Instruction Reuse - The percentage of instruction cache lines allocated during wrong path execution which were later referenced during correct path execution.**

## 3.5 Summary

The major result of this study is that deep speculation does not significantly alter cache performance in most of the tested applications. Therefore, future processors which rely on deep speculation to maintain high issue rates are unlikely to suffer severe performance problems due to increased cache miss cycles or bus traffic caused by

mispredicted path execution. However, the study did give insight into the behavior and effects of speculative cache accesses:

- In all except one of the benchmarks, data misses increased by less than 15% for speculation depths of up to 50 instructions.
- Data traffic ratio increases as speculation depth increases. This means that data caching is more effective as speculation increases.
- Correct path data misses actually decrease as speculation increases. This suggests that wrong path references act as data prefetches.
- The prefetch effect usually far outweighs the pollution effect of wrong path misses. In the benchmarks studied, between 50% and 80% of the mispredicted path references acted as prefetches for later correct path references.
- Mispredicted path instruction references are also later referenced in the program between 50% and almost 70% of the time.
- Because of prefetching, a slight gain is achieved by allocating cache lines for all speculative write misses rather than only allocating lines for confirmed write misses.
- Better branch prediction reduces the prefetch effect because it reduces the number of wrong paths that are executed.
- With speculation depth of as little as 10 instructions, 85% of the instructions are issued speculatively, and the median number of outstanding branches during any instruction issue is two.

The idea that executing mispredicted paths has positive benefits is both surprising and intriguing. It suggests that cache misses can be reduced by prefetching data down mispredicted paths. More specifically, it might show that a prefetching algorithm need not be concerned with trying to prefetch data only from the taken execution path. The next chapter describes a new prefetching algorithm which was inspired by these results.

# CHAPTER 4

# Instruction Cache Prefetching

## 4.1 Cache Miss Reduction

Instruction cache misses are detrimental to the performance of high-speed microprocessors. As the differential between processor cycle time and memory access time grows and the degree of instruction-level parallelism in superscalar architectures increases, the performance degradation caused by cache misses will become even more apparent. Designers have proposed several strategies to increase the performance of the cache memory systems which will be implemented in next-generation microprocessors. The option most often used in the past has been to increase the cache size and/or its associativity. However, this strategy consumes additional chip area and, since compulsory misses are unaffected, becomes less effective as the number of cache sets or associativity increases. In addition, increasing the cache associativity lengthens the cache access time and could adversely affect the chip's overall cycle time [23].

To improve performance while retaining the small size and speed of a direct-mapped cache, Jouppi proposed adding a small buffer called a victim cache to a conventional direct-mapped cache design to improve performance [30]. Victim caching reduces the number of conflict cache misses by holding onto recently displaced lines. When a conflict miss occurs, the displaced line is stored in the small (1 to 5 entry) fully-associative victim cache. A cache lookup then involves a parallel check in the main cache and the victim cache. If the access hits the victim cache, the lines in the main and victim caches are swapped and a conflict miss is avoided.

Farrens and Pleszkun propose adding two instruction queues between the fetch unit and a small on-chip instruction cache [16]. In addition, a prepare-to-branch (PBR) instruction is added to the architecture's instruction set. Instructions are fetched from one queue, the instruction queue (IQ), while the other queue, the instruction queue buffer (IQB), is being filled with sequentially located instructions or instructions beginning at a target address if a PBR instruction is decoded. The IQB supplies the IQ with instructions and can initiate a cache miss in advance of instruction fetch. Studied instruction cache sizes were 640 bytes or less and the system's effectiveness relies on the PBR instruction.

Cache prefetching is another method to increase cache performance and has been widely studied [16][20][24][26]. Prefetching is an attempt to fetch lines from memory into the cache before their instructions are referenced by the processor's fetch unit. To be effective, the prefetch strategy must accomplish two things. It must be able to guess which cache lines will soon be referenced and it must initiate the prefetch requests far enough in advance of instruction fetch so that the miss latencies are significantly reduced or eliminated entirely. Theoretically, an optimal prefetch algorithm could remove all cache misses by prefetching all instructions immediately before they are needed. Unfortunately, non-sequential program flow makes it impossible for the prefetcher to always predict the correct execution direction. Much work has been done to develop methods which anticipate the direction of program flow and to prefetch instructions in this direction. In this chapter, a new prefetching algorithm is proposed which makes no attempt to predict the correct direction. In fact, it relies heavily on prefetching the wrong direction. Not only does this method outperform previously proposed prefetching schemes, but it does so at a lower hardware cost.

The word prefetch can be found in several different contexts in current computer literature and it is important that I clarify exactly what I mean by cache prefetching and what problem this work intends to address. Cache prefetch algorithms are studied which reduce instruction cache misses by prefetching instruction lines from memory into the

cache. A source of confusion is that the term prefetch is also used to denote the act of fetching multiple words from the cache into the fetch unit of the execution pipeline. The goal of cache prefetching is to reduce cache misses. The goal of what we will call instruction prefetching is to assist in instruction decode or to increase the instruction issue rate. The Intel Architecture processors (i486 and Pentium) utilize cache-to-buffer prefetching to alleviate the decode problems associated with variable instruction size and complex encodings [28]. Superscalar processors like the Alpha or Power2 architectures prefetch multiple lines from the cache so that multiple instructions can be issued per cycle even during branch execution [14][74]. The PowerPC also prefetches multiple instructions from the cache into prefetch buffers [74]. It does this primarily because the instruction fetch must share a single port to the unified cache with data memory requests and thus it cannot fetch an instruction from the cache every cycle. What is important to note here is that these four processors (and others like them) perform instruction prefetching and not cache prefetching. In all of the above examples, instruction prefetching never initiates requests to memory. Instruction prefetching stops if the correct lines are not found in the cache. Finally, another use of the term prefetching applies to data prefetching. Data prefetching attempts to reduce data cache misses by exploiting a program's data access patterns in order to prefetch data from memory [10][32]. This chapter does not address data prefetching issues.

## 4.2 Prefetching Methods

Instruction prefetching can be done passively by modifying the cache organization to promote prefetching or by including additional hardware mechanisms to execute an explicit prefetching algorithm.

### 4.2.1 Long Cache Lines

The simplest form of prefetching is the use of long cache lines [55]. When a line is replaced, new instructions are brought into the cache in advance of their use by the CPU, thereby reducing or eliminating miss delays. Longer cache lines also reduce the amount of space required for tag storage. The disadvantages are that longer lines take longer to fill, they increase memory traffic, and they contribute to cache pollution due to the larger replacement granularity. A long instruction line which is only partially accessed will displace many existing instruction words which may be needed in the future. As can be seen in Figure 4.1, there is a point after which longer line sizes are ineffective.

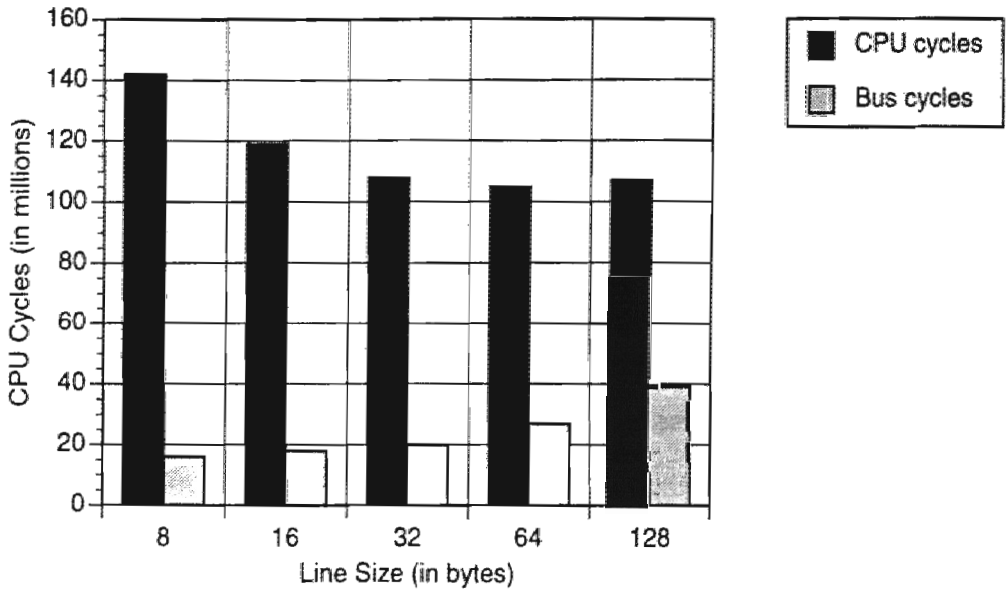


**Figure 4.1 Effect of Increasing Line Size - The cache is 8K bytes with a constant 8 byte refill. The application is gcc.**

### 4.2.2 Next-Line Prefetching

Another approach to instruction prefetching is next-line prefetching. It tries to prefetch sequential cache lines before they are needed by the CPU's fetch unit. In this scheme, the current cache line is defined as the line containing the instruction currently

being fetched by the CPU. The next line is the cache line located sequentially after the current line. If the next line is not resident in the cache, it will be prefetched when an instruction located some distance into the current line is accessed. This specified distance is measured from the end of the cache line and is called the fetchahead distance, see Figure 4.2. Next-line prefetching predicts that execution will fall-through any conditional branches in the current line and continue along the sequential path. The scheme requires little additional hardware since the next line address is easily computed. Unfortunately, next-line prefetching is unlikely to reduce misses when execution proceeds down non-sequential execution paths caused by conditional branches, jumps, and subroutine calls. In these cases, the next line guess will be incorrect and the correct execution path will not be prefetched. Performance of the scheme is dependent upon the choice of fetchahead distance. If the fetchahead distance is large, the prefetch is initiated early and the next line is likely to have been received from memory in time for CPU fetch. However, increasing the fetchahead distance increases the probability that a branch will be encountered in the current line and execution will continue in a non-sequential direction rendering the next-line prefetch ineffectual. This useless prefetch increases both memory traffic and cache pollution. In spite of these shortcomings, next-line prefetching has been shown to be an effective strategy, sometimes reducing cache misses by 20-50% [25].

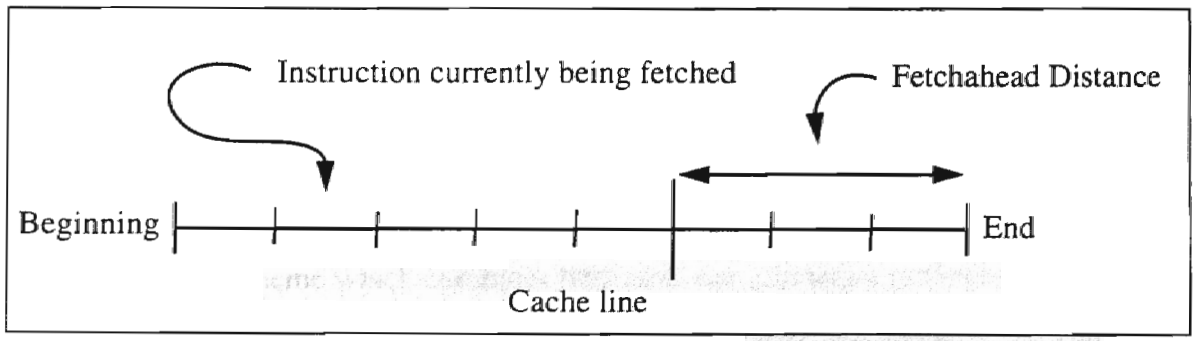


**Figure 4.2 Fetchahead Distance - In next-line prefetching, once instruction fetch occurs within the fetchahead distance, the next consecutive cache line will be prefetched.**

### 4.2.3 Target-Line Prefetching

Target-line prefetching addresses next-line prefetching's inability to correctly prefetch non-sequential cache lines. When instructions in the current line are being executed, the next cache line accessed might be the next sequential cache line or it might be a line containing the target of a control instruction found in the current line. Since unconditional jump and subroutine call instructions have a fixed target and conditional branch instructions are often resolved in the same direction as they were when last executed, a good heuristic is to base the prefetch on the previous behavior of the current line, i.e., prefetch the line which was referenced next the last time the current line was executed. Target-line prefetching uses a target prefetch table maintained in hardware to supply the address of the next line to prefetch when the current line is accessed. The table contains current line and successor line pairs. When instruction execution transfers from one cache line to another line, two things happen in the prefetch table. The successor entry of the previous line is updated to be the address of the new current line. Also, a lookup is done in the table to find the successor line of the new line. If a successor line entry exists in the table and that line does not currently reside in the cache, the line is prefetched from memory. By using this scheme, instruction cache misses will be avoided or at least their miss penalty will be reduced if the execution flow follows the path of the previous execution.

### 4.2.4 Hybrid Schemes

A hybrid scheme which combines both next-line and target prefetching was proposed in Hsu and Smith [26]. In this scheme, both a target line and next line can be prefetched, offering double protection against a cache line miss. Next-line prefetching works as previously described. Target-line prefetching is similar to that above except that if the successor line is the next sequential line, it is not added to the target table. This saves

table space thus enabling the table to hold more non-sequential successor lines. They compared the performance of this scheme with next-line and target-line algorithms using supercomputer reference traces and their results were impressive - miss rates are reduced by 50-60%. In addition, they showed that the performance gain of the hybrid method was roughly the sum of the gains achieved by implementing next-line and target prefetching separately.

Performing target prefetching with the help of a prefetch target table is not without drawbacks, however. First, significant hardware is required for the table and the associated logic which performs the table lookups and updates. This requires additional chip area and could increase CPU cycle time. Secondly, the extra hardware has only limited effectiveness in that it cannot be used to remove certain types of misses. First time accessed code does not profit from table-based target prefetching since the table must first be set up with the proper links or current-successor pairs. Thus, compulsory misses are unaffected by target prefetching. Furthermore, unlike a branch prediction table, even when the correct information does exist in the table it cannot always be utilized. Upon re-execution of the code when the links are properly set, prefetching will only occur if the target line has been previously displaced from the cache. In the likely event that the line is still in the cache, the table entry space and lookup are wasted because prefetching is not needed. This suggests that target prefetching using a table is best suited for small caches with low associativity where lines are often displaced and then rereferenced. This was the proposed application environment in [26].

It is interesting to note several points common to the above schemes. One is that prefetch decisions are made at the cache line level. No instruction-specific information is used. This makes sense because a prefetch decision must be made early and many cycles may pass before instruction recognition can take place in the decode stage of the pipeline. Another point is that the above schemes try to predict the correct execution path and then prefetch only down the predicted path. For instance, using a small fetchahead distance will

bias the next-line prefetching scheme toward the correct path by lowering the probability of a control instruction being within the fetchahead distance. Target prefetching predicts that the correct direction in which to prefetch is the direction of the previous execution. Even though the hybrid algorithm may prefetch lines down the wrong path, since it sometimes prefetches both a next line and a target line for the current line, such actions are unintentional and rarely occur. Prefetching the correct path satisfies intuition because only lines soon to be executed should be prefetched. The alternative, fetching wrong path lines into the cache, will likely increase memory traffic and cause cache pollution.

### 4.2.5 Wrong-Path Prefetching

The work in the last chapter shows that the intuition expressed above is partially false. Executing instructions down mispredicted paths actually reduced the number of cache misses occurring during correct path execution. This suggests that prefetching instruction cache lines down mispredicted paths might have a positive result.

I propose a new algorithm called *wrong-path prefetching* which is similar to the hybrid scheme in the sense that it combines both target and next-line prefetching. The next line is prefetched whenever instructions are accessed inside the fetchahead distance as described earlier. The major difference is in the target prefetching component. No target line addresses are saved and no attempt is made to prefetch only the correct execution path. Instead, the line containing the target of a conditional branch is prefetched immediately after the branch instruction is recognized in the decode stage. Thus, both paths of conditional branches are always prefetched: the fall-through direction with next-line prefetching, and the target path with target prefetching. Unfortunately, because the target is computed at such a late stage, prefetching the target line when the branch is taken is unproductive. In this case, if the target address is not in the cache, a fetch miss and a prefetch request of the same line will be generated simultaneously. Similarly, prefetching unconditional jump or subroutine call targets is useless since the targets are always taken

and the prefetch address would be produced too late. To reiterate, the target prefetching part of the algorithm can only perform a potentially useful prefetch for a branch which is not taken. This is why the algorithm is called wrong-path prefetching. If execution returns to the branch in the near future, and the branch is then taken, the target line will probably reside in the cache because of the prefetch.

The hardware requirements for wrong-path prefetching are roughly equivalent to what is required for next-line prefetching since the target prefetch addresses are generated by the existing decoder and no target addresses are saved. The obvious advantage of wrong-path prefetching over the hybrid algorithm is that there is a lower hardware cost. The performance of wrong path prefetching might also compare favorably with other schemes. Wrong-path prefetching can prefetch target paths which have yet to be executed unlike the table-based schemes which require a first execution pass to create the cache line links. In addition, wrong-path prefetching should perform better than correct-path only schemes when there exists a large disparity between the CPU cycle time and the memory speed. This is because other algorithms try to prefetch down target paths which will be executed almost immediately, and if memory has a long latency, the prefetch may not be initiated soon enough. Conversely, wrong-path prefetching prefetches lines down a path which is not immediately taken thus it potentially has more time to prefetch the line from a slow memory before the path is executed. However, the performance of wrong-path prefetching does not come without cost. Unavoidably, prefetching down not-taken paths will put lines into the cache that are never accessed. This will increase both memory traffic and cache pollution. For the algorithm to be successful, the benefits of prefetching must overcome the added pollution misses. The extra traffic cannot be reduced, but memory bandwidth can be viewed as a hardware resource to be utilized to reduce the performance degradation caused by instruction cache misses.

Again, it should be emphasized that wrong-path prefetching is fundamentally different from the both-path instruction prefetching done in some current superscalar

processor designs. In these architectures, words from both paths are copied from the cache to the prefetch buffer. After one of the paths is executed, the wrong path words are removed from the buffer. Instruction prefetching never causes a memory-to-cache transfer, so the number of cache misses will not be affected. In the proposed wrong-path prefetching scheme, lines containing instructions from not-taken paths are routinely fetched from memory and stay resident in the cache.

## 4.3 Preliminary Feasibility Study

Since the wrong-path prefetching algorithm relies on prefetching target lines that are not taken, initial experiments were performed to isolate the effects of prefetching lines only down not-taken paths. The experiments were run on an i486 SysVR4 Unix platform using the SPEC benchmark `gcc` as the workload [60]. Traces were generated using IDtrace and then fed into a prefetch/multi-cache simulator [48]. The prefetch simulator was programmed to prefetch lines from only not-taken paths of conditional branches. Therefore, a taken branch caused the cache line containing the fall-through address to be prefetched. A not-taken branch caused the target line to be prefetched. No other lines were prefetched. The left graph in Figure 4.3 compares the number of cache misses when no prefetching was performed with the miss performance of 4 variations of the described wrong-path-only prefetching algorithm. The WPO-1, 2, and 3 algorithms represent prefetching 1, 2, and 3 consecutive wrong path lines respectively. For instance, the WPO-2 algorithm would prefetch two cache lines down the not-taken direction of every conditional branch. The profile algorithm, PROF-1, will be explained shortly. It can be seen that prefetching only the lines from paths not immediately executed exhibits surprisingly good results. The prefetching effect far outweighs the extra pollution generated by sometimes prefetching unused cache lines.

One problem with this prefetching approach is the large amount of extra traffic generated, as shown by the right graph in Figure 4.3. One possible way to reduce this

traffic would be to eliminate the prefetches of paths which are never taken. Prefetching these paths cause extra traffic and contribute to cache pollution. Thus, prefetching should not occur for conditional branches which are always taken or always not-taken. To examine the effects of removing these wasted prefetches, gcc was profiled to create a list of the conditional branches which both the target and the fall-through paths are taken sometime during execution. Prof-1 uses this profile data to decide when to prefetch. It prefetches a cache line from the not-taken path of a conditional branch only if the branch is in the profile list, i.e., if both directions of the branch are taken sometime during program execution. Since some branch paths are never taken, the number of prefetches will be reduced. However, this should not degrade the overall performance because only prefetches applied to non-executed paths are removed. In fact, it could be supposed that Prof-1 would have slightly better performance than WPO-1 because of the reduction in cache pollution.

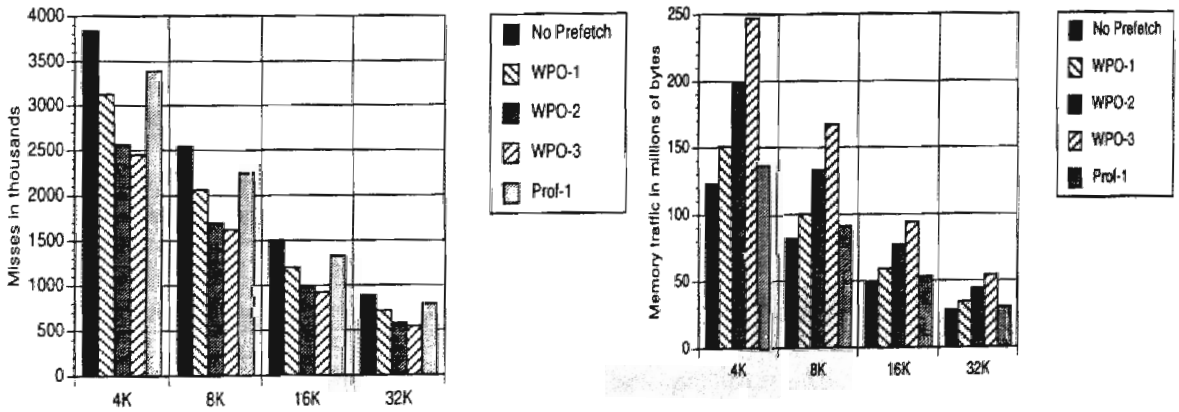


**Figure 4.3 Wrong-Path Prefetching Feasibility - Results from prefetching only down the not-taken direction of conditional branches. The cache is direct-mapped with 32 byte line size. The benchmark was gcc run on an i486 Unix platform.**

Surprisingly, comparing the WPO-1 and Prof-1 results in Figure 4.3 shows that the expected traffic reduction was accompanied by an unexpected decrease in performance. The unanticipated poor performance of Prof-1 has two explanations. One is that the prefetched cache line contains more than just the not-taken path instructions. The 32-byte

line size is large when compared to the average basic block size and probably contains 4 to 8 basic blocks. Therefore, it contains other paths which, in fact, are executed. Another explanation is that multiple branches can point to the same target and one branch can prefetch for another. Suppose that Branch A and Branch B share the same target address, Branch A is never taken, i.e., control is never passed from Branch A to the target address, and Branch B is taken sometime during the program's execution. Prefetching the not-taken direction of Branch A would bring the target line into the cache and eliminate a cache miss caused by a later execution of Branch B.

These preliminary experiments show that prefetching down not-taken paths can significantly reduce cache misses at the cost of higher memory traffic. Methods which attempt to select which branches to prefetch reduce the memory traffic but can also impair the algorithm's ability to reduce cache misses.

## 4.4 Experimental Method

This section details the experimental procedure used to study the performance characteristics of the previously mentioned prefetching algorithms.

### 4.4.1 Memory System Simulator

A detailed, trace-driven memory system simulator was written to model the performance of the instruction cache, memory bus, prefetch unit, and processor fetch unit in a typical current or next-generation microprocessor. The simulator can model the behavior of a wide range of system configurations by allowing the user to vary cache parameters (set size, associativity, line size, line refill rate, flush interval), prefetch algorithm parameters (algorithm type, fetchahead, target table size and associativity, target threshold), and rough processor and system characteristics (instruction issue rate, prefetch unit structure, memory ports, number of cache tag ports, and memory wait cycles). Furthermore, the simulator displays detailed event statistics such as prefetch success or

failure, prefetch initiation time, and resource utilization times which enables the user to gain insight into the interaction between prefetching and cache behavior.

### 4.4.2 Memory System Model

#### 4.4.2.1 Microprocessor/Memory Model

The modeled base microprocessor is a single issue, pipelined architecture with RISC-like properties. Each instruction takes one cycle to execute and all instructions are of uniform 4 byte length. The CPU clock speed is assumed to be high so the disparity between clock speed and memory access time is large. Conventional split L1 instruction and data caches are implemented which connect directly to memory or to a second-level, L2, cache. Instruction cache accesses are not pipelined and cache hits complete in one cycle.

Instruction cache configurations vary widely in current generation processors, see Table 4.4. The values for the base model cache parameters are given in Table 4.5. Each of these parameters will be varied in different comparison experiments discussed in Section 4.5. If the value of a cache parameter is not mentioned in a figure title, it is the default value listed in the table.

The base memory architecture was chosen to be readily constructed with present-day technology. This is important in the performance comparison of prefetch algorithms for several reasons. First, the following simulation results demonstrate performance which is guaranteed achievable and does not depend upon possible future technological advances. Secondly, keeping the hardware simple is less likely to negatively influence clock speed which would negate perceived performance gains. At the end of the chapter, I

will speculate how the studied algorithms and prefetching in general might be affected by future technology.

| Processor | Cache Type | Size (Kbytes) | Assoc. | Line Size (bytes) | Refill Rate (bytes/cycle) |
|---|---|---|---|---|---|
| Intel i486 [28] | Unified | 8 | 4-way | 16 | 4 |
| Intel Pentium [5] | Split | 8 | 2-way | 32 | 8 |
| AMD K5 [44] | Split | 8 | 2-way | 16 | 8 |
| DEC 21164 [44] | Split | 8 | Direct | 32 | 16 |
| Hitachi HARP-1 [44] | Split | 8 | Direct | 32 | 8 |
| PowerPC 601 [74] | Unified | 32 | 8-way | 64[a] | 8 |
| PowerPC 603 | Split | 8 | 2-way | 32 | 8 |
| PowerPC 620 [44] | Split | 32 | 8-way | 64 | 8 |
| IBM Power [74] | Split | 8 | 2-way | 64 | 8 |
| IBM Power2 [74] | Split | 32 | 2-way | 128 | 32 |
| MIPS R4000MC | Split | 8 | Direct | 16/32[b] | 16 |
| SGI TFP [25] | Split | 16 | Direct | 32 | 4 |

**Table 4.4 Cache Configurations for Some Current Microprocessors.**

a. 2 sectors/line - tagged per line but transferred on sector boundaries.
b. Configurable

| Parameter | Base Model Value |
|---|---|
| Cache Size | 16K bytes |
| Associativity | Direct |
| Line Size | 32 bytes |
| Refill Rate | 16 bytes/cycle |
| Wait Cycles | 4 |

**Table 4.5 Base Model Cache Parameter Values**

#### 4.4.2.2 The OMRB Structure

Prefetches can occur independently of instruction fetch so there must be some mechanism to implement multiple cache and/or memory requests. The model uses a non-blocking cache structure to facilitate multiple instruction requests [10][33]. A non-blocking cache allows execution to continue during the handling of a cache miss. This is most useful in the data cache of a processor employing out-of-order execution. When a data reference misses the cache, other independent instructions can be executed while the miss is being handled. A non-blocking instruction cache is not normally necessary. Since missed instruction dependency information is unknown, instructions cannot be issued while an instruction cache miss is being handled. However, when using an instruction prefetching algorithm, instruction fetch must continue while a prefetch miss is being handled. Non-blocking cache structures are also useful as storage areas for multiple memory requests which can occur in the same cycle.

In the architectural model, a structure similar to Kroft's MHSR buffer is situated between the instruction cache and memory, see Figure 4.4. It is called the outstanding memory request buffer or OMRB and contains information about all outstanding fetch and prefetch requests. An entry for a memory request is queued in the OMRB until the request is initiated and the full cache line is received from memory. An instruction fetch or prefetch memory request is serviced by the following steps:

- The memory request is sent to the OMRB. The requested address is compared with the addresses in the other entries in the OMRB. If a previous request does not already cover the new request, a new entry is added in the OMRB. This requires a fully associative search of all entries. However, the number of OMRB entries is limited to four so the parallel search is not too expensive.
- When the bus is available, the request of highest priority is selected from the set of non-issued requests in the OMRB. Instruction fetch has the highest priority, then

next-line prefetching, and finally target prefetching. A read request signal along with the address of the selected request is sent to memory on the bus.

- No more requests are transmitted on the bus from the OMRB during the next $w$ wait cycles. Memory requests cannot overlap since the memory can set up only one address at a time. During this time, however, more requests can be added into the OMRB.
- When the setup time is complete, the first segment of the requested cache line is received on the bus. Line wraparound is implemented so that the first segment always contains the requested word. The line is sent to memory and forwarded directly to the fetch unit.
- The remainder of the cache line segments are sent in subsequent cycles. The entry is then marked invalid, the bus is free, and other requests can be serviced.

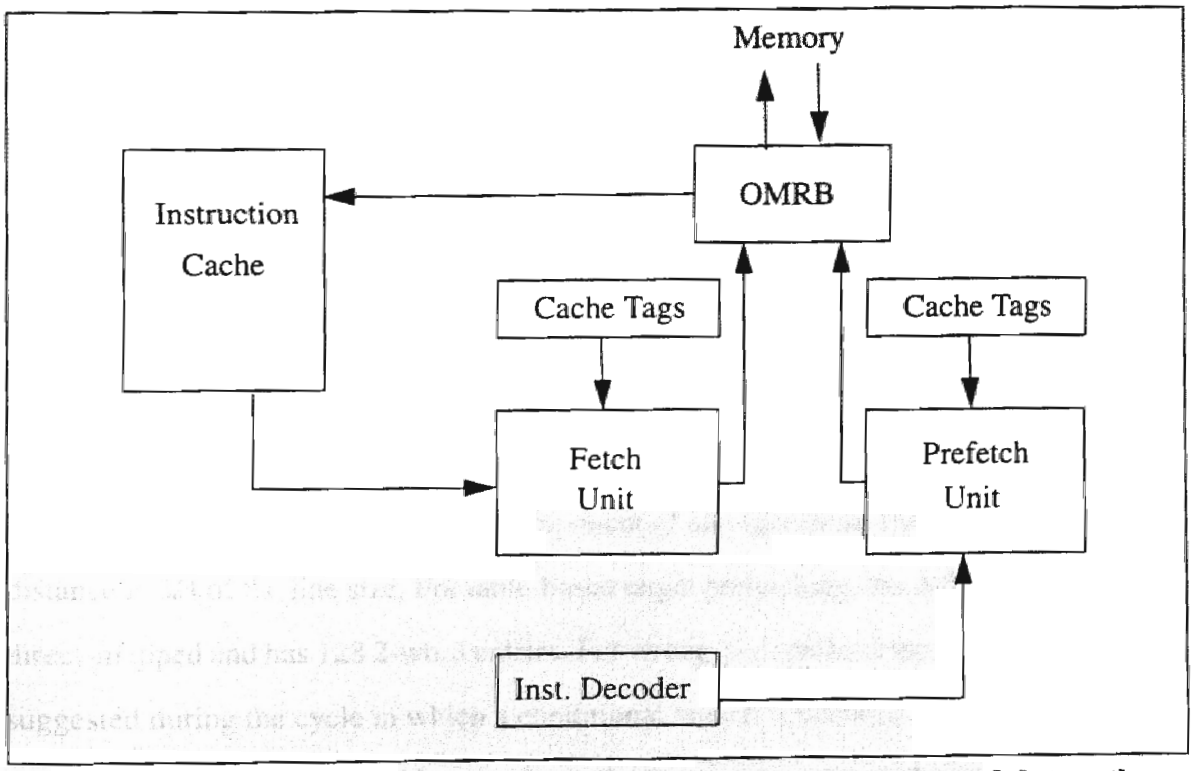


**Figure 4.4 Hardware Model - This figure shows the connection and datapaths between the cache, fetch and prefetch units, the outstanding memory request buffer structure (OMRB), and memory.**

Figure 4.6 shows a cycle time diagram of a memory request in a memory system requiring 2 bus transfers to fill a cache line and 4 wait cycles for memory address setup. If the request gains immediate access to the bus, the requested word will arrive in 6 cycles and the request will be completed in 7 cycles.

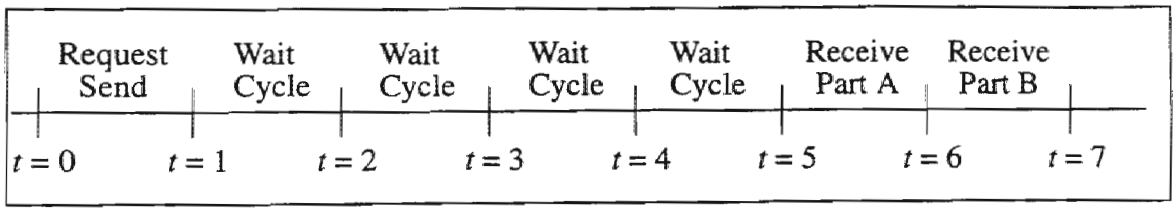


**Figure 4.5 Time Diagram of Memory Request Example - The memory system requires two cycles to fill a cache line and 4 wait cycles for memory address setup.**

#### 4.4.2.3 Prefetch Unit

The prefetch unit suggests when a line should be prefetched, computes the line's address, and performs a cache tag lookup to see if the line is resident in the cache. If the line is not resident, the prefetch is initiated and a memory request is sent to the OMRB. The cache tag structure can be accessed simultaneously by both the fetch and prefetch units. This requires that the tag structure be either dual-ported or replicated. Only one prefetch cache tag lookup can occur per cycle. If both a next-line and target-line prefetch is suggested in one cycle, the target-line address takes precedence. The next-line prefetch will be suggested during the next cycle if necessary.

For the next-line prefetching component of all algorithms, the default fetchahead distance is 3/4 of the line size. For table-based target prefetching, the default target table is direct-mapped and has 128 2-word entries. For wrong-path prefetching, a target prefetch is suggested during the cycle in which a conditional branch is decoded. The instruction decoder forwards the target address to the prefetch unit where the cache tag lookup occurs.

To summarize the changes necessary for prefetching, the following hardware additions are required:

- OMRB structure for a non-blocking instruction cache.

- Cache tag replication (or two tag ports) and tag comparison logic.
- Next-line prefetching requires logic for fetchahead distance calculation and next address calculation.
- Target-line prefetching requires next-line hardware, target table, logic to compare and update table entries, and data paths between table and cache tags.
- Wrong-path prefetching requires next-line hardware and a data path between instruction decoder and prefetch unit.

### 4.4.3 Benchmarks, Traces, and Tools

The simulator described in Section 4.4.1 can accept three different types of traces. The first two types are application traces from the SPEC benchmark suite generated on two different hardware platforms[60]. The traces are gathered using IDtrace on an Intel i486 SysVR4 Unix system and pixie on a DECstation 5000 with a MIPS R3000 processor. The benchmarks are listed in Table 4.6. Unfortunately, most of the SPEC benchmarks do

| Benchmark | Description |
|---|---|
| cc1 | Main component of gcc |
| compress | Unix compression utility |
| espresso | Boolean function minimization |
| eqntott | Boolean equation to truth table translation |
| sc | Spreadsheet program |
| xlisp | Lisp interpreter running 8-queens problem |

**Table 4.6 SPEC Benchmark Description**

not sufficiently exercise the instruction cache so that meaningful conclusions can be obtained from the simulation results. Table 4.7 shows the few number of misses and negligible traffic generated by the C SPEC benchmarks for even a rather small instruction cache. It is highly likely that the floating point benchmarks fare no better. Only cc1 causes

significant activity. If the benchmarks were to be used in cache studies, it is unclear how the results of each benchmark should be combined into one overall number. If the results are summed, the value from cc1 would dominate rendering the other results meaningless. If they were averaged or normalized with respect to cycle counts, then the findings observed in the cc1 results would be diluted by the other weaker results. My solution is to use cc1 alone as a representative application benchmark and show that the cache behavior observed using cc1 traces is indicative of the actual behavior of the instruction cache.

| Benchmark | i486 Misses | i486 Traffic | R3000 Misses | R3000 Traffic |
|---|---|---|---|---|
| cc1 | 46 | 12 | 72 | 15 |
| sc | 10 | 2 | 13 | 3 |
| xlisp | 15 | 2 | 11 | 3 |
| espresso | 4 | 1 | 5 | 1 |
| eqntott | 0 | 0 | 0 | 0 |
| compress | 0 | 0 | 0 | 0 |

**Table 4.7 Instruction Cache Activity Caused by SPEC Benchmarks - The miss columns give the number of misses per thousand instructions. The traffic columns give the percentage of time the bus is active with instruction information. The cache is 8Kbytes, direct-mapped, with line size of 16 bytes and line refill rate of 8 bytes/ cycle.**

To arrive at the actual behavior, the simulator also accepts a third type of trace, system-level traces. These traces were obtained by Nagle and Uhlig using a hardware monitoring setup called Monster [45]. The platform is a MIPS R3000-based DECstation 5000/200 running a Mach 3.0 operating system kernel. The traces contain physical instruction and data addresses gathered directly from the system bus while executing between 40 and 130 million kernel and user mode instructions. The benchmark programs are listed in Table 4.8 and sample cache activity is shown in Table 4.9. Note the increased level of activity produced by the system traces compared with the SPEC benchmark

numbers in Table 4.7. The cc1 application is the only one which comes close to simulating real cache activity.

Since the addresses are physical addresses, the cache modeled using the system traces is a physically-mapped cache[1]. For the experiments based on the application traces, the cache is virtually-mapped.

| System Trace | Description |
|---|---|
| gcc | Gnu C compiler |
| gs | Postscript file viewer |
| mpeg-play | Compressed video displayer |
| sdet | A multiprocess, system performance benchmark. From the SPEC SDM benchmark suite. |
| verilog | HDL hardware development tool |

**Table 4.8 System Trace Benchmark Description**

### 4.4.4 Performance Measurement

Most cache studies use the number of cache misses or miss ratio as the performance measure. However, miss reduction is not a sufficient measure of the performance of prefetched caches because it does not account for changes in miss latencies produced by prefetching. For instance, suppose the full miss latency for a processor is five cycles and one algorithm initiates a prefetch three cycles before the fetch while the other initiates a prefetch only one cycle before the fetch. An instruction miss will occur in both cases but the net result is not the same. The first prefetch algorithm will have reduced the miss penalty by two more cycles, thus giving better performance. Two

1. The traces will also allow modeling a physically-tagged, virtually-mapped cache if the cache size divided by the associativity is less than Mach's 4K page size. In such a cache, the set index part of the virtual address remains unchanged after virtual-to-physical address translation. The translation can be performed in parallel with cache set lookup and physical address can then be used for tag comparison.

statistics are used to compare the algorithms: total CPU cycles and miss cycles. The latter is the number of cycles wasted due to instruction misses, i.e.,

Total CPU cycles = Instructions executed + Miss cycles.

The measures of memory bus traffic are the total bus cycles and the bus utilization. The former is the number of cycles during which the bus is busy sending or receiving memory requests. The bus is deemed not busy during wait cycles. The bus utilization is the percentage of total CPU cycles during which the bus is busy.

In the simulations using the system traces, the results are based upon the sum of the cycle counts for all five system trace benchmarks. For the application-based simulations, the results are based upon the cycle counts from the cc1 benchmark run to completion.

| System Trace | Instructions | Misses | Traffic |
|---|---|---|---|
| gcc | 126M | 111 | 16 |
| gs | 86M | 110 | 16 |
| mpeg-play | 111M | 94 | 15 |
| sdet | 43M | 139 | 16 |
| verilog | 52M | 115 | 17 |

**Table 4.9 Instruction Cache Activity Generated by System Traces - The miss column is the number of misses per thousand instructions. The traffic column is the percentage of time the bus is active with instruction information. The cache is 8Kbytes, direct-mapped, with line size of 16 bytes and line refill rate of 8 bytes/cycle.**

## 4.5 Comparison Experiments

This section discusses the major results of a comprehensive empirical study of prefetching algorithms using the trace driven simulator. The first subsections compare algorithm performance for different cache sizes, line sizes, refill rates, and cache associativity. Both the CPU cycle time and the bus activity are observed. The subsections

also show that the application trace, cc1, and the combined system traces gives similar results.

The next two subsections show the effects of varying prefetch algorithm parameters. The algorithm comparison studies reveal that the performance gain of table-based target prefetching is not worth its additional hardware cost. Section 4.5.5 discusses attempts to extract higher performance by varying the target table parameters. Section 4.5.4 and Section 4.5.5 describe the reasons behind the hybrid algorithm's ineffectiveness and explain why these results differ from those found in Hsu [26]. The following subsection examines the effect of the fetchahead distance on the next-line and wrong-path algorithm performance.

The final two subsections attempt to forecast the effect of prefetching on future microprocessors. One certain feature of next-generation processors will be higher clock speeds. Even with an L2 cache, the shorter CPU cycle time will probably translate into a greater number of memory wait cycles. How will prefetching algorithms respond to this increased latency? The direction processor evolution also tends toward an increasing instruction issue rate and it is likely that multiple memory ports will be necessary to achieve them. The final subsections show how the prefetching algorithms help the memory system cope with the increased demands caused by these architectural changes.

### 4.5.1 Cache Size

The left graph in Figure 4.6 shows the number of CPU cycles needed to execute the system traces with no prefetching and with different prefetch algorithms. The right graph shows the percentage reduction in the execution cycles due to the prefetching algorithm. Wrong-path prefetching performed the best in terms of CPU cycles for all cache sizes, producing up to a 16% speedup over that of no prefetching. It performs up to 4% better than the other algorithms.The hybrid algorithm performed only slightly better

than next-line prefetching. It seems to extract little performance gain from its additional hardware.

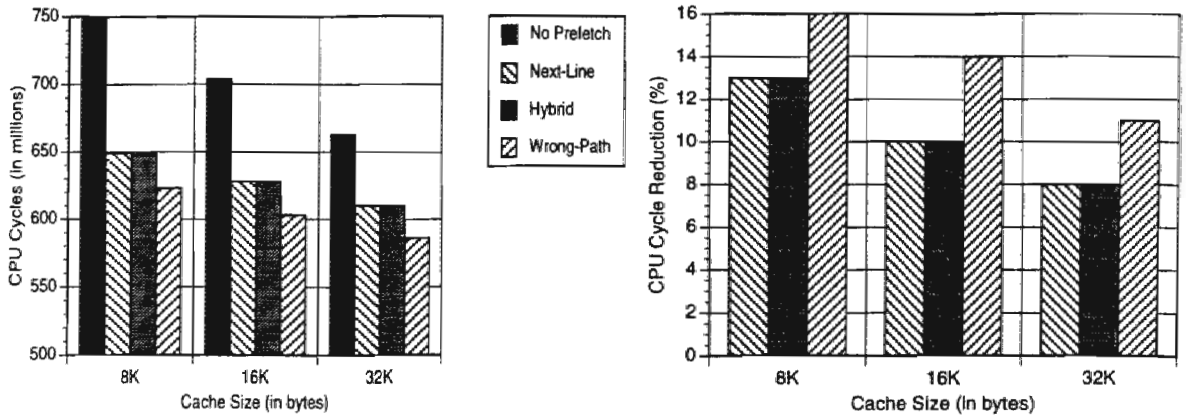


**Figure 4.6 Cycle Time for Different Cache Sizes - The left graph shows total CPU cycles and the right graph shows the cycle improvement gained by prefetching. System traces are used for simulation input.**

It is interesting to note in the left graph that a wrong-path prefetched 8K cache gives better performance than a non-prefetched 32K cache and a next-line or hybrid prefetched 16K cache.

Since prefetching can only reduce the miss latency cycles, which in the above caches comprise 20-40% of the total execution cycles, the prefetching algorithms must be removing a higher percentage of miss cycles than total CPU cycles. Figure 4.7 shows that the miss cycle reduction is large. Wrong-path prefetching reduces miss cycles by almost 40% in an 8K cache.

The major disadvantage of prefetching is the additional memory traffic it generates as can be seen in Figure 4.8. Two factors combine to cause this increased bus utilization. The first is that prefetching generates unnecessary references which increases the bus

traffic. The second is that prefetching reduces the total CPU cycle count. More traffic in a shorter time period translates into higher bus utilization.

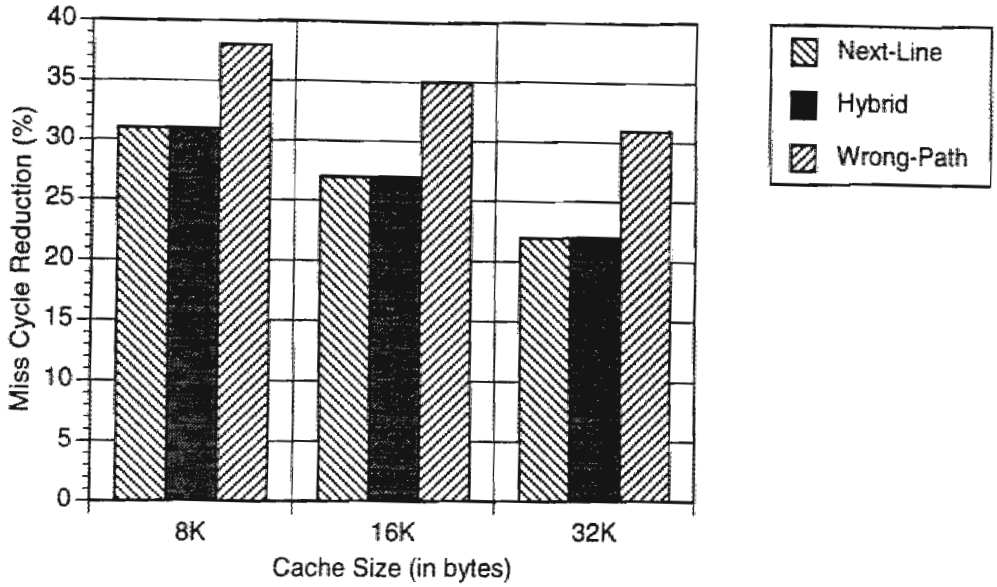


**Figure 4.7 Miss Cycle Reduction for Different Cache Sizes - System traces are used for simulation input.**

Bus bandwidth must be viewed as a resource just as a functional unit or memory port. For instance, if the bus bandwidth is allowed to grow by doubling the refill rate, implementing a prefetching scheme can significantly reduce the required cache size without greatly increasing the bus traffic. Figure 4.9 shows the effect of this resource allocation. Cache A,B, and C are non-prefetched 32K caches with a 32 byte line size and 8, 16, and 32 byte/cycle refill rates respectively. Cache D is a wrong-path prefetched 8K cache with a 32 byte line and 32 byte/cycle refill. Cache D has the best CPU cycle performance and its bus traffic is only slightly higher than that of Cache A. Caches B and

C are included in the graph to verify that the performance improvement is a result of prefetching and not of the increased refill rate.

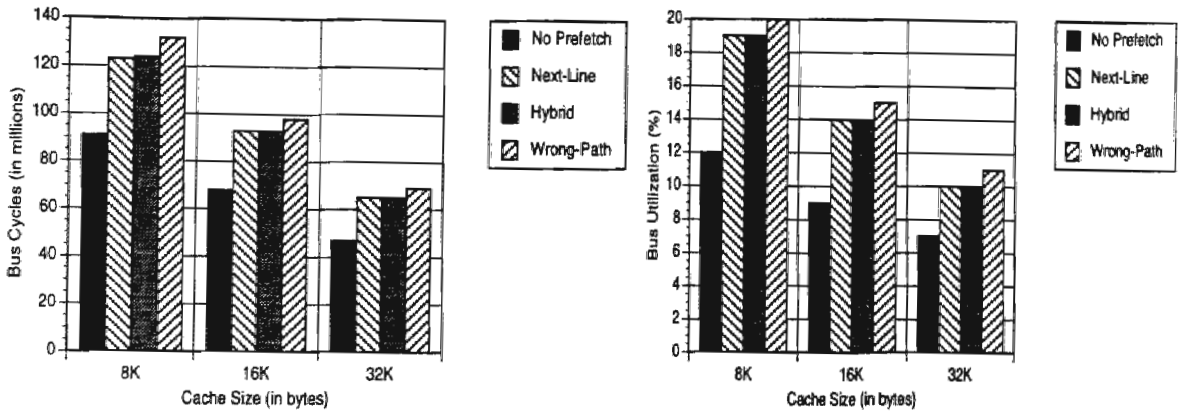


**Figure 4.8 Bus Traffic For Different Cache Sizes - The left graph is the total bus cycles and the right graph is the bus utilization. System traces are used for simulation input.**

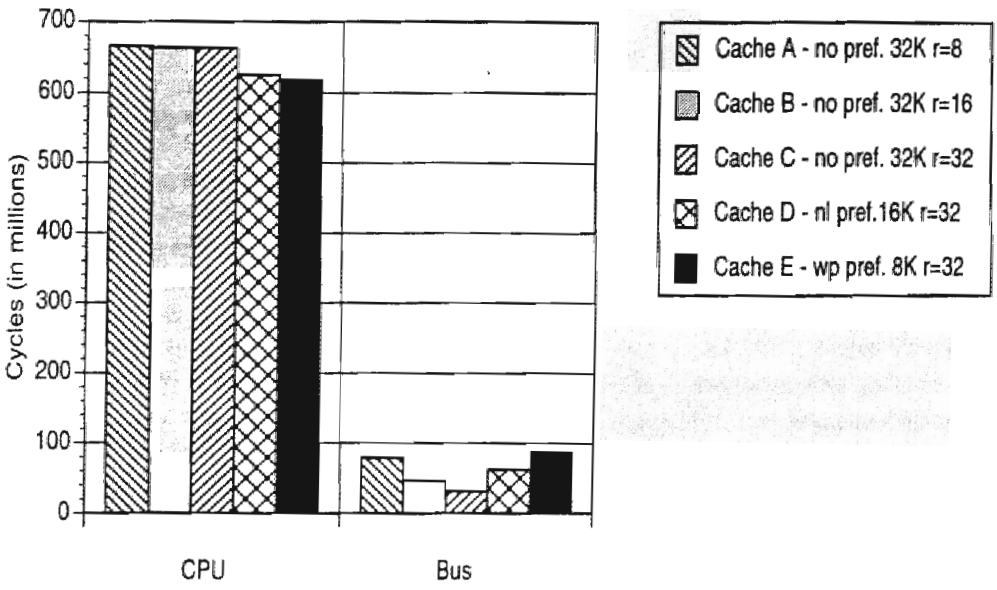


**Figure 4.9 Prefetching Allows Better Performance with Smaller Cache - Caches A,B, and C are non-prefetched 32K caches with 8, 16, and 32 byte/cycle refill rates. Cache D is a next-line prefetched 16K cache with 32 byte/cycle refill rate. Cache E is a wrong-path prefetched 8K cache with 32 byte/cycle refill rate.**

The results from the cc1 application study are similar and also show that the wrong-path prefetching algorithm is superior, see Figure 4.10. It reduces CPU cycles by up to 14% while increasing traffic by 18% for an 8K cache and around 5% for a 32K cache. These results are comparable with those found using the system traces which indicates that cc1 is a reasonably good benchmark for application-based cache simulation. It is important to note, however, that prefetching removes many more instruction miss cycles in cc1 than in the system traces (62% vs. 32% for 32K cache), yet the CPU cycle reduction is not as impressive in the cc1 experiments (5% vs. 11% for 32K cache). This is because a smaller portion of the total execution cycles using cc1 trace are miss cycles. The smaller reduction of a larger number of miss cycles in the system trace had a larger overall effect than a larger reduction of a smaller number of miss cycles in the cc1 trace.

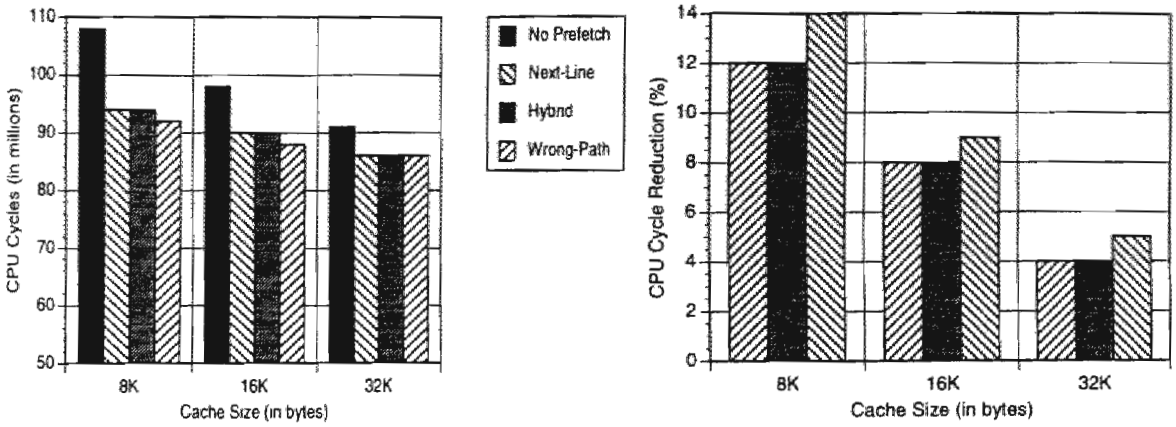


**Figure 4.10 Cycle Times For Different Cache Sizes - The left graph shows total CPU cycles and the right graph shows the cycle improvement gained by prefetching. The cc1 application trace was used for simulation input.**

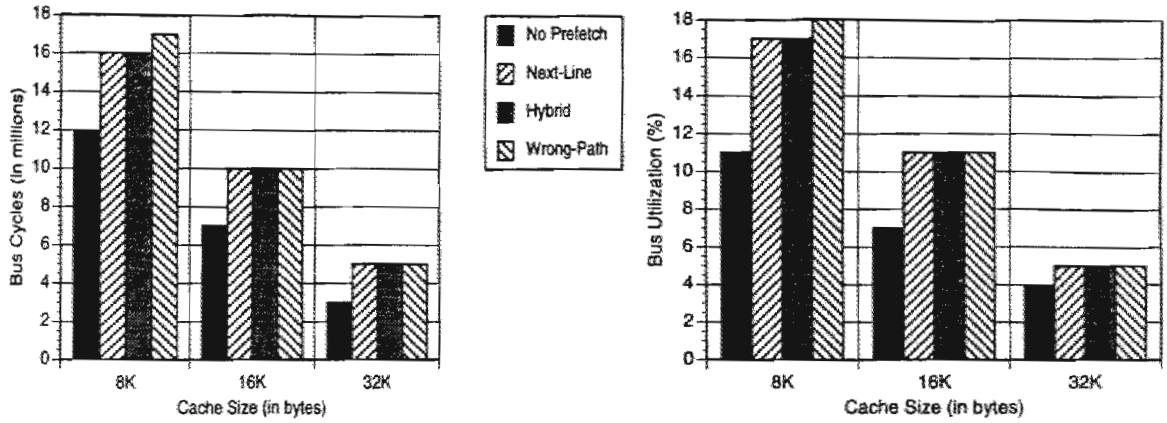


**Figure 4.11 Bus Traffic For Different Cache Sizes - The left graph is the total bus cycles and the right graph is the bus utilization.The cc1 application trace was used for simulation input.**

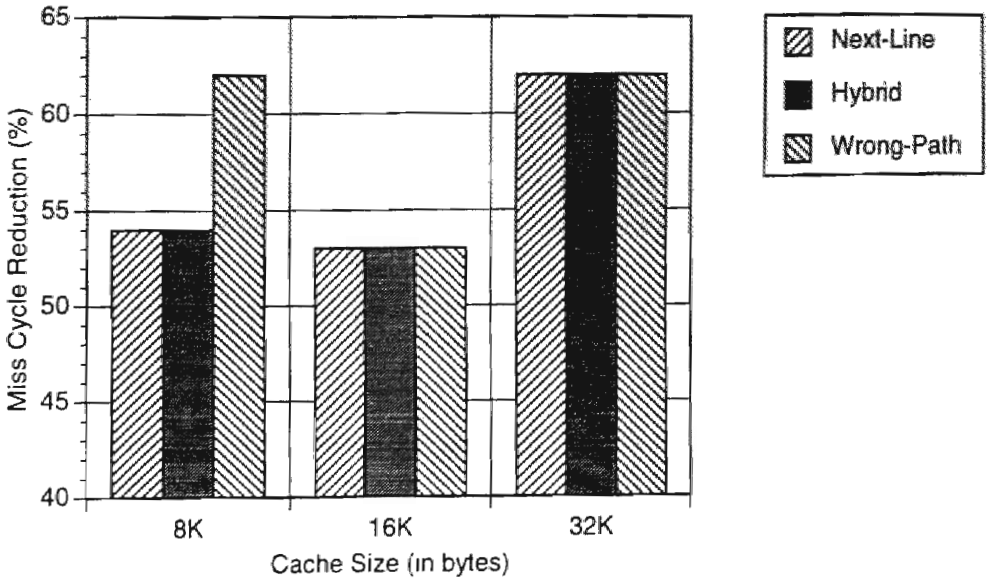


**Figure 4.12 Miss Cycle Reduction For Different Cache Sizes - The cc1 application trace was used for simulation input.**

## 4.5.2 Refill Size

The cache line refill rate plays an important role in the effectiveness of prefetching. A small refill rate means that many bus cycles are required to transfer a line from memory

to the cache. This exacerbates two detrimental effects. First, prefetching inevitably causes unnecessary lines to be transferred to the cache. A small refill rate puts a high cost, in terms of bus cycles, on these wasted transfers. Secondly, longer miss handling increases the fetch memory request delay. When a fetch miss occurs during the service of a prefetch request, the memory request cannot be sent until the prefetch is completed. Increased line segment transfer cycles, caused by the smaller refill rate, increase the fetch delay and thus the execution time. Figure 4.13 and Figure 4.14 show the effect of refill rate on prefetch performance. As expected, the higher the refill rate, the better the performance.

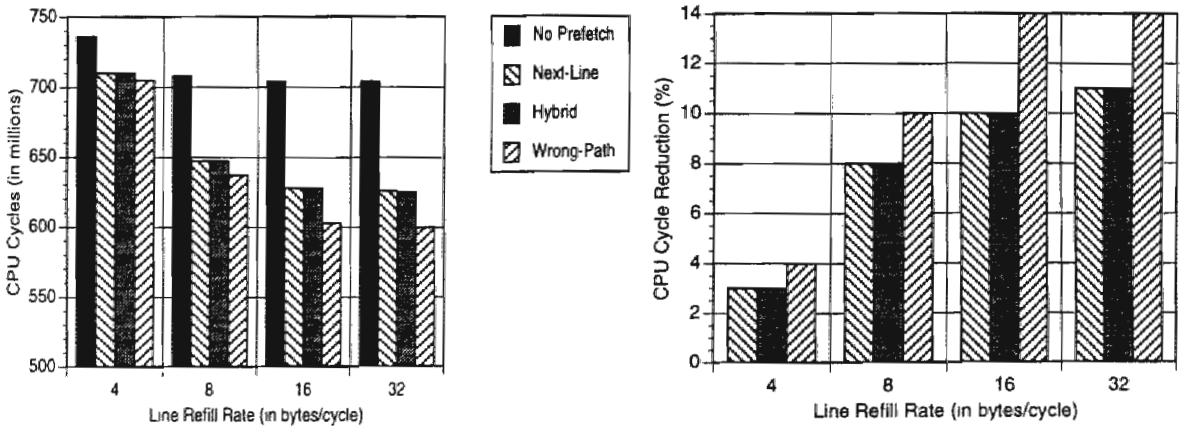


**Figure 4.13 Cycle Times For Different Line Refill Rates - The left graph shows total CPU cycles and the right graph shows the cycle improvement gained by prefetching. System traces were used for simulation input.**

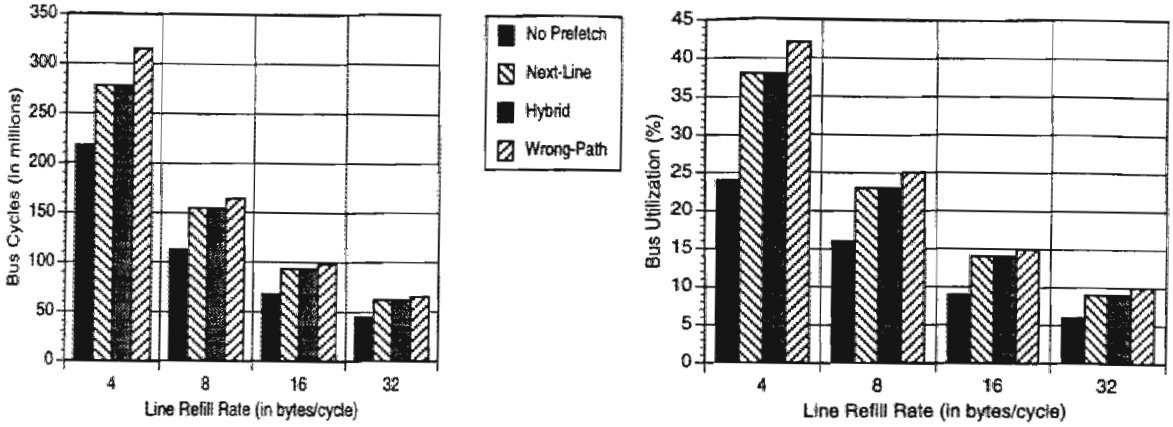


**Figure 4.14 Bus Traffic For Different Line Refill Rates - The left graph is the total bus cycles and the right graph is the bus utilization. System traces were used for simulation input.**

### 4.5.3 Cache Associativity

Figure 4.15 shows the effect of cache associativity on prefetch performance. An anticipated problem of prefetching, especially wrong-path prefetching, is its potential to pollute the cache with non-accessed lines. Cache configurations which can absorb this pollution will enhance prefetching effectiveness. This is verified in the associativity figures. Caches with higher associativity will have fewer conflict misses and so fewer needed lines will be displaced by prefetched lines. However, there is an opposing effect as associativity increases.

Figure 4.16 shows that the miss cycle reduction actually declines as associativity increases. Prefetching algorithms excel at bringing lines back into the cache which were previously displaced. Much of their performance benefit is derived from this behavior. High associativity takes a chunk out of the prefetch gain by reducing the number of line displacements in the cache. One might then suggest that higher associativity be used in place of prefetching. The results show, however, that prefetching works in conjunction with increased associativity. Furthermore, prefetching is unlikely to affect the cache hit

access times, whereas highly associative caches are likely to increase the CPU cycle time or require pipelined cache accesses which would negate its perceived advantages [23].

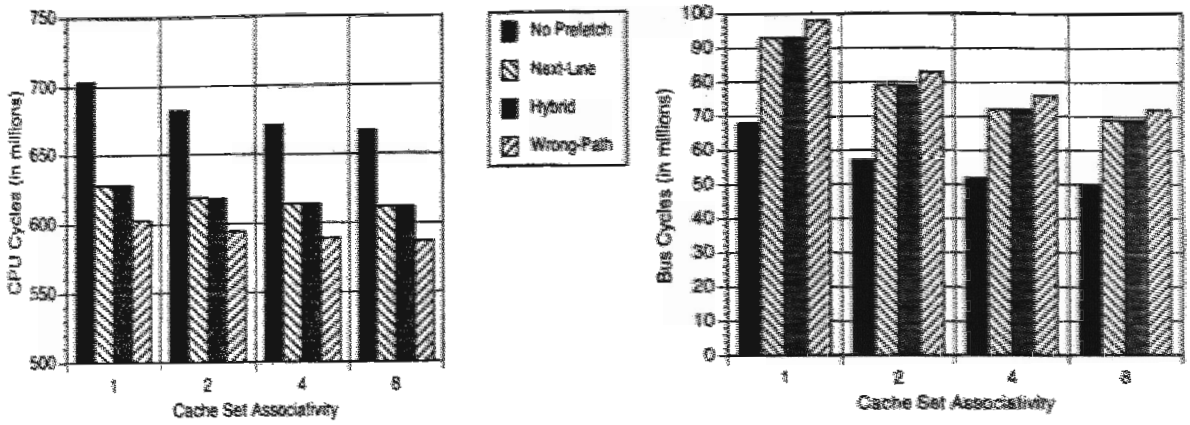


**Figure 4.15 CPU and Bus Cycles for Different Cache Associativities - The system traces were used for simulation input.**

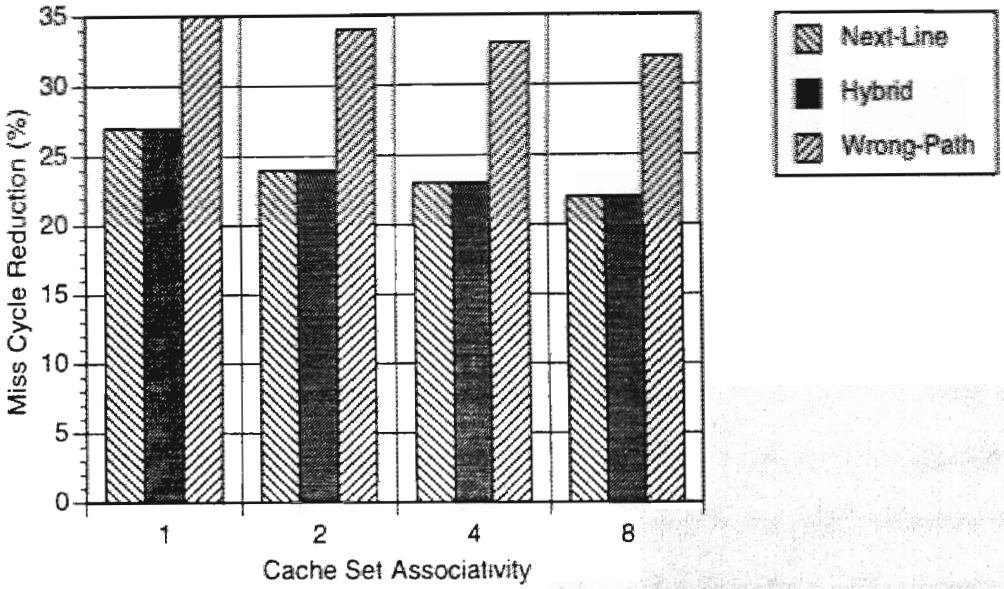


**Figure 4.16 Miss Cycle Reduction for Different Associativities - The system traces were used for simulation input.**

### 4.5.4 Prefetch Gains and Accuracy

To summarize the results thus far, prefetching is effective in reducing CPU cycle time in all cache configurations. The cost is an increase in bus traffic. Wrong-path

prefetching performed slightly better than next-line prefetching in all cases, sometimes by as much as 4%. The hardware cost difference between the two algorithms is small. On the other hand, the hybrid scheme shows little improvement over next-line prefetching. The negligible gain over next-line prefetching is certainly not worth the additional hardware expense. This was not the result found by Hsu and Smith in their comparison study. The performance gain from the hybrid algorithm approximated the sum of the gains individually achieved by next-line and table-based target prefetching.

Besides the fact that their study was based upon a different architecture platform and that they used different benchmarks, the major difference is the difference in the modeled cache sizes. They studied supercomputer cache behavior with cache sizes ranging from 128 to 2K bytes in size. To be effective, table-based target prefetching requires a high line turnover rate in the cache. Large caches and high associativity limit the turnover rate and subsequently limit the effectiveness of table-based prefetching.

To determine the usefulness of the different components in the prefetching algorithms, special memory reference tags were added to the simulator to distinguish the types and results of prefetch initiations. Figure 4.17 gives a breakdown of where the prefetches are coming from in the different algorithms. A *prefetch gain* is defined to be a prefetch which removed an instruction cache miss, i.e, had the prefetch not occurred, a subsequent instruction fetch would have missed the cache. The next-line component dominates the target-line component in both the hybrid and wrong-path schemes but the difference is especially severe in the hybrid scheme. Table-based prefetching generates few prefetch gains and has a negligible effect on miss reduction. This is due to a combination of two factors:

- Compulsory misses are not eliminated since a previous execution is required to add the necessary link in the target table.

- After the first miss and the table is updated, the line will be cache resident. In order for the line to be prefetched, the line will first have to be displaced by the cache. This is less likely to occur in a larger cache.

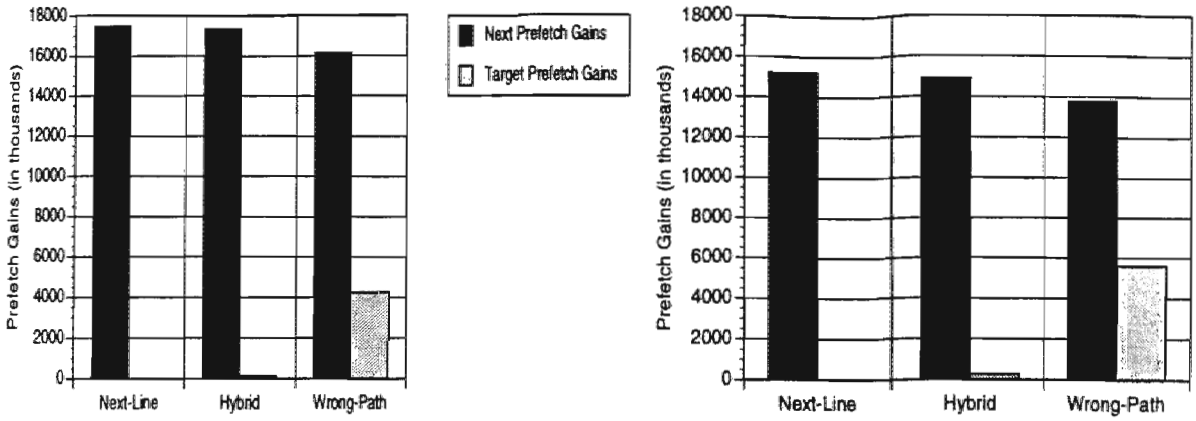


**Figure 4.17 Prefetch Gains for Different Fetchahead Values - The algorithms in graph on the left uses a fetchahead distance of 24 bytes. In the right graph, the fetchahead distance is 8. The system traces were used for simulation input.**

Examining the hybrid scheme's behavior in a highly associative cache will verify its reliance on conflict misses for performance. The left graph of Figure 4.18 compares the number of prefetches generated in a direct and 8-way associative cache using the hybrid algorithm. In the associative cache where conflict misses are minimal, few prefetches are even generated by the algorithm. This highlights the hybrid scheme's inability to reduce the number of compulsory misses. On the other hand, the right graph shows the more

balanced target prefetch generation of wrong-path prediction. Prefetching not-taken paths does eliminate compulsory misses.

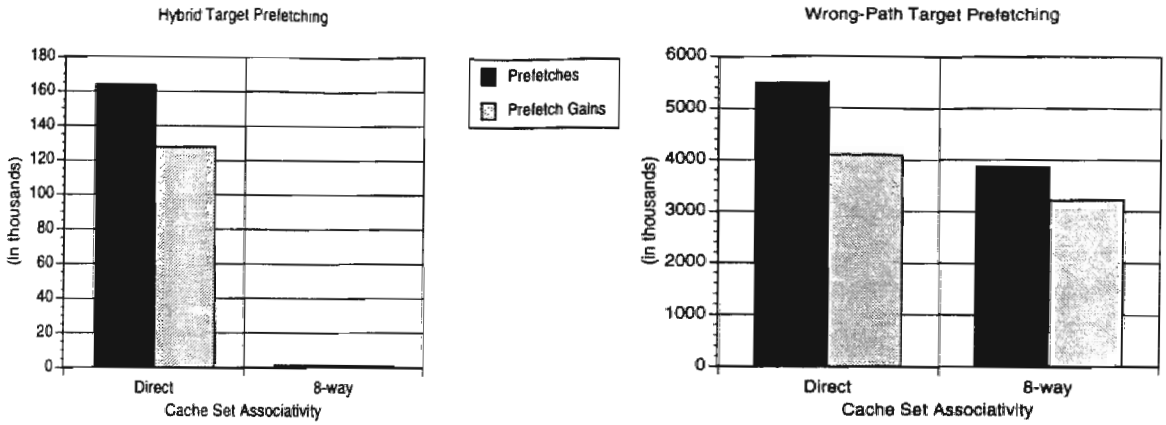


**Figure 4.18 Target Prefetch Compare - The left graph shows the target prefetches made by the hybrid algorithm. The right graph shows those made by the wrong-path algorithm. The cache is 16K bytes with line size of 32 byte lines and refill rate of 16 bytes/cycle. The cc1 application trace was used for simulation input.**

It is interesting to note in the right graph in Figure 4.18 that a majority of the prefetches initiated by the wrong path algorithm are, in fact, prefetch gains. Figure 4.19 shows this more clearly by displaying the prefetch accuracy of the different wrong-path components. Target line prefetching is surprisingly over 75% accurate. This means that over 75% of the time, the line containing the target of a not-taken branch is referenced before it is displaced. Next-line prefetch accuracy declines as the fetchahead distance increases. As the fetchahead distance increases, there is a higher probability that a branch

will be taken before execution reaches the end of the current cache line and the consecutive cache line will not be referenced.

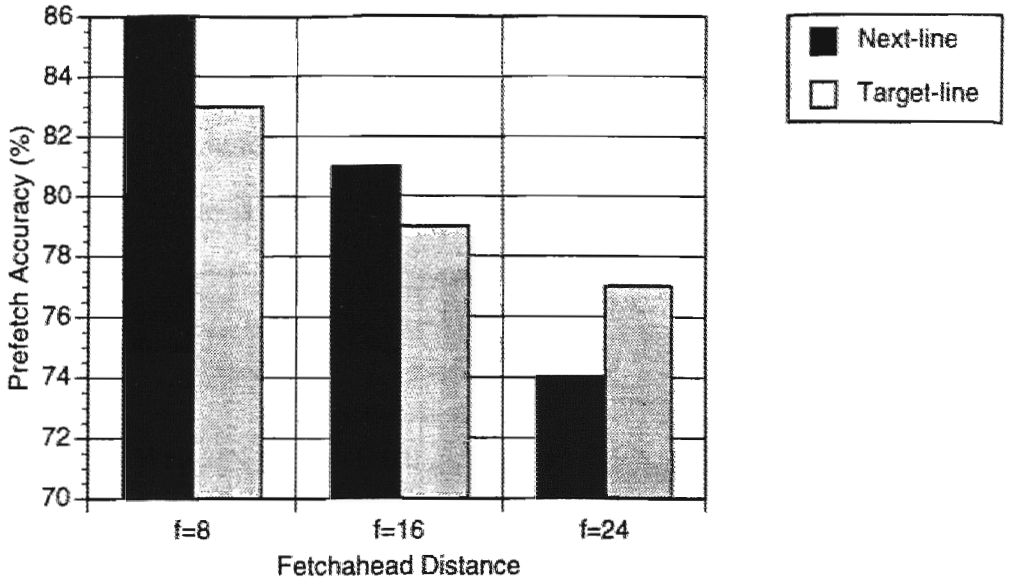


**Figure 4.19 Wrong-Path Prefetch Accuracy - The system traces were used for simulation input.**

### 4.5.5 Target Table Size

Changing the configuration of the target table slightly affects the number of target prefetch gains in the hybrid algorithm but has a negligible effect on the CPU counts. Figure 4.20 shows effects of changing both the number of target table entries and the associativity. In this and other cache configurations, no appreciable improvement occurred if the table size was bigger that 128 or 256 entries. It is interesting to note that in this example a fully associative table reduces its effectiveness. This might indicate that many different entries are added to the table causing high turnover in the fully associative table. Direct or low associativity protects some worthwhile entries from being overwritten by the many useless ones.

As stated earlier, the hybrid algorithm uses a 128 entry, direct-mapped table in all algorithm comparisons. While an associative table might have generated more prefetch

gains, Table 4.10 shows that it would have little effect on total CPU cycles. This is primarily due to the fact that the number of target-line prefetches is dwarfed by the number of next-line prefetches. The number of target-line prefetch gains is around 200-300 thousand. The number of next-line prefetch gains is around 11-12 million. Using a cache bigger that 1K bytes would increase the difference even further since target-line prefetching effectiveness relies on a cache with a high rate of replacement.

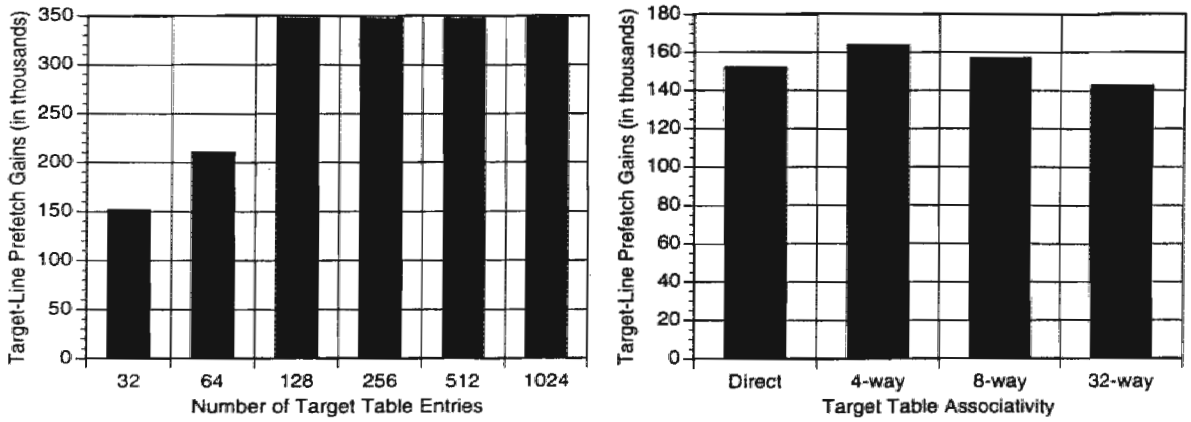


**Figure 4.20 Effect of Target Table Parameters - The left graph shows the effect of changing the number of entries in a direct-mapped table. The right graph shows the effect of changing the associativity in a 32 entry table. The caches are 1K, direct-mapped, line size 16, and refill rate 16 bytes/cycle.**

| Table Associativity | CPU cycles (in thousands) |
|---|---|
| 1 | 132393 |
| 4 | 132382 |
| 8 | 132392 |
| 32 | 132456 |

**Table 4.10 Target Table Associativity and CPU Cycles**

### 4.5.6 Effect of Fetchahead Distance

The prefetch distance parameter can be used to moderate the increase in bus traffic at the expense of miss cycle reduction. Figure 4.21 and Figure 4.22 show that by reducing

the fetchahead distance, bus utilization is reduced. However, at the same time the prefetch algorithm's performance drops substantially. The optimal fetchahead distance is either 1/2 or 3/4 of the line size.

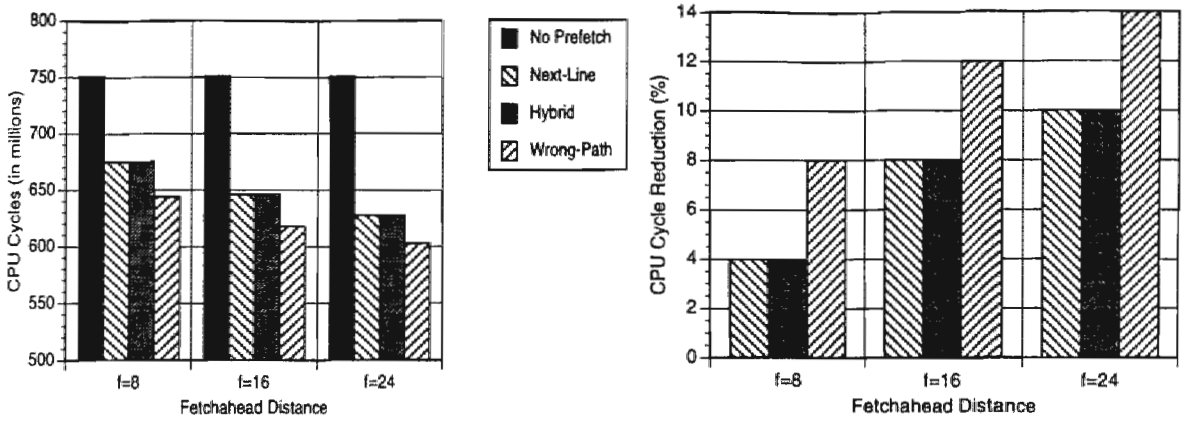


**Figure 4.21 CPU Cycles for Different Fetchahead Distances - The system traces were used for simulation input.**

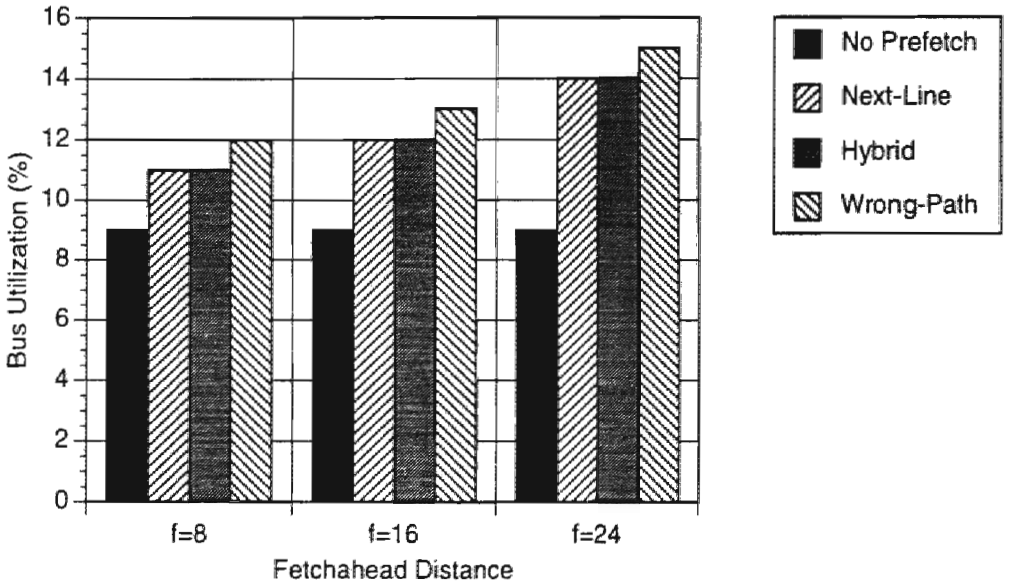


**Figure 4.22 Bus Utilization for Different Fetchahead Distances - The system traces were used for simulation input.**

### 4.5.7 Prefetch Distance

How do the prefetch algorithms scale to increased memory latency? As the memory latency increases, prefetching algorithms must prefetch farther in advance of the current execution point in order to cover the latency period. Therefore, the performance advantage of prefetching should diminish as the latency increases. However, of the three algorithms studied, the wrong-path algorithm might suffer the least from increased latency since its target prefetching component does not prefetch immediately used instructions. Figure 4.23 shows the results of varying the number of memory wait cycles. As expected, the CPU cycle time increases as the number of wait cycles is increased. Unexpectedly, the CPU cycle reduction caused by prefetching also increases as $w$ is increased. In other words, as memory latency increases, prefetching becomes more effective.

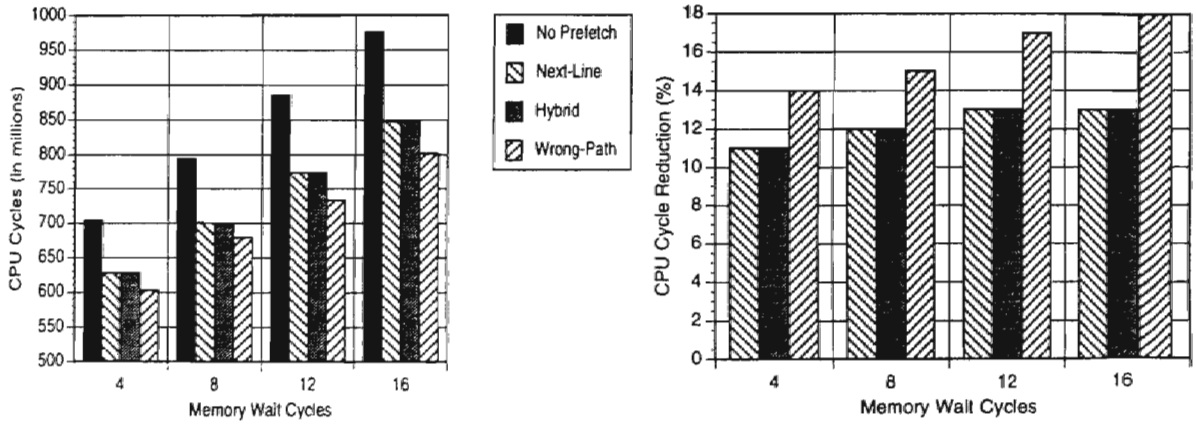


**Figure 4.23 CPU Cycles for Different Memory Latencies - The system traces were used for simulation input.**

The reason for this is that the reduction in the number of instruction miss cycles becomes more visible as the portion of total CPU cycles due to instruction misses increases. Higher latency causes more instruction miss cycles which, in turn, allows the prefetching algorithm to have a greater overall effect as shown in Figure 4.23. The initial thinking was not entirely incorrect though. Figure 4.24 does show that the effectiveness of prefetching diminishes as the memory latency increases.

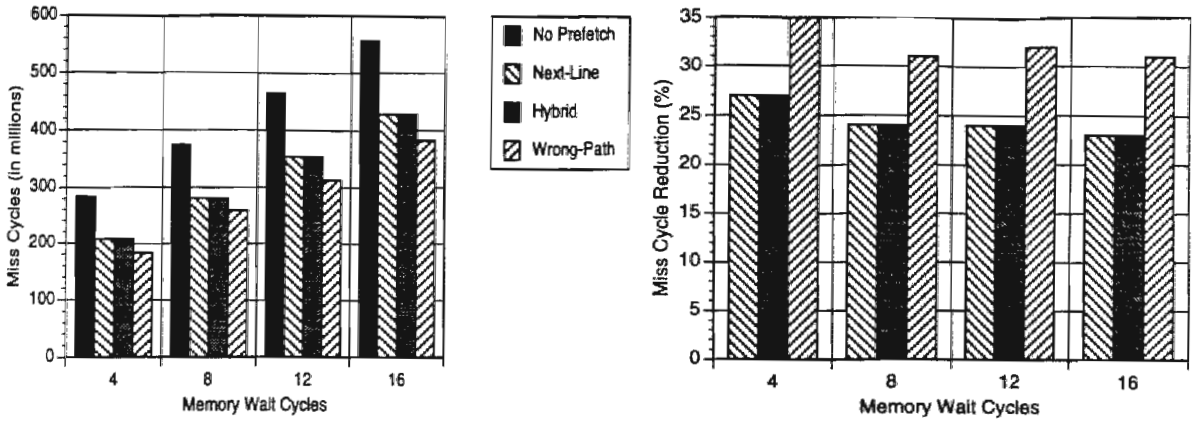


**Figure 4.24 Miss Cycles for Different Memory Latencies - The system traces used for simulation input**

The above figures perhaps show that wrong-path prefetching is more able to cope with a high memory latency than next-line prefetching but the difference is small. Since the majority of wrong-path prefetch gains come from the next-line component, the target prefetch effects are hard to see in the CPU cycle graphs. To investigate the effect of memory latency on each prefetching component, I define the *prefetch distance* to be the number of cycles between a prefetch initiation and the first instruction fetch request to the prefetched line. By having the simulator record these values for every next-line and target-line prefetch gain in the wrong-path algorithm, I can verify that target-line prefetches do indeed prefetch farther ahead of the current execution point. Figure 4.25 shows the results. For both a high and a low fetchahead value, the target-line prefetch gains have statistically higher prefetch distances.

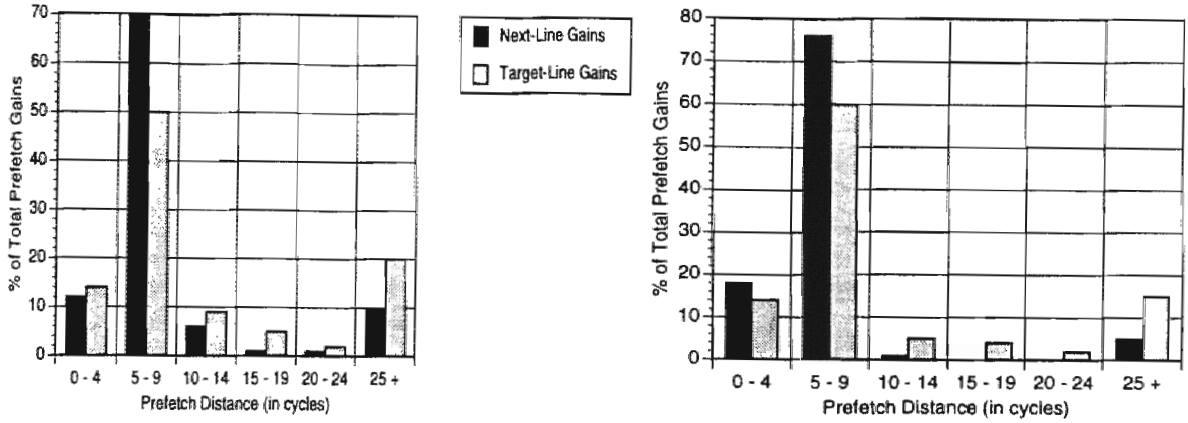


**Figure 4.25 Prefetch Distance Distribution - The graphs show the distribution of prefetch distances for all next-line and target-line prefetch gains within the wrong-path algorithm. The caches are 16K, direct-mapped, line size 32, and refill rate 16 bytes/cycle. In the left graph, the next-line fetchahead distance is 24 while on the right it is 8.**

## 4.6 Effect on Higher Performance Machines

Until now this study has been restricted to microprocessor architectures which issue only one instruction per cycle. The latest generation processors, however, can issue multiple instructions per cycle although the overall instructions per cycle (IPC) is still less than two. It is predicted that future microprocessors will be able to maintain an instruction per cycle (IPC) rate greater than two. It has been shown that prefetching allows increased performance on single issue architectures. To what extent does it also demonstrate performance gains in superscalar or VLIW machines with higher IPC? There are two opposing factors which are similar to those concerning the variation in the number of memory wait cycles. In multi-issue machines, since the execution rate is higher, the prefetch algorithm must prefetch farther in advance of the current execution point. This will degrade prefetch performance since a larger portion of the prefetches will not arrive in time. Conversely, increasing the issue rate reduces the time spent executing instructions and makes the effects of misses more pronounced. In multi-issue machines, a larger

fraction of the total execution time is due to miss latency cycles. Since prefetching removes a portion of the miss cycles, the effects of prefetching could be greater than in single-issue microprocessors.

Another element of design complexity which goes hand in hand with issue rate is the number of memory ports, i.e, the number of memory requests which can be processed concurrently. Without entering the heated debate concerning the maximum instruction-level parallelism that exists in general-purpose code [7][35][47][57][71], it is pretty obvious that multiple memory ports will be required to achieve an IPC of greater than two [59][71]. How will multiple memory ports effect the performance of prefetching? Right off the top, adding memory ports will expand the memory-to-cache bandwidth so the seriousness of prefetch-induced traffic should be lessened. In addition, multiple ports will reduce the backlog of waiting memory requests which will have two positive effects. The first is a reduction in fetch miss request delays caused by in progress prefetches. With only a single port, if a previous prefetch is receiving a cache line on the bus, a fetch miss memory request will be stalled until the line is transferred. Multiple ports will often allow the fetch request to be initiated immediately. Secondly, prefetches will be initiated earlier which will increase the prefetch distance.

The effects of increasing the instruction issue rate and adding additional memory ports to the cache are examined in this section. Multiple instruction issue is simulated by maintaining a fetch window which, at the beginning of each cycle, contains the maximum number of instructions which can be issued per cycle. Each instruction which does not cause a fetch miss is removed from the window. At the beginning of the next cycle, the fetch window is refilled from the trace. This is only an approximation to that of an actual processor since number of instructions retired per cycle also depends upon data dependencies, exceptions, and branch mispredictions. The model for multiple memory ports is an interleaved memory with the same number of banks as ports. As long as two memory requests do not address the same bank, they can be handled in parallel. Each port

has its own memory bus so request operations are truly independent if no bank conflicts occur. If a conflict does occur, one request waits in the OMRB until the port is available. The memory is partitioned into banks at the cache line level, i.e., one memory bank contains every word in the cache line. Consecutive cache lines are stored in consecutive banks.

The following graphs show the results of varying the issue rate and the number of ports when using the system traces as input to the simulator. Notice that in Figure 4.26 when the issue rate is doubled the CPU cycles does not decrease by a factor of two. This is because the miss latency cycles are not reduced. Figure 4.27 and Figure 4.28 show that the effectiveness of prefetching increases as the issue rate increases. Next-line prefetching improves performance by up to 16% and wrong-path prefetching by 21% in a machine with an issue rate of 4.

The graphs seem to show that the number of memory ports is rather unimportant but this is misleading. First, as mentioned previously, multiple memory ports are probably necessary to achieve an average IPC of 2 or 4. In addition, multiple ports are also required to handle the memory traffic. With superscalar execution, the same amount of memory traffic will occur in a shorter amount of time. Figure 4.29 shows the port utilization which is defined as the percentage of CPU cycles during which the port is busy sending or receiving information. For multiple port configurations, the highest utilization percentage is given but, since the requests are relatively randomly distributed over all the banks, there is little variation in the utilizations between the ports. It can be observed that even when no prefetching is used, additional ports are necessary to keep the traffic down to a reasonable level. If prefetching is implemented, multiple ports are a must.

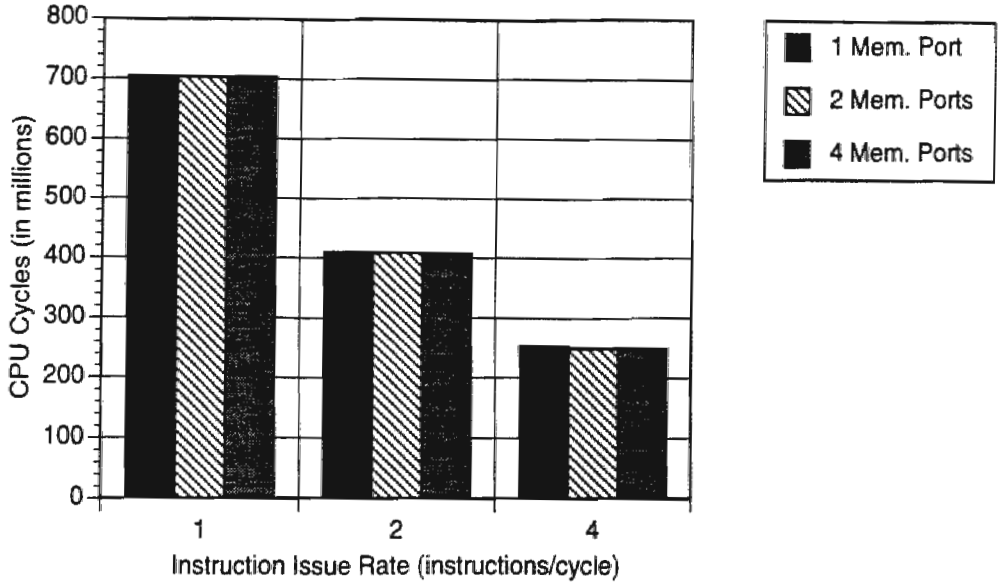


**Figure 4.26 Effect of Issue Rate and Memory Ports with No Prefetching - The graph shows the cycles required to execute the system traces for different issue rates and different numbers of memory ports.**

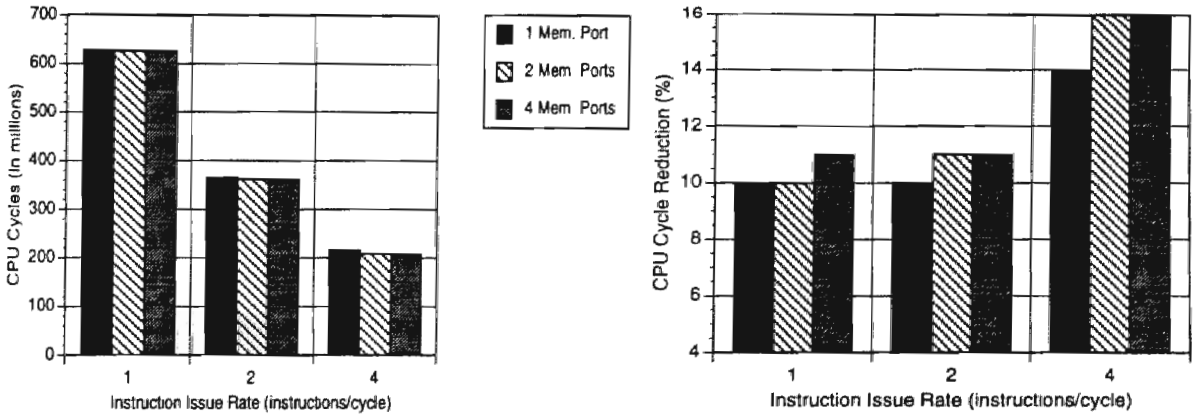


**Figure 4.27 Effect of Issue Rate and Memory Ports with Next-Line Prefetching - The left graph shows the CPU cycles required to execute the system traces. The right graph shows the percent reduction in CPU cycles over prefetching used.**

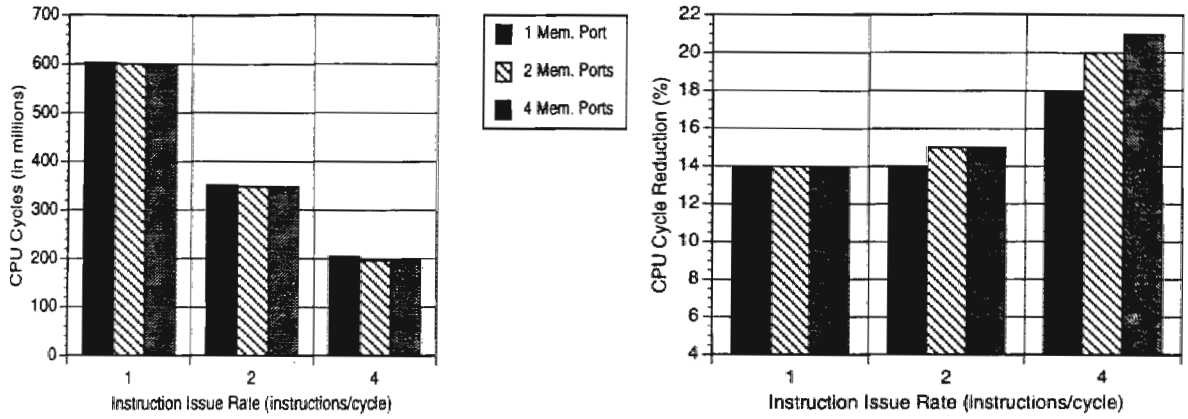


**Figure 4.28 Effect of Issue Rate and Memory Ports with Wrong-Path Prefetching - The left graph shows the CPU cycles required to execute the system traces. The right graph shows the percent reduction in CPU cycles from when no prefetching used.**

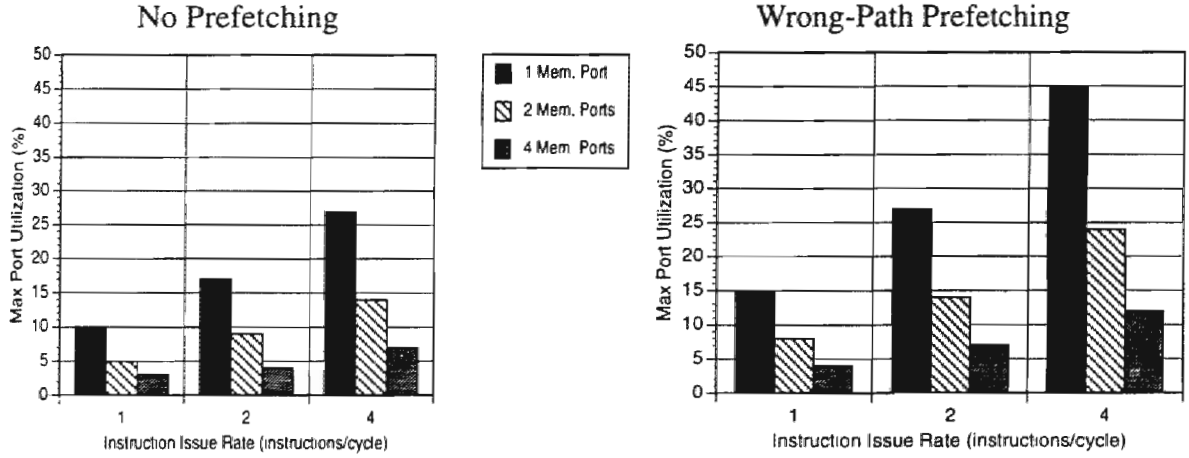


**Figure 4.29 Port Utilization - These graphs show the percentage of cycles the busiest memory port spends transferring information during the execution of the system traces. The left graph is with no prefetching, the right graph with wrong-path prefetching.**

## 4.7 Algorithm Extensions

This section outlines several attempts to improve the performance of wrong-path prefetching. The first modification was to prefetch multiple target lines. If requests are not

waiting for the bus when a target-line prefetch transfer is completed, the next consecutive line is also transferred. Since the address is already set up in memory, no additional wait cycles are required and the cost of the wait cycles needed in the initial prefetch request is amortized over two line transfers. The modification is rather unproductive as can be seen in Figure 4.30. The CPU cycle time remains essentially unchanged and the bus traffic is

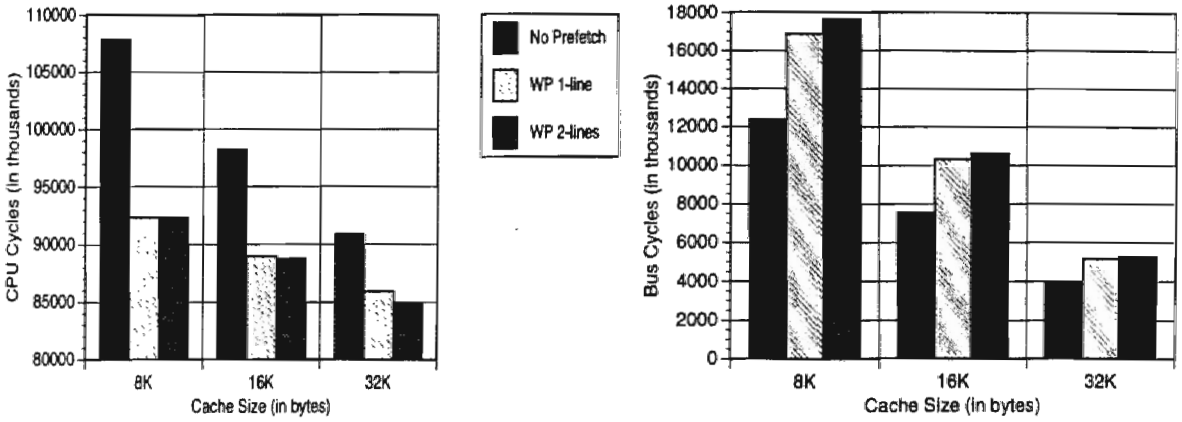


**Figure 4.30 Multiple Target Line Prefetch - The left graph shows the minimal reduction in CPU cycles when two target lines are prefetched. The right graph shows the increase in bus traffic. The cc1 trace was used as simulator input.**

increased.

Another attempt at performance improvement was to change the procedure for generating the target-line prefetch address. If the target of a branch is located at the end of a cache line, it might not be a good idea to prefetch that line since only the end will get executed. If instead, the next consecutive line were prefetched, traffic might be reduced. The *target threshold* is defined to be the point in the target line at which the next consecutive line is prefetched rather than the target line itself. Figure 4.31 shows the results for different target threshold values, $t$, using a line size of 32 bytes. If the difference between the target address lies within $t$ bytes of the beginning of the target line, the target line is prefetched. Otherwise, the next consecutive line following the target line is prefetched. The value t=32 represents regular wrong-path prefetching. Again, the results

were unimpressive. The difference in the CPU and bus cycles are in the noise margin of the experiments.

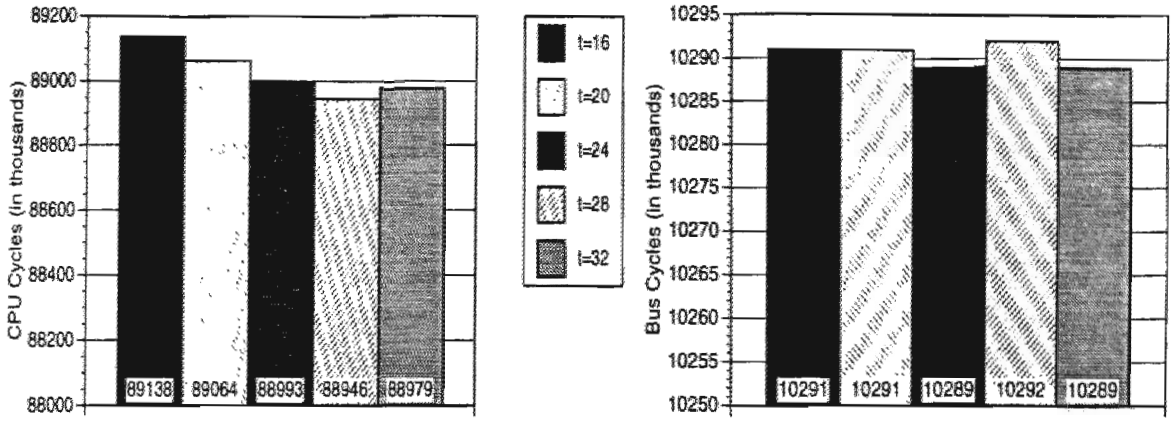


**Figure 4.31 Effect of Target Threshold Variation - The graphs show the CPU and bus cycles if the target line is not always prefetched. The line size is 32 bytes. The t values represent the target thresholds. If target addr - line beginning addr > t then the next consecutive line is prefetched rather than the target line. The cc1 trace is used as simulator input.**

A major hindrance to the performance of wrong-path prefetching is that it initiates target prefetches very late since the control instruction must be decoded before the prefetch request can be issued. Because of this, the wrong-path scheme can never prefetch down a taken conditional branch path or an unconditional jump path. If the target of a pending control instruction were known earlier, misses could be avoided by also prefetching these paths. In addition, the prefetch distance would be increased for all target prefetches. One way to initiate prefetches earlier is to add a prefetch instruction to the architecture. Using an instruction cycle to prefetch data in scientific code has met with some success and it could be useful in prefetching instruction cache lines. A drawback is that the performance gain achieved by miss reduction may be overshadowed by the increased time necessary to execute all of the added prefetch instructions. To investigate the effectiveness of the method, prefetch instructions were inserted into the instruction trace at the beginning of each basic block. The prefetch address was just the target address of the block ending control instruction. One might be tempted at this point to suggest that

the program be profiled to determine the probability of the branch being taken. As shown by the many experiments in this work, that would be a bad idea. Section 4.3 showed that the target path should be prefetched even if it appears that the target direction is never taken.

The prefetch algorithm is still wrong-path prefetch, the target is prefetched regardless of the taken direction of the branch. The difference is that the target prefetches are initiated by an special instruction rather than branch instruction decode. The next-line component of the algorithm remains unchanged. Figure 4.32 shows the results for memory latencies of 4 and 12 cycles. Notice that the performance of prefetch instruction scheme is better or worse than regular wrong-path prefetching depending upon if the time required to execute the prefetch instruction is included in the total CPU cycle time. There are several ways the prefetch execution time can be reduced or eliminated:

- Be more clever in inserting the prefetches - An intelligent compiler could reduce the number of prefetch instructions by inserting the prefetches outside of loops. It could also increase their effectiveness by moving them farther away from the control instruction, thereby increasing the prefetch distance.
- Hide the execution time in superscalar execution - A prefetch instruction has no dependencies and could be scheduled along with any group of instructions.
- Add a prefetch field to a VLIW instruction - The execution time would be hidden at the cost of increased instruction length devoted solely to prefetching.

The instruction would slightly reduce the hardware in the prefetch unit by eliminating the data path from the decoder. On the other hand, inserting prefetch instructions, even if their execution time is hidden, would increase the code size and the required fetch bandwidth. This result shows that a special prefetch instruction could possibly enhance a prefetching scheme but the cost is unclear and would require further architecture analysis.

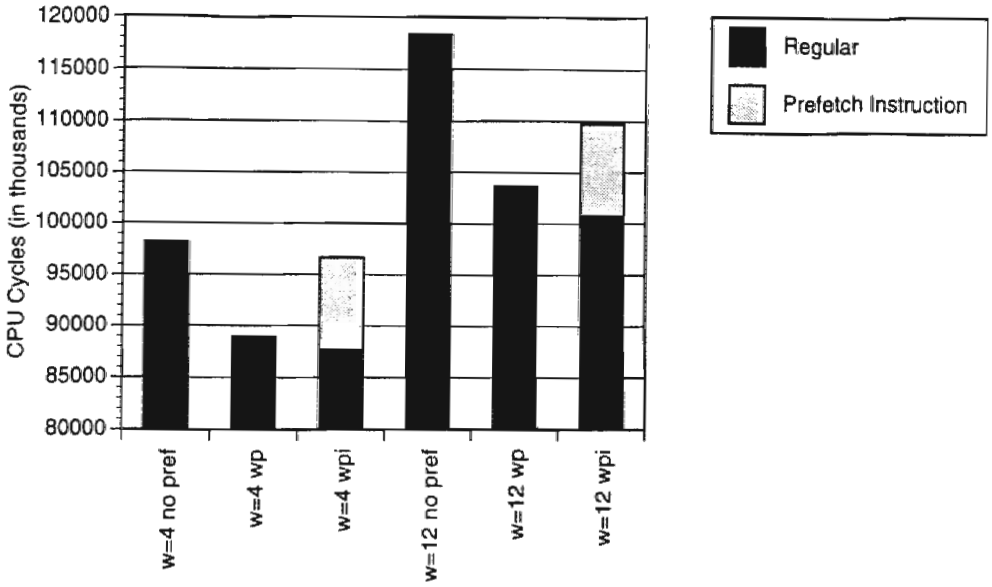


**Figure 4.32 Wrong-Path Prefetching using a Prefetch Instruction - The graphs compares the CPU cycles for no prefetching (no pref), regular wrong-path prefetching (wp), and wrong-path using a prefetch instruction. The results for 4 and 12 cycle memory latencies are shown. The cc1 trace was used as simulator input.**

## 4.8 Summary and Conclusions

This chapter sheds light on the applicability of instruction cache prefetching schemes in current and next-generation microprocessor designs. It also describes a new prefetching algorithm inspired by previous mispredicted path studies. The highlights of the extensive experimental results are:

- Prefetching achieves significant performance gains in terms of CPU cycle reduction - up to 14% reduction with standard cache configurations.
- Wrong-path prefetching achieves higher performance than other algorithms in all studied cache configurations. At the same time, its hardware cost is equivalent to next-line prefetching. Somewhat surprisingly, 75% of all not-taken path prefetches result in miss reductions.
- The cost of prefetching is the increased bus traffic. Bus utilization rises from 12% for no prefetching to up to 25% for prefetching with standard cache configurations. Bus bandwidth must be viewed as a resource which can be employed to reduce miss

cycles. By implementing prefetching and adding hardware to increase refill rate, equivalent memory performance can be achieved with a much smaller cache.

- Table-based target prefetching performs poorly with current generation instruction cache sizes rendering the hybrid algorithm performance equivalent to that of next-line prefetching. The negligible performance gain is not worth the cost of the additional hardware required to implement the target table.
- Prefetching is more effective as memory latencies increase. Wrong-path prefetching reduced CPU cycles by 18% when the wait cycles rose to 16.
- Prefetching is likely to be more effective in next-generation superscalar processors with IPCs greater than 2. Wrong-path prefetching reduced CPU cycles by over 20% in a processor with a 4 instruction issue rate.

The major result of this study is that instruction cache prefetching will become more attractive as a method to reduce cache miss cycles. The trends in next-generation microprocessors point both to larger disparities between CPU cycle times and main memory access time and to higher IPC rates. Even if an L2 cache is implemented to reduce the cycle time disparity, these trends will result in there being a higher proportion of miss cycles in the total execution time. Furthermore, unlike data misses, instruction miss cycles cannot be hidden by performing other work during the miss delay. In the past, miss latency has increased but the number of misses has been reduced by increasing the cache size. However, this is less effective as caches get larger and larger caches have longer access times. Increasing cache associativity also reduces misses but the same disadvantages apply.

Prefetching will effectively reduce instruction miss cycles without increasing the CPU cycle time. Future microprocessors are likely to have more complex memory systems and might include an OMRB structure and multiple memory ports for other architectural features. Therefore, the cost of prefetching in terms of hardware and bus traffic might be less significant.

A prefetch algorithm's ability to shrink required cache size is not a minor advantage in new processor design. Currently proposed architectures such as the MIPS TFP processor and the PowerPC 620 rely upon dual-ported data caches to handle multiple data requests per cycle [25][44]. A multi-ported cache must be kept small for space and speed reasons. If someday there is a need for a dual-ported instruction cache, a scheme which could wring 32K performance out of an 8K cache would be highly valued. Even if not dual-ported, a small instruction cache will leave more space available for a big data cache.

Once the decision has been made to use instruction cache prefetching, there seems to be no reason not to use wrong-path prefetching. It outperforms the other methods - recall Figure 4.9 where the wrong-path prefetched 8K cache outperformed the 16K next-line cache. Also important, its hardware cost is equivalent to next-line prefetching and far less than the hybrid scheme. The bus traffic is only slightly higher than next-line prefetching.

Results from this study are applicable even to processor designs which don't employ prefetching. One piece of conventional wisdom which this study disproves is that not-taken paths should be avoided if at all possible. A current trend in hardware algorithm design is to profile code behavior to lesson wrong path effects. Section 4.3 described a situation where profiling removed beneficial effects. Table-based prefetching is another example of the lengths taken to avoid wrong path actions. A significant amount of hardware is added to predict the correct direction to prefetch. While this seems reasonable, better performance is obtained if the hardware is removed and wrong path effects are allowed to occur. In the next-line prefetching algorithm, the prefetched consecutive cache line is not always immediately referenced because a taken branch was executed near the end of the current line. Common wisdom would call this an unavoidable cost of next-line prefetching. This might not be so. The line has a good chance of being referenced in the near future thereby removing a cache miss.

This idea has implications toward instruction fetch mechanisms. For instance, the fetch strategy in the IBM Power2 architecture is to fetch both directions of a conditional branch. This is necessary so that the required issue rate can be maintained through the branch. However, if the fetch in either direction misses the cache, the fetch is suspended until the correct branch direction is determined. Results in this chapter indicate that following through with the cache miss would reduce the overall number of miss cycles.

# CHAPTER 5

# Instruction Cache Prefetching Model

## 5.1 Introduction

This chapter discusses an analytical model developed to describe the behavior of a prefetched, direct-mapped cache. An accurate cache model is useful for studying both the performance and behavior of cache designs. Although cache performance analysis is predominantly done with either trace-driven simulation or direct hardware monitoring, both are costly and time consuming. A model can produce results with rough accuracy quickly thereby reducing the test space and the required simulation time. In addition, trace-driven simulation is impossible for new architectures since applications do not yet exists. In this case a model is indispensable to fit a cache to the new design.

Cache analysis can also provide an understanding of the behavior of a cache algorithm or of the structure of the cache itself. Model lead to the identifation of different types of misses and the understanding of the interaction of memory references in multiprocess systems. This information can assist in design decisions so that the cache can be structured to handle the types of misses found in the applications for which the architecture is targeted.

Many cache models have been proposed although a literature search did not uncover a model accounting for the effect of prefetching algorithms. A simple axiomatic formula was proposed by Chow and is based upon empirically derived constants [11]. The misses is given by $AC^{B}$ where $C$ is the size of the cache. An intuitive basis for the equation was not given and neither was experimental validation. A more complex formula was

proposed by Singh et al. for fully-associative cache memories [54]. It defines the footprint function $u(t,L)$ to be number of lines referenced up to time $t$. Then

$$u(t, L) = WL^a t^b d^{\log L \log t}$$

where $W$ is a measure of the working set size, $a$ is a measure of the spatial locality, $b$ is a measure of the temporal locality, and $d$ is measure of the interaction between spatial and temporal locality. It is left as an exercise to find the correct values for $W$, $a$, $b$, and $d$.

Strecker first analyzed the transient behavior of cache memories during context switched execution of multiple processes [66]. He noticed a periodicity in cache behavior. Initially, there is a transient period where the process loads the cache with its own data lines. Execution then settles down into a steady-state period with fewer misses. The miss function is of the form (a+bn)/(a+n) where n is the number of cache locations filled and a and b are constants derived by measuring the cache misses for two different cache sizes. While the model is accurate when compared to measured miss rates, Stone points out that a great amount of simulation is required just to fill in the equation's constants [65].

Thiebaut and Stone propose a model which mimics the behavior of two processes executing in round robin fashion competing for cache lines [68]. They develop a statistical model requiring little empirical input. It is based upon the idea that at the beginning of each processes time slice there is a period called a reload transient where lines displaced by the previously running process are reloaded into the cache. The average miss-ratio depends on the length of the miss-laden transient period. The miss equation is based upon a binomial distribution and the line conflict probability is derived from the assumption that all lines are equally likely to be mapped to a set. Thus, the probability that two lines are mapped to the same set is 1/N where N is the number of sets in a direct-mapped cache. Laha, Pael, and Iyer describe a way to estimate the reload transient by accumulating sampled segments of a larger trace [34].

Agarwal et al., extended the multiple period idea to develop the AHH model [2]. Misses are classified based upon one of four causes:

- Start-up effects - similar to the reload period when the processes' working set is loaded into the cache,
- Non-stationary behavior - misses caused by a gradual change in the working set over time,
- Intrinsic interference - interference between the program's references due to cache set conflicts,
- Extrinsic interference - conficts between multiple processes - context switching and multi-process line invalidations.

The full traces is discretized into time granules and the miss equation is based upon the effects of each of the four factors mentioned above for each granule. The model's line conflict component uses a binomial probability distribution assuming a random mapping of lines into cache sets.

Thiebaut relates the cache miss ratio to fractal geometry [69]. He shows that the miss ratio can be computed from the plot of the inter-miss gap distribution and that the miss ratio is a function of the fractal dimension of the inter-miss gaps. The model is based upon a one-dimensional hyperbolic random walk where the probability of the gap size being greater than $u$ is

$$\Pr[U > u] = \left(\frac{u}{u_0}\right)^{-\theta} \qquad u \geq u_0$$

where $u_0$ is a constant and theta is the fractal dimension.

Finally, Quong proposed an intuitive probabilistic model based on time gaps between successive block references [51][52]. In essence, it states that the greater the number of line references between block executions, the greater the probability that the lines in the block will have been displaced from the cache. This model naturally accounts for reload effects as well as non-stationary and intrinsic interference behavior. However, it is based upon on a single processes' block execution pattern and thus cannot account for

the effects of extrinsic interference. This "gap model" is a variant of the LRU stack model of an address trace by Spirn [61] and Rollins [53].

The gap model seems to be the most natural one to extend to encompass prefetching effects. Prefetching alters the order of block executions and thus the gaps between block references. If the gap model can be modified to reflect this reordering, the model is exhibit accurate behavior as well as illuminate the behavior of the prefetch algorithm itself. The next section gives a derivation of the gap model equation for a regular cache and one with different prefetching algorithms. Section 5.2.3 compares the results predicted by the model to measured results using the simulator discussed in Chapter 4. Section 5.4 summarizes the applicability of the derived model.

## 5.2 The Gap Model

### 5.2.1 Derivation of Non-Prefetched Cache Model

The gap model estimates instruction cache misses by calculating the expected value of the number of misses over all possible program mappings. In short, the statistical number of misses is based upon the probability that cache lines in a basic block will be displaced by the execution of other basic blocks before the block is reexecuted. The goal of the derivation is to create an equation which translates the gaps between basic block executions into a cache miss estimate. If this can be done, a program's gap information can be obtained from its execution trace which contains no cache information such as set size, line size, or associativity. The gap information can then be used as input to the analytical model which can approximate cache misses for many cache configurations.

The following model is derived for a direct-mapped instruction cache with L sets (lines). The program's execution trace is divided into basic blocks. A basic block is a sequence of instructions which are always executed together. There is one entry point at the beginning of the block and one exit point at the end. Control instructions within the

instruction trace delimit the basic blocks. Gap, $G$, of block, $b$, is the sequence of blocks which are executed between two executions of $b$. For example,

$$b \quad b_i \quad b_j \quad b_i \quad b_k \quad b.$$
$$\longleftarrow \text{Gap } G \longrightarrow$$

Block b will have a different gap each time it is executed. Let $s_i$ be the size of block $b_i$ in cache lines. Then l is the length of gap $G$ in cache lines and so, continuing with the above example, $l = s_i + s_j + s_i + s_k$. In addition, $u$ is defined to be the size of gap $G$. The size, $u$, is the unique number of cache lines in the gap, $u = s_i + s_j + s_k$. At this point few assumptions should be specified:

- For $1 \le i \le n$, $s_i \ll \mathrm{L}$ - Each block $b_i$ is much smaller than the cache size. This implies that lines within the same block do not conflict with each other.
- Block $b_i$ is equally likely to begin at any line of the cache. For all mapping of the program to memory, there are $n!$ orderings of blocks. If $n$ is large, the assumptions follow.
- $s_i \ge \mathrm{LS}$ - Most blocks are larger than the line size, LS. This assumption holds true if $\mathrm{LS} \le 32$.

If $x$ be the probability that two lines collide in the cache, then $x = 1/\mathrm{L}$. This is reasonable when the program size is much larger than the cache size since, over all possible mappings of the program to the address space, each line has equal probability of mapping into any one line in the cache. As the cache size approaches the program's size this estimate becomes increasingly inaccurate. In general, if $|\mathrm{P}| = (i+f)\,\mathrm{L}$ where i and f represent integral and fractional terms, respectively, i.e., $i \in Z$ and $0 \le f \le 1$, then

$$x = \frac{L\,[f(i+1)\,i + (1-f)\,i\,(i-1)]}{|P|^2}$$

$$= \left(\frac{i^2 + 2fi - i}{(i+f)^2}\right) x \frac{1}{L} \qquad \text{(Eq. 5.1)}$$

$$\approx \frac{1}{L} \qquad \text{when } |P| >> |L|.$$

The simplification x = 1/L will be used in the equations below. However, Equation 5.1 in the actual miss computations.

Now define the random variable $X$ to be the number of cache lines in $b$ replaced during execution of gap $G$. The variable $X$ can be thought of as the number of misses blamed on $b$. Let $X'$ be the random variable describing the first line of $b$, i.e.,

$$X' = \begin{cases} 1 & \text{First line replaced} \\ 0 & \text{First line survived} \end{cases}.$$

The probability that a line will survive through one basic block is $(1 - sx)$ so the probability that the first line of block $b$ will survive gap $G$ is

$$\Pr[X' = 0] = (1 - s_i x)\,(1 - s_j x)\,(1 - s_k x)\,. \qquad \text{(Eq. 5.2)}$$

Notice that even though the block $b_i$ was executed twice within $G$, it is only used once in the above survival probability equation since a repeated execution of a block will not further effect line displacement in the gap. If $s$ is the number of lines in $b$, it is assumed that each line if $b$ has a equal probability of surviving, and the probability that the first line of $b$ is replaced is $\Pr[X' = 1] = 1 - \Pr[X' = 0]$, then the expected value of the number of misses due to $b$ is

$$E[X] = s[1 - (1 - s_i x)(1 - s_j x)(1 - s_k x)]. \qquad \text{(Eq. 5.3)}$$

Now, it can be fairly assumed that the size of each basic block is much smaller than the size of the cache. Thus, $xs_i = \frac{s_i}{L} \ll 1$ and the approximation $1 - s_i x \approx e^{-s_i x}$ can be used. Then

$$\begin{aligned} E[X] &\approx s[1 - (e^{-xs_i})(e^{-xs_j})(e^{-xs_k})] \\ &= s[1 - e^{-x(s_i + s_j + s_k)}] \\ &= s(1 - e^{-xu}) \\ &\approx s(1 - e^{-u/L}) \end{aligned}$$

Finally, the equation which estimates the expected number of instruction cache misses for program $T$ is

$$E[X] = \sum_{b \in T} s(1 - e^{-u/L}) \qquad \text{(Eq. 5.4)}$$

There are two major sources of error in this approximation. The first is the inaccuracy when calculating the block size in terms of cache lines. Cache parameter independent trace information gives block and gap sizes in terms of bytes. These values must be converted into lines to find $s$ and $u$. If a cache line has length $l$ bytes and block $b_i$ is $k$ bytes long, then

$$s_i = (k + l - 1)/l \qquad \text{(Eq. 5.5)}$$

on average, over all possible starting positions. Depending upon the program mapping, the computed $s_i$ could be too big or too small. A constant coefficient, $r$, called the raggedness coefficient will be used to adjust the size calculation, i.e.,

$$s_i = r(k + l - 1)/l. \qquad \text{(Eq. 5.6)}$$

The coefficient is dependent upon the modeled line size but is fixed for all cache sizes.

The other problem with the model is that it assumes that blocks equally likely to conflict with another one. This is untrue since blocks mapped sequentially in memory will not conflict with each other. As the model now stands, all blocks in $u$ are equally likely to displace lines from $b$ causing the model to overestimate the number of misses. As the cache sizes increases the prediction will get worse because fewer blocks can potentially conflict. Quong attempts to minimize these intrablock conflict errors by stipulating that blocks within the same procedure cannot conflict with one another. The variable $u$ is then redefined to be the total size of the unique blocks in $G$ which are not in the same procedure as $b$. The raggedness coefficient is adjusted to further hide this error. I found that, when using this method, the model underpredicted small cache sizes and overpredicted large caches meaning that the average procedure length was somewhere in the middle. Since there is a large disparity in a program's procedure lengths, and the average procedure length is unknown to the cache designer, I chose to divide the program into segments the size of some base cache size. Each block is assigned a tag equal to the block's beginning address modulo the base cache size. Two blocks with the same tag cannot collide with one another since they would be mapped to different cache lines in the base cache. Only blocks with tags different from $b$ can be included in $b$'s gap size calculation. Of course, this only holds for a cache with the base cache size, but it serves as an approximation for other cache sizes. Again, the raggedness coefficient is manipulated to help account for the inaccuracy.

### 5.2.2 Parameter Acquisition

The values of s and u for each basic block b are derived from the program's execution trace.Equation 5.4 specifies that for each basic block, the size of the block and the length of the gap since the block's last execution must be calculated. This data must be stored in a gap information file for use in multiple miss prediction calculations. Unfortunately, a benchmark of 100 million instructions executes roughly 20 million basic blocks and

recording data for each executed block would be too costly. However, it is not necessary to record exact gap sizes. The variable $u$ appears in Equation 5.4as a negative exponential and the calculation is not sensitive to small errors in the gap sizes. The gap sizes can be dividing into bins with sizes of increasing powers of 2. For instance, the first bin would hold gaps of size 0-1, the next bin would hold sizes 2-3, then 4-7, 8-15, etc. Quong suggests that 20 bins are sufficient to give reasonable accuracy. He found that double the number of bins had little effect on the calculations [51].

Equation 5.2 can be used to estimate the probability that a line will survive a gap of a certain size. For small gaps, say $u$ = L/20, the chance of survival is $1 - ux = 1 - 1/20 = 0.95$ or 95% likely to survive. Gaps of the largest bin size or greater can be combined in the largest bin since the probability of a line surviving through a gap of $2^{20}$ or greater is equal to

$$e^{-2^{20}/2^{16}} = e^{-16} \approx 0$$

even for a 64K cache. Cold misses (a block execution with no gap) can be modeled by assigning the block a gap of infinite size thus assuring a miss will be predicted.

The gap information file used in the following miss predictions contains the following information for each basic block: the size (in bytes) of the basic block and the gap structure of 20 bins. Each non-empty bin contains the number of gaps of that bin size and the total number of blocks included in all gaps in the bin. The bin sizes are in bytes and must be converted into lines. The bin sizes cannot be terms of cache lines in the gap information file since the file needs to be cache parameter independent. Suppose $\text{bin}_i$ holds gaps of size $2^i < u$ (in bytes) $< 2^{i+1}$, the number of gaps in $\text{bin}_i$ is $n$, and the total number of blocks making of the $n$ gaps is $B$. Then the average blocks per gap is $B/n$ and the average block size in bytes, $BS_b$, is $2^i/(B/n)$. Equation 5.6 can be then be used to convert $BS_b$ to $BS_l$, the block size in terms of cache lines. Finally, $u = BS\ (n/B)$ for all gaps in $\text{bin}_i$. After $s$ is calculated using Equation 5.6, the number of misses produced by $\text{bin}_i$ is $sne^{-u/L}$.

The results for each bin in each block entry are summed to arrive at the total predicted misses for the program.

### 5.2.3 Non-Prefetched Model Verification

Four SPEC benchmarks listed in the previous chapter are used to verify the accuracy of the gap model. A gap processor program produced gap information files from the execution trace of each benchmark. The execution trace of each benchmark was input to a gap processor program which gathered the model information. The base cache size used to reduce intrablock conflict was 512 bytes. Using the gap information Equation 5.4 predicted number of misses for different cache set and line sizes. The memory simulator discussed in Chapter 4 provided a measure of the actual number of cache misses. Most of the graphs below show a measured curve and two predicted curves. The individual predicted curve uses a raggedness coefficient, $r$, adjusted specially for that benchmark. The uniform predicted curve is based upon a $r$ value constant over all four benchmarks. Obviously the individual predicted curve will more closely match the measured curve than will the uniform curve. Unlike in Quong's work, the $r$ values are different for each line size. For each line size, $r$ was chosen to be the value which enabled the predicted value to best match the measured value for a 512 byte cache. The following graphs show the accuracy of the gap model for the tested benchmarks.

The model successfully predicts the number of misses in the cc1 benchmark. However, it slightly overpredicts for large cache sizes and large line sizes because the model assumptions begin to break down in these areas. For large caches, the line collision probability becomes less than 1/L since addresses are not mapped to all lines will equal probability. In addition, the model overestimates the amount of intrablock conflict for larger sizes which also increases the predicted number of misses. This problem is evident in the miss predictions for all benchmarks. When the line size is large, multiple blocks can fit into one cache line which increases the number of predicted misses. Furthermore, as the

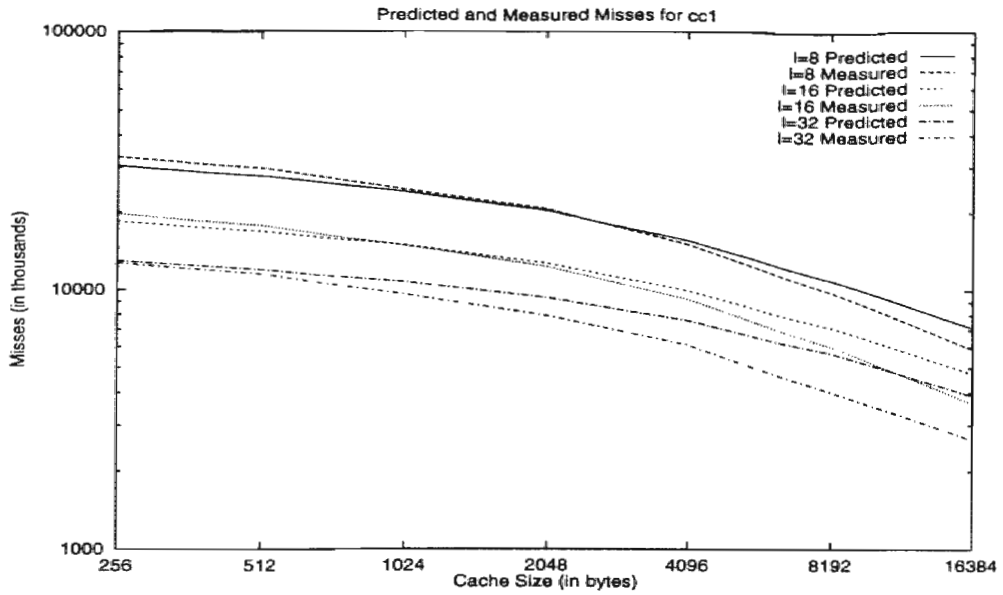


**Figure 5.1 Gap Model Accuracy for cc1 - The graph shows the measured and predicted curves for lines sizes of 8, 16, and 32 bytes.**

line size increases the block size conversion from bytes to cache lines becomes more inaccurate. This can be seen in Equation 5.6.

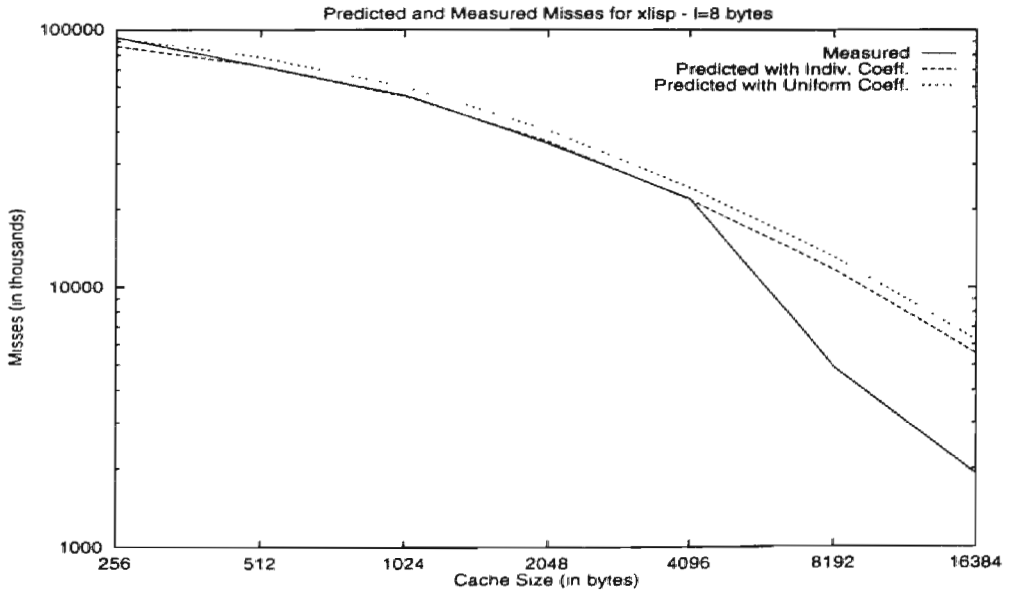


**Figure 5.2 Gap Model Accuracy for xlisp (l=8) - Line size is 8 bytes. The individual predicted curve represents the miss prediction using a constant based only on xlisp. The uniform predicted curve uses a constant which is the same for all four benchmarks.**

The other three benchmarks aren't as accurately predicted. In all them, the model overpredicts the number of misses in large caches. The xlisp benchmark appears to have a work set size of around 4Kbytes. When the cache is larger than 4K, the only misses observed are the initial compulsory misses and relatively few conflict misses. The model is able to predict the sharp decline in misses once the cache size is larger that the working set. The espresso working set seems to fit into a 2Kbyte cache.

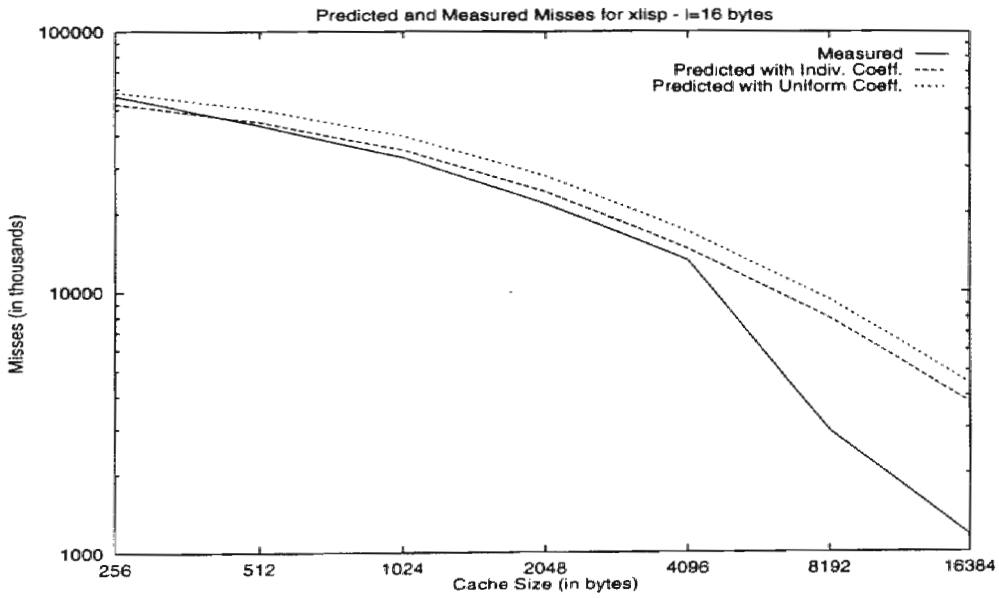


**Figure 5.3 Gap Model Accuracy for xlisp (l=16) - Line size is 16 bytes.**

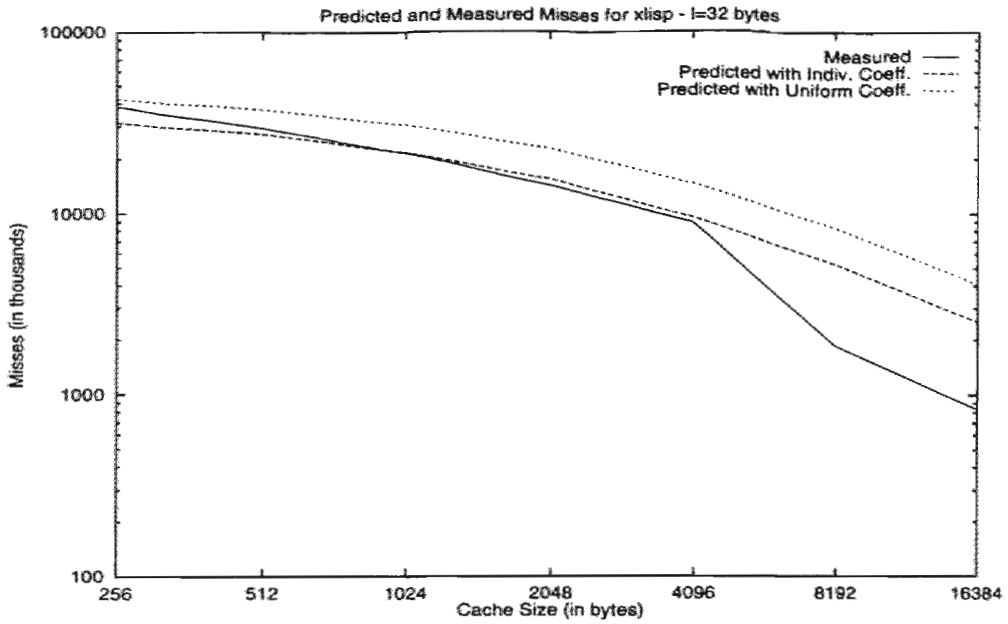


**Figure 5.4 Gap Model Accuracy for xlisp (l=32) - Line size is 32 bytes.**

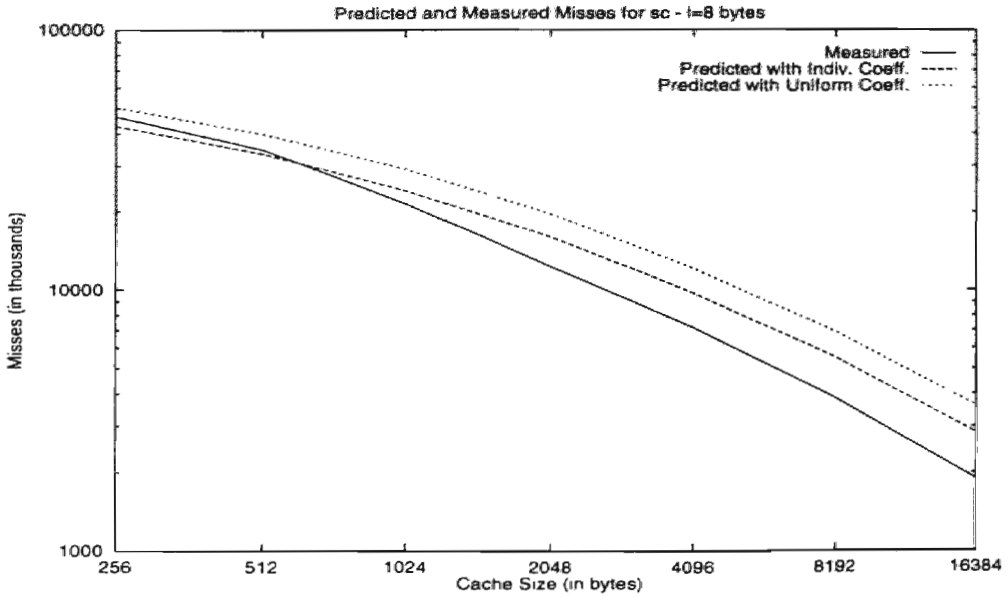


**Figure 5.5 Gap Model Accuracy for sc (l=8) - Line size is 8 bytes.**

### 5.2.4 Next-Line Prefetch Model Derivation

Since the gap model is based upon basic block behavior, it is an appealing candidate to extend to include the effects of prefetching schemes. That was the primary reason for choosing the gap model as an analysis tool. By saving a little more information in the gap information file and modifying Equation 5.6, the gap model can be used to

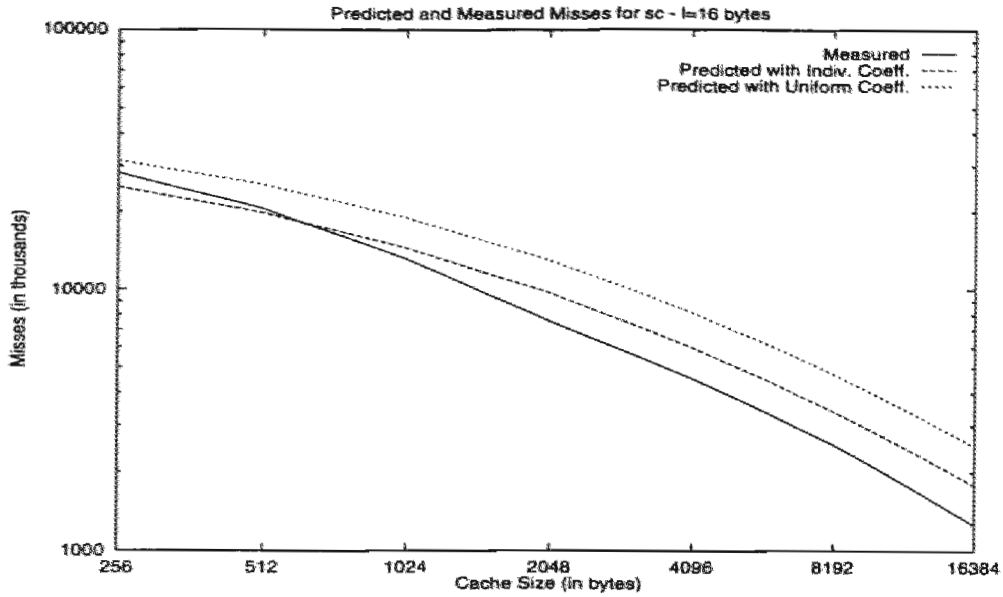


**Figure 5.6 Gap Model Accuracy for sc (l=16) - Line size is 16 bytes.**

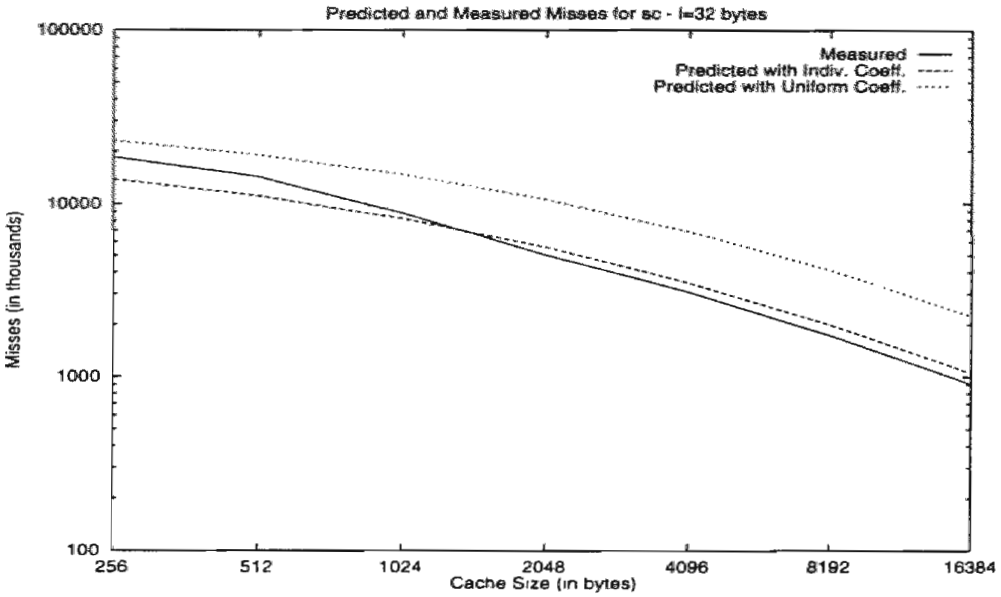


**Figure 5.7 Gap Model Accuracy for sc (l=32) - Line size is 32 bytes.**

predict the number of misses in a next-line prefetched cache. Unfortunately, the number of cache misses has little correlation to actual cache performance since prefetching significantly alters miss latencies as well as the number of misses. However, developing the model is still a worthy exercise since it can give insight into the behavior of a

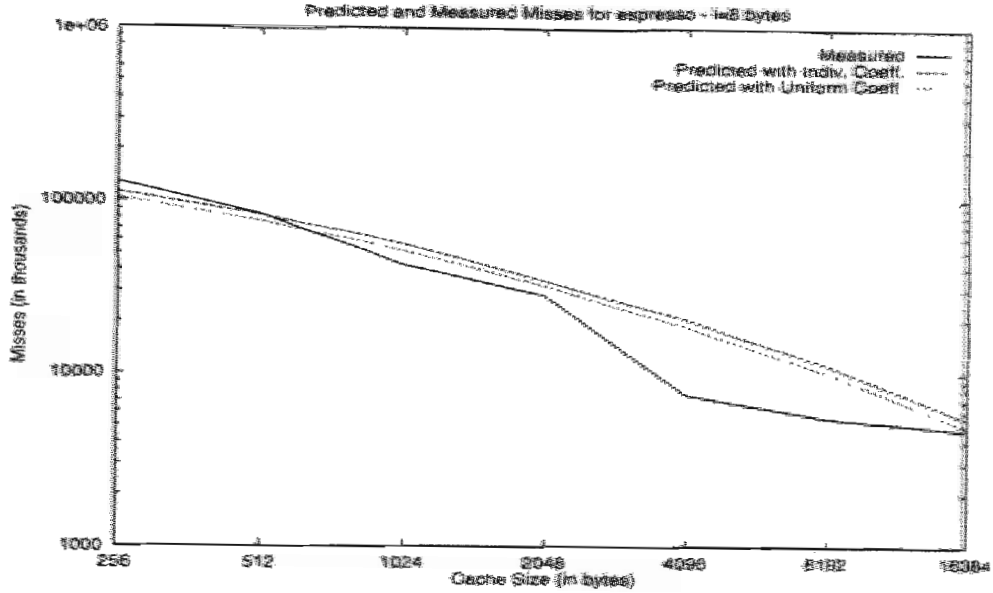


**Figure 5.8 Gap Model Accuracy for espresso (l=8) - Line size is 8 bytes.**

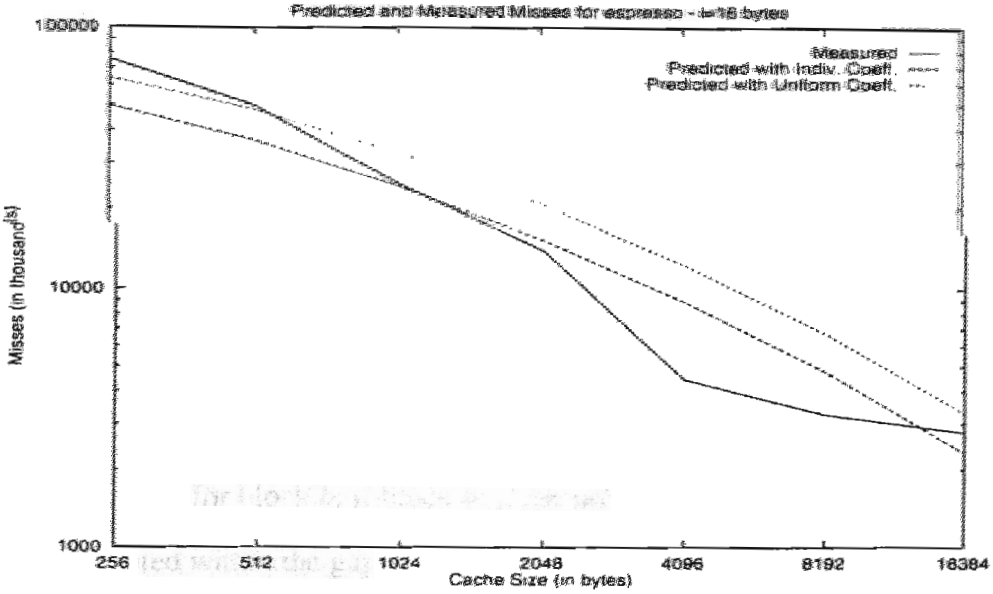


**Figure 5.9 Gap Model Accuracy for espresso (l=16) - Line size is 16 bytes.**

prefetched cache. To analyze the true performance of prefetched and non-prefetched caches, cycle time equations will be derived in Section 5.3.

It will be assumed that the prefetches are performed instantly and that the fetchahead distance is the full line length. These assumptions imply that if cache line $i$ is being executed, then line $i+1$ will reside in the cache. Thus, only the first line of block can

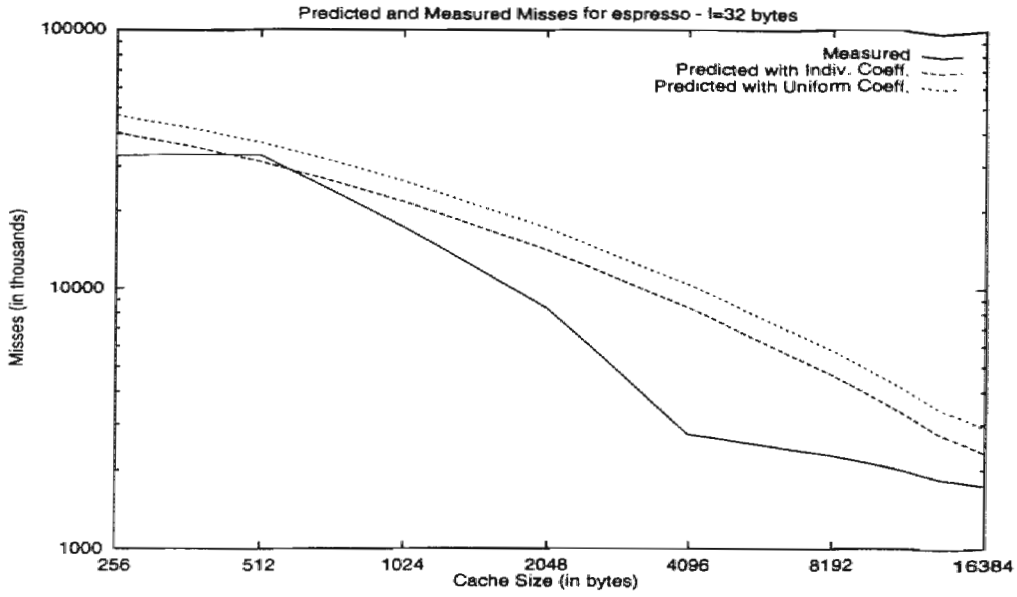


**Figure 5.10 Gap Model Accuracy for espresso (l=32) - Line size is 32 bytes.**

potentially cause a cache miss. In addition, the line succeeding an executed block will always be prefetched into the cache.

Let $X'$ be the same random variable as before, namely

$$X' = \begin{cases} 1 & \text{First line of b replaced} \\ 0 & \text{First line of b survives} \end{cases}$$

Let $Y'$ be another random variable such that

$$Y' = \begin{cases} 1 & \text{Prefetch of first line of } b \text{ replaced} \\ 0 & \text{Prefetch of first line of } b \text{ survives} \end{cases}$$

A prefetch will occur for block $b_i$ if block $b_{i-1}$, the preceding block in the program mapping, is executed within the gap of $b$. For most prefetches, it is supposed that block $b_{i-1}$ will be executed immediately before $b_i$ in which case the prefetched line is guaranteed to be resident. However, sometimes $b_{i-1}$ is executed somewhere in the middle of the gap and, in this case, the probability of the line existing in the case is less than one and is proportional to the number of lines accessed between the executions of $b_{i-1}$ and $b_i$. To compute the probabilities for X and Y two gap sizes will be used: $u$ for the distance between the two executions of $b$, and $v$ for the distance between $b_{i-1}$ and $b_i$ if $b_{i-1}$ is

executed within the gap of $b$. So, the probability that the first line of $b$ which was prefetched by $b_{i-1}$ will be resident when $b_i$ is executed is

$$\Pr [Y' = 0 | b_{i-1} \in G_b] = e^{-w/L}$$

Now define a third random variable $Z'$, such that

$$Z' = \begin{cases} 1 & \text{First line of } b \text{ causes miss} \\ 0 & \text{First line of } b \text{ does not cause miss} \end{cases}$$

Then,

$$\Pr [Z' = 0] = \begin{cases} e^{-u/L} & b_{i-1} \notin G_b \\ e^{-w/L} & b_{i-1} \in G_b \end{cases} \qquad \text{(Eq. 5.7)}$$

If Z is defined to be the number of lines of block $b$ replaced during $G$, the expected number of misses blamed on $b$ is

$$E [Z] = 1 - \Pr [Z' = 0] .$$

Notice that the block size, $s$, is not included in the expected value equation as it was in Equation 5.4. This is because only the first line reference in a block can potentially miss the cache. Finally, the expected number of misses for the entire program is

$$E [Z] = \sum_{b \in T} (1 - \Pr [Z' = 0]) . \qquad \text{(Eq. 5.8)}$$

The $Y'$ random variable incorporates the prefetch effect into the model. What about the pollution cause by next-line prefetching? Pollution can be generated whenever the fall-through line after a block is prefetched but the block-ending branch is taken. During the execution of gap $G$, additional lines are accessed which increase $u$ and $v$ beyond the sum the of block sizes. In fact, one additional line is accessed for every unique

block in $G$ which ends in a taken branch. However, it is unclear how many of these extra accesses actually bring an additional line into the cache during the execution of the gap. They could assess lines already present or they could access lines which will referenced later in the gap. The latter would be a prefetch for another block within $G$. It is possible to record the action of each prefetch when creating the gap information file but the file would be much larger as would the time required to create it. In the previous chapter, it was noticed that next-line prefetches are about 75% accurate, i.e., only 25% of the time is a prefetched line not accessed. So, as an approximation, I state that 1 out of every 4 blocks in $u$ and $v$ which end in a taken branch cause one additional line reference. The probability that the target is taken in each block is recorded in the gap file.

### 5.2.5 Next-Line Prefetch Model Verification

The predicted graphs shown in this section use the same raggedness coefficients as

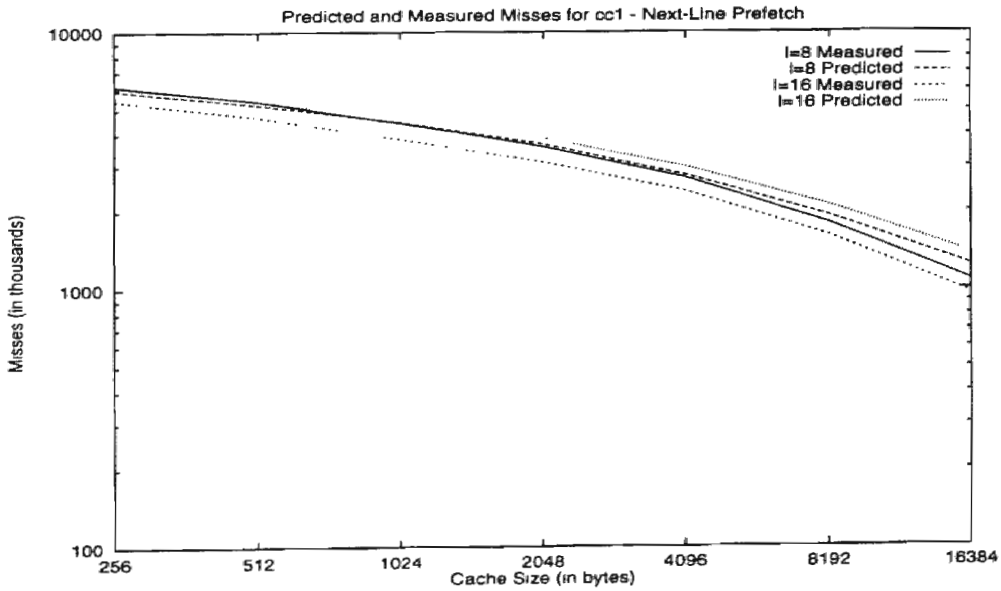


**Figure 5.11 Next-Line Prefetching Model for cc1 - The line sizes are 8 and 16 bytes.**

used in Section 5.2.3. The next-line model predicts prefetched misses at least as well as the original gap model predicted regular gap misses.

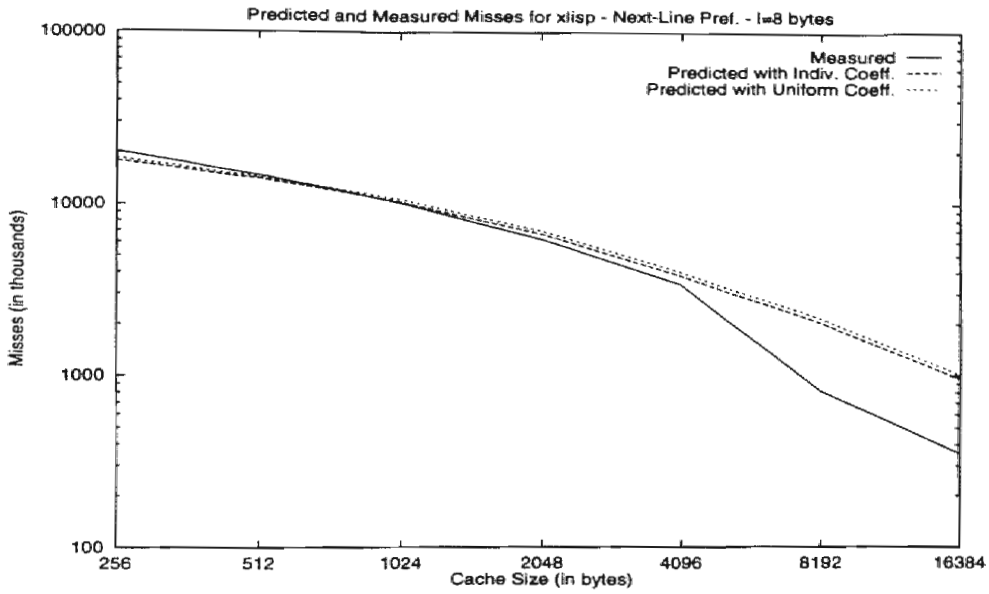


**Figure 5.12 Next-Line Prefetching Model for xlisp - The line size is 8 bytes.**

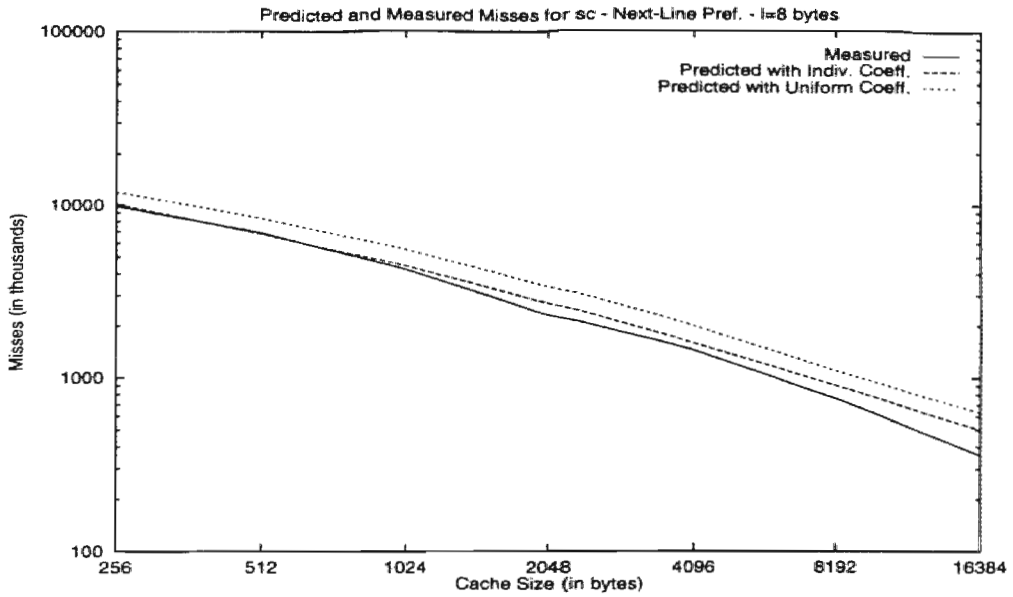


**Figure 5.13 Next-Line Prefetching Model for sc - The line size is 8 bytes.**

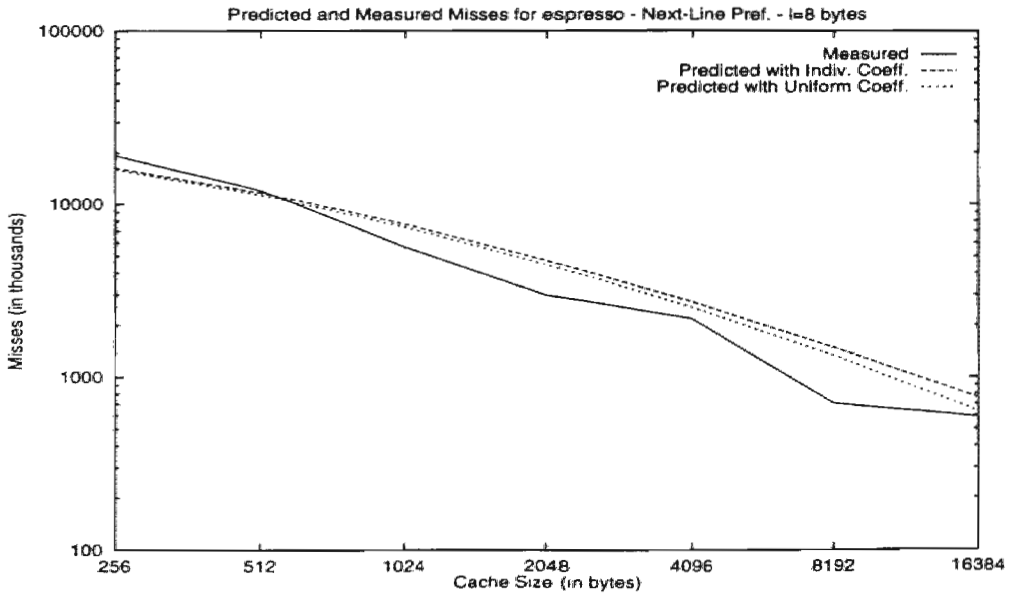


**Figure 5.14 Next-Line Prefetching Model for espresso - The line size is 8 bytes.**

### 5.2.6 Wrong-Path Prefetch Model

To derive the wrong-path model, the gap model will be extended in much the same way as it was in the next-line model. Define $\{T_{ij}\}$ to be the set of blocks which 1) end with

conditional branches and 2) the target of the conditional branch is $b_i$. Then, in the wrong-path algorithm, a prefetch could be initiated if block $b_{i-1}$ is executed within $G$ or if any $T_{ij}$ is within $G$. The probability that the prefetch will survive until the next execution of $b_i$ is proportional to the number of cache lines accessed between the last prefetch and the execution of $b_i$. Let $u$ and $v$ be the same as in the next-line model, i.e., the size of $G$ and the number of cache lines referenced between $b_{i-1}$ and $b_i$. Now define $t_j$ to be the number of lines referenced between $T_{ij}$ and $b_i$. The set $\{t_j\}$ would be empty if no target blocks $T_{ij}$ were executed within $G$. Then, using the same random variables Z and $Z'$,

$$\Pr [Z' = 0] = e^{-\min \{u, v, t_j\} x} \qquad \text{(Eq. 5.9)}$$

and

$$\mathrm{E} [Z] = \sum_{b \in T} (1 - e^{-\min \{u, v, t_j\} x}) . \qquad \text{(Eq. 5.10)}$$

It is easier to account for the pollution in the wrong-path algorithm. In the next-line algorithm, this was done by slightly increasing $u$ and $v$ depending upon the exiting behavior of the executed blocks. In this case, one additional line will be referenced for each block in the gap regardless of the exiting behavior. If the block ends in a conditional branch, both paths are be prefetched and so one prefetch is wrong and causes an additional access. If the block ends with an unconditional jump, call, or return, the next-line component produces an additional access. The approximation is used to decide how many prefetches actually bring in new lines. In this case, 1 out of every 4 unique blocks in $G$ add 1 line to the size of $u$. The values $v$ and $t_j$ are increased in a similar fashion.

### 5.2.7 Wrong-Path Prefetch Model Verification

In the prediction graphs shown below, the raggedness coefficients are the same values as used in the non-prefetched and next-line prefetched cache results. Like the next-line prefetched model, this model is at least as accurate as the original gap model.

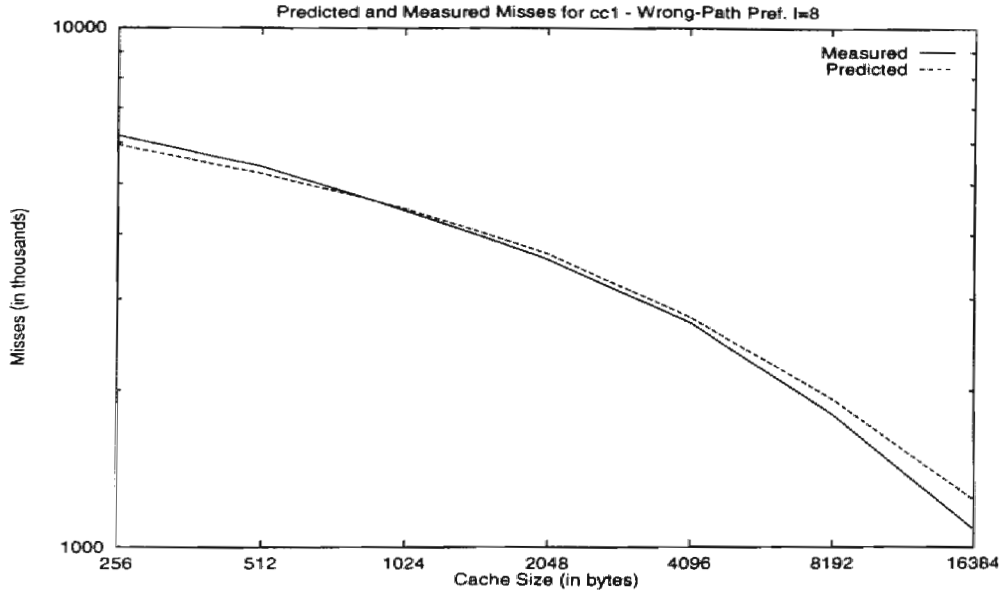


**Figure 5.15 Wrong-Path Model for cc1 - The line size is 8 bytes.**

## 5.3 CPU Cycle Model

As stated in Section 4.4.4, the number of misses or miss ratio is an insufficient performance parameter. While the above models do give reasonably accurate figures for the number of misses in prefetched caches, they give no insight into the total number of CPU cycles required to execute the benchmark. This section describes equations which approximate the total CPU cycles for an architecture with non-prefetched and prefetched direct-mapped caches. The miss numbers obtained from the gap models are inputs into the model.

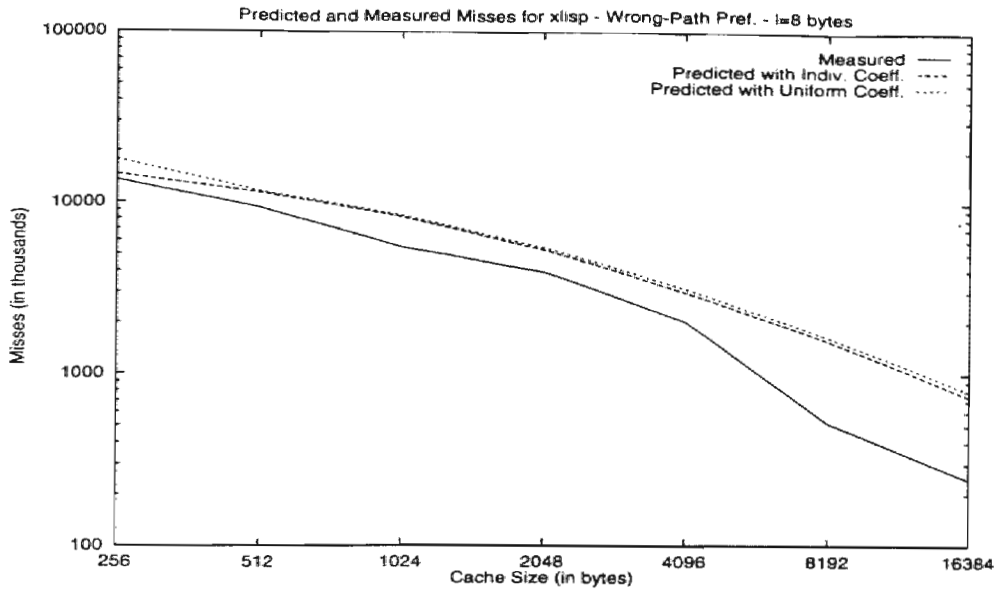


**Figure 5.16 Wrong-Path Model for xlisp - The line size is 8 bytes.**

It is unrealistic to assume that a model can be produced which will accurately predict the performance of a non-prefetched cache, let alone a more complex prefetched one. Runtime depends not only on miss rates but also upon instruction execution time, refill rate, and memory latency. The interactions between these parameters is complex and programs react in different ways to parameter variations. Compiler studies that shown that even slight modifications in instruction execution order can significantly change the overall cache performance.

This section gives equations which describe a cache's cycle performance in terms of delay factors. The cause of the delay is analyzed and, when possible, a formula based upon empirical results is given. If a formula cannot be derived, cache parameter dependencies will be discussed. It will be assumed that every instruction will take one cycle to execute. The refill rate and memory latency will vary and, along with cache size and line size, be parameters of the model.

### 5.3.1 Non-Prefetched Cache

The non-prefetched cache model is relatively straightforward. The equation is

$$\text{Cycles} = i + \text{fm}_o\,(w + d_1) + \text{pm}_f d_2. \qquad \text{(Eq. 5.11)}$$

The variable definitions are given in Table 5.1. The constant $d_1$ represents the average number of cycles required for an instruction miss to gain control of the bus in order to send a fetch request. No data effects are considered in this model so only outstanding instruction misses can contend for bus cycles. With no prefetching, partial misses can occur only when the line refill rate is less than the cache line size. An instruction fetch can request part of a line which is in transit from memory. The average resulting delay, $d_2$, will be greater than zero but less than $w$.

The memory simulator was modified to display the values of the parameters found in Equation 5.11 and benchmark simulation were run in order to derive formulas approximating the behavior of $d_1$, $d_2$, and $\text{pm}_f$. It was observed that $d_1$ is a function of $c$, $l$, and $r$ but is independent of the memory latency. It is strongly dependent upon the ratio $l/r$ as can be seen in the formula

$$d_1\,(c, l, r) \approx \begin{cases} 4^{\log\,(l/r)\,-\,3} & l/r \geq 4 \\ 0 & \text{otherwise} \end{cases} \qquad \text{(Eq. 5.12)}$$

When $l/r$ is held fixed, $d_1$ decreases slightly as $c$, $l$, and $r$ are increased. This is because increasing any of those 3 parameters will decrease the number of miss requests thereby reducing the contention for the bus. This will lower the probability that a fetch

memory request will be delayed. A delay is not observed if the bus can refill the line in 1 or 2 cycles.

| Parameter | Definition |
|---|---|
| $i$ | Number of instructions executed |
| $fm_o$ | Full misses - misses requiring full latency period (no prefetch) |
| $w$ | Memory latency |
| $pm_f$ | Partial misses - Fetch misses not requiring full latency |
| $d_1$ | Average fetch initiation delay |
| $d_2$ | Average partial miss delay |
| $c$ | Cache size (in bytes) |
| $l$ | Line size (in bytes) |
| $r$ | Refill rate (bytes/cycle) |

**Table 5.1 Equation Parameter Definitions**

The same holds true for $d_2$. It also is a function of $c$, $l$, and $r$ but is independent of $w$. As $c$, $l$, and $r$ are increased, $d_2$ is slightly reduced again because the number of misses is reduced. The formula for $d_2$ is

$$d_2(c, l, r) \approx \begin{cases} (\log(l/r) - 3) & l/r \geq 4 \\ 0 & \text{otherwise} \end{cases} . \qquad \text{(Eq. 5.13)}$$

Finally, $pm_f = K fm$ where $K$ is a function of $c$, $l$, and $r$. For fixed $l/r$, $K$ decreases as $l$ and $r$ increase. It is slightly dependent upon $c$ but the correlation is unclear. The partial misses can be approximated by

$$pm_f(c, l, r) \approx \begin{cases} \frac{3^{\log(l/r) - 3}}{20} fm & l/r \geq 4 \\ 0 & \text{otherwise} \end{cases} . \qquad \text{(Eq. 5.14)}$$

Equation 5.11 and the derived parameter formulas can now be used in conjunction with the gap model miss prediction to calculate meaningful memory system performance estimates when using a non-prefetched cache.

### 5.3.2 Prefetched Cache Performance

The prefetched cache model is an extension of the non-prefetched model. The equation is

$$\text{Cycles} = i + \text{fm}_n (w + d_1) + \text{pm}_f d_2 + \text{pm}_n d_3 \qquad \text{(Eq. 5.15)}$$

for a next-line prefetched instruction cache and

$$\text{Cycles} = i + \text{fm}_w (w + d_1) + \text{pm}_f d_2 + \text{pm}_n d_3 + \text{pm}_t d_4 \qquad \text{(Eq. 5.16)}$$

for a wrong-path prefetched cache. Definitions of the additional parameter are given in Table 5.1. The $\text{pm}_n$ and $\text{pm}_t$ terms describe the number of misses whose latency is shortened because of a recently executed next-line prefetch or target-line prefetch respectively. The average latency over all next-line partial misses is $d_3$. The average latency for partial target misses is $d_4$. The miss predictions is Section 5.2 showed that $\text{fm}_t < \text{fm}_n < \text{fm}_o$ but it is unclear that this will translate into higher performance because $d_1$, $\text{pm}_f$, and $d_2$ might all increase because of increased cache activity. In addition, the $\text{pm}_n$ and $\text{pm}_t$ terms will be non-zero and further contribute to the cycle time.

Formulas were much more difficult derive for the parameters in prefetch equation for two reasons. One was that the parameter values strongly depended upon many cache parameters. The other, probably related to the first, is that different benchmarks react quite differently to variations in parameter values making it impossible to define a formula applicable to even the four observed benchmark. The following are some observations which help to describe the behavior of the prefetch algorithms.

| Parameter | Definition |
|---|---|
| $\text{fm}_n$ | Full misses using next-line prefetching |
| $\text{pm}_n$ | Partial misses due to next-line prefetching |

**Table 5.2 Prefetch Equation Parameter Definitions**

| Parameter | Definition |
|---|---|
| $fm_w$ | Full misses using wrong-path prefetching |
| $pm_t$ | Partial misses due to target-line prefetching |

**Table 5.2 Prefetch Equation Parameter Definitions**

First, $fm_n$ and $fm_w$ can be estimated using the previously discussed gap model. The values of $pm_f$ and $d_2$ are almost identical to those found in the non-prefetching model. This makes sense because $pm_f d_2$ represent the fetch latency due to multiple cycle transfers. The factor will be non-zero when instructions are fetched from a line which is only partially cache resident. Thus, $d_2$ represents the time between receiving the first line segment and the referenced line segment. This time will be unaffected by the additional bus contention caused by prefetching and there is no reason to believe that the number of partial fetch misses would change significantly with prefetching. Furthermore, the factor $pm_f d_2 \approx 0$ unless $l/r$ is large, say $l/r \geq 32$.

The parameters $pm_n$ and $pm_t$ vary substantial for different programs. However, for $l/r > 4$, the sum of the full and partial misses in next-line prefetching is roughly equivalent to the sum of full and partial misses in wrong-path prefetching. In addition, $pm_n$ is roughly the same in both next-line and wrong-path prefetching. Therefore, $fm_n \approx fm_w + pm_t$, i.e., wrong-path prefetching doesn't remove many misses but instead it converts them into less costly partial misses. When the line can be transferred in one or two cycles, $l/r < 4$, the miss sum and $pm_n$ is less in wrong-path prefetching than in next-line prefetching. More of the misses are completely removed by wrong-path prefetching. This observation that wrong-path prefetching is more closely tied to refill rate suggests that the first segment of the line received by the cache is often not the first segment needed. The segment containing the target address is the first segment received. It seems that often instructions at locations other than the target are accessed first. This is in agreement with the profile experiment in Section 4.3 which showed that even prefetched lines down not-taken paths contribute to miss reduction.

In a prefetched cache, the delay parameters strongly depend upon the line size and delay cycles as well as *l/r*. The increased bus activity in a prefetched cache probably makes these dependencies visible. The values $d_1$ and $d_3$ are slightly higher in wrong-path prefetching than in next-line, again probably due to the higher bus activity.

## 5.4 Summary

The gap model does a reasonable job approximating the number of misses in a non-prefetched instruction cache. Benchmarks with a large amount of cache activity seem to be predicted the best. The model begins to fail as the cache size grows to contain the benchmark's working set. The model's extension to encompass the effects of prefetching was surprisingly effective. The prefetch models accurately predicted the reduced number of misses for both next-line and wrong-path prefetching. Unfortunately, the model extension exercise is mostly academic, since the observed miss reduction does not directly translate into improved performance. Further equations were given which describe the performance of a cache memory system based upon cache misses, cache configuration parameters, refill rate, and memory delay cycles. It was found that deriving formulas for the equations' parameters which are representative over many benchmarks was impossible. However, dependency trends are observed which further describe the interaction of prefetching algorithms and various cache configurations.

# CHAPTER 6
# Summary

I have proposed a new instruction cache prefetching algorithm which derives substantial performance gains from prefetching down not-taken target paths. This idea counters the conventional design of prefetching algorithms where a strong and sometimes costly effort is made to prefetch down only immediately taken execution paths. However, compared with other conventional schemes, wrong-path prefetching achieves higher performance at a lower or at least equivalent hardware cost. Cycle time was reduced by up to 16% and the number of cycles wasted due to instruction misses was reduced by up to 62%. Even more encouraging, instruction cache prefetching was shown to be more effective in future architecture designs incorporating multi-instruction issue and increased memory wait cycles. Wrong-path prefetching reduced CPU cycle time by over 20% in a 4 instruction per cycle simulation. These are significant cycle reductions which can be obtained at low hardware cost and without negatively affecting CPU cycle time. Regardless of the magnitude cycle time reduction, the effectiveness of wrong-path prefetching allows one to conclude that some hardware intensive methods such as table-based prefetching are not worth their implementation costs.

The disadvantage of all prefetching algorithms is that they increase bus traffic. However, bus bandwidth must be viewed as any other hardware resource which can be utilized to decrease cycle time. If the bus is allowed to expand to handle the extra traffic generated by prefetching, sizable hardware area can be freed from the cache structure. For instance, simulations show that if wrong-path prefetching is implemented and the bus width is increased from 4 to 32, an 8 Kbyte cache will outperform a non-prefetched 32

Kbyte cache. Because of improvements in die mounting technology, chip bin count is becoming less of an issue and thus, wide cache to memory buses are feasible. Since software working sets are becoming larger and the growth in cache size is not limitless, the bus/cache size trade-off is appropriate.

If one phrase were used to summarize my thesis it would be that wrong-path effects can positively affect cache behavior. The references generated by the execution of these not-taken paths have an overall prefetching effect. The impression that mispredicted-path references would increase cache pollution is not entirely false but it was found to have only a secondary effect and is outweighed by prefetching. The effectiveness of wrong-path prefetching typifies the benefits wrong-path references can have toward cache performance.

# BIBLIOGRAPHY


[1] M. Accetta, " Mach: A new kernel foundation for UNIX development," In the *Summer 1986 USENIX Conference*, USENIX Association, Bekeley, California, 1986.

[2] A. Agarwal, J. Hennessy, and M. Horowitz. "An analytical cache model," *ACM Transactions on Computer Systems*, **7**, no. 2, 184-215, May 1989.

[3] A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Kluwer Academic Publishers, Norwell, MA, 1989.

[4] A. Agarwal, R. Sites, and M. Horowitz. "ATUM: A new technique for capturing address traces using microcode," In *Proceedings of 13th Annual Symposium on Computer Architecture*, pp. 119-127, June 1986.

[5] D. Alpert, and D. Avnon, "Architecture of the Pentium microprocessor," *IEEE Micro*, pp. 11-21, June 1993.

[6] T. Ball and J. Larus, "Optimally profiling and tracing programs," In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, Jan. 1992.

[7] M. Butler, T. Yeh, and Y. Patt, "Single instruction stream parallelism is greater than two," In *Proceedings of the 21st Internation Symposium on Computer Architecture*, pp. 276-286, May 1991.

[8] J. B. Chen, *The Impact of Software Structure and Policy on CPU and Memory System Performance*, Technical Report CMU-CS-94-145, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1994.

[9] J. B. Chen, "Software methods for system address tracing," In *Proceedings of the 4th Annual Workshop on Workstation Operating Systems*, October 1993.

[10] T. Chen and J. Baer, "Reducing memory latency via non-blocking and prefetching caches," In *Proceedings of the 5th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 51-61, Oct. 1992.

[11] C. K. Chow. "Determining the optimum capacity of a cache memory," *IBM Technical Disclosure Bulletin*, **17** no. 10, pp. 3163-3166, March 1975.

[12] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution

profiling," In *SIGMETRICS*, Nashville, TN, ACM, pp. 128-137, 1994.

[13] B. Cmelik, *SpixTools Introduction and User's Manual*, Technical Report SMLI TR-93-6, Sun Microsystems Laboratory, Mountain View, CA, February 1993.

[14] Digital Equipment Corp., *Alpha Architecture Handbook*, 1992.

[15] S. Eggers, D. Keppel, E. Koldinger, and H. Levy, "Techniques for efficient inline tracing on a shared-memory multiprocessor," In *Proceedings of 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 37-47, May 1990.

[16] M. K. Farrens and A. R. Pleszkun, "Improving performance of small on-chip instruction caches," In *Proceedings of 16th Annual Symposium on Computer Architecture*, pp. 234-241, June 1989.

[17] K. Flanagan, K. Grimsrud, J. Archibald, B. Nelson, *BACH: BYU Address Collection Hardware*, Technical Report TR-A150-92.1, Dept. Of Electrical and Computer Engineering, Brigham Young University, Provo, UT, Jan. 1992.

[18] J. Gee, M. Hill, D. Pnevmatikatos, and A. Smith, *Cache Performance of the SPEC Benchmark Suite*, Technical Report, Dept. of Computer Science, University of Californial, Berkeley, 1992.

[19] G. Gircys, *Understanding and Using COFF*, O'Reilly & Associates, Sebastopol, CA.

[20] G. F. Grohoski and J. H. Patel, "A performance model for instruction prefetch in pipelined instruction units," In *Proceedings of the 9th International Symposium on Parallel Processing*, pp. 248-252, August 1982.

[21] R. Groves, and R Oehler, RISC System/6000 Processor Architecture, IBM RISC System/6000 Technology, SA23-2619, IBM Corporation, pp. 16-24, 1991.

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

[23] M. D. Hill, "A case for direct-mapped caches,"

[24] M. D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, Doctoral Thesis, Department of Computer Sciences, University of Californial, Berkeley, California.

[25] P. Hsu, "Design of the TFP microrocessor," *IEEE Micro*, 1993

[26] W.-C. Hsu and J. Smith, "Prefetching in supercomputer instruction caches," *Supercomputing '92*, pp. 588-597, November. 1992.

[27] Intel Corp., *i486 Hardware Reference Reference Manual*, 1990.

[28] Intel Corp., *i486 Microprocessor Programmer's Reference Manual*, 1990.

[29] Intel Corp., *UNIX SystemV Rel. 4.0 Programmer's Guide*, Order #465800-001, 1990.

[30] N. Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," In *Proceedings of the 17th Int. Symp. on Computer Architecture*, pp. 364-373, Aug. 1990.

[31] Kane, Gerry, *MIPS R2000 RISC Architecture,* Prentice Hall, Englewood, Cliffs, NJ, 1987.

[32] A. Klaiber and H. Levy, "An architecture for software-controlled data prefetching," In *Proceedings of the 18th Int. Symp. on Computer Architecture*, pp. 43-53, May 1991.

[33] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," In *Proceedings of the 8th Int. Symp. on Computer Architecture*, pp. 81-87, May 1981.

[34] S. Laha, J. Patel, and R. Iker, "Accurate low-cost methods for performance evaluat of cache memory systems," *IEEE Transactions on Computers*, **C-37**, no. 11, pp. 1325-1336, November 1988.

[35] M. Lam and R. Wilson, "Limits of control flow on parallelism," 1992.

[36] J. Larus, "Efficient program tracing," *Computer*, pp. 52-60, May 1993.

[37] J. Larus and T. Ball, "Rewriting executable files to measure program behavior," *Software Practice and Experience*, **24**, pp. 197-218, Feb. 1994.

[38] D. Mazieres and M. D. Smith, *Abstract Execution in a Multitasking Environment*, Technical Report 31-94, Center for Research in Computing Technology, Harvard University, Cambridge, MA, November 1994.

[39] E. McLellan, "The Alpha AXP Architecture and 21064 processor," *IEEE Micro*, pp. 26-47, June 1993.

[40] Microprocessor Report, MicroDesign Resources, Sebastopol, CA, Aug. 23, 1993.

[41] Microprocessor Report, MicroDesign Resources, Sebastopol, CA, Sept.12, 1994.

[42] Microprocessor Report, MicroDesign Resources, Sebastopol, CA, March 28, 1994.

[43] Microprocessor Report, MicroDesign Resources, Sebastopol, CA, Oct. 3, 1994.

[44] Microprocessor Report, MicroDesign Resources, Sebastopol, CA, Oct. 24, 1994.

[45] D. Nagle, R. Uhlig, and T. Mudge, *Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures*, Technical Report TR-147-92, The University of Michigan, 1992.

[46] D. Nagle, R. Uhlig, T. Stanley, T. Mudge, S. Sechrest and R. Brown, "Design tradeoff for software-managed TLBs," In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 27-38.

[47] A. Nicolau and J. Fisher, "Measuring the parallelism available for very long instruction word architectures," *IEEE Transactions on Computers*, **C-33**, no. 11, pp. 968-976, November 1984.

[48] J. Pierce and T. Mudge, *IDtrace: A Tracing Tool for i486 Simulation*, Technical Report CSE-TR-203-94, Dept. of Electrical Engineering. and Computer Science, University of Michigan, Jan. 1994.

[49] J. Pierce and T. Mudge, "The effect of speculative execution on cache performance," In *Proceedings of the Int. Parallel Processing Symposium*, pp. 172-179, April 1994.

[50] J. Quinlan, and K. Lai, *Tynero: A Multiple Cache Simulator*, Technical Report, Intel Corp., Hillsboro, OR, May 1991.

[51] R. Quong, "Expected i-cache miss rates via the gap model," In *Proceedings of the 21st Internation Symposium on Computer Architecture*, pp. 372-383, April 1994.

[52] R. Quong, *Expected Values for Cache Miss Rates for a Single Trace*, Technical Report TR-EE 93-26, Purdue University, July 1993.

[53] T. Rollins, W. Strecker, "Use of the LRU stack depth distribution for simulation of paging behavior," *Communications of the ACM* **20**, no. 7, pp. 785-798, Novemeber 1977.

[54] J. Singh, H. Stone, and D. Thiebaut, "An analytical model for fully associative cache memories," *IEEE Transactions on Computers*, **41**, no. 7, 811-825, July 1992.

[55] A.J. Smith, "Cache memories," *ACM Computing Surveys*, pp. 473-530, September 1982.

[56] J.E. Smith, "A study of branch prediction strategies," In *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135-148, May 1981.

[57] M. D. Smith, M. Johnson, and M. 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.

[58] M. D. Smith, *Tracing with Pixie*, Technical Report CSL-TR-91-497, Center for Integrated Systems, Stanford University, Nov. 1991.

[59] G. Sohi, and M. Franklin, "High-bandwidth data memory systems for superscalar processors," In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 53-62, April 1991.

[60] SPEC. *The SPEC Benchmark Suite*, SPEC Newsletter, 3:3-4, 1991.

[61] J. Spirn, *Program Behavior: Models and Measurements. Operating and Programming Systems Series*. Elseier Scientific Publishing Co., Inc., New York, 1977.

[62] A. Srivastava and A. Eustace. "ATOM: A system for building customized program analysis tools," In *Proceedings of SIGPLAN '94 Conf. on Programming Language Design and Implementation*, June 1994.

[63] A. Srivastava and D. Wall, *A Practical System for Intermodular Code Optimization at Link-Time*, Research Report 92/6, DEC Western Research Laboratory, Palo Alto, CA, Dec. 1992.

[64] C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. Shen, "Instruction level profiling and evaluation of the IBM RS/6000," In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Toronto, Canada, pp. 180-189, 1991.

[65] H. Stone, *High-Performance Computer Architecture*, Addison-Wesley Publishing Co., Inc. Reading, Massachusetts, 1993.

[66] W. D. Strecker, "Transient behavior of cache memories." *ACM Transactions on Computer Systems*, **1**, no. 4, pp. 281-293, November 1983.

[67] Sun Microsystems, *The Sparc Architecture Manual*, 1989.

[68] D. F. Thiebaut and H. S. Stone. "Footprints in the cache." *ACM Transactions on Computer Systems*, **5**, no. 4, pp. 305-329, November 1987.

[69] D. F. Thiebaut, "On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio," *IEEE Transactions on Computers*, **C-38**, no. 7, pp. 1012-1026, July 1989.

[70] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest, "Trap-driven simulation with Tapeworm II," In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1994.

[71] D. 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.

[72] D. Wall, *Link-Time Code Modification*, Research Report 89/17, DEC Western Research Laboratory, Palo Alto, CA, Sept. 1989.

[73] D. Wall, "Systems for late code modification," In *Code Generation -- Concepts, Tools, Techniques*, Springer-Verlag, pp. 275-293, 1992.

[74] S. Weiss and J. Smith, *Power and the PowerPC*, San Mateo, CA: Morgan Kaufmann, 1994.

[75] T-Y Yeh, and Y. Patt, "Two-level adaptive training branch prediction," In *Proceedings of the 24th ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 51-61, Nov. 1991.