Optimizing the Cache Performance of Non-Numeric Applications

by

Chi-Keung Luk

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Dedication

To Mo-Kit Kwong, my beloved grandmother in memory.

Abstract

Optimizing the Cache Performance of Non-Numeric Applications

Chi-Keung Luk
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2000

The latency of accessing instructions and data from the memory subsystem is an increasingly crucial performance bottleneck in modern computer systems. While cache hierarchies are an important first step, they alone cannot solve the problem. Further, though a variety of latency-hiding techniques have been proposed, their success has been largely limited to regular, numeric applications. Few promising latency-hiding techniques that can handle irregular, non-numeric codes have been proposed, in spite of the popularity of such codes in computer applications.

This dissertation investigates hardware and software techniques for coping with the *instruction-access latency* and *data-access latency* in *non-numeric* applications. To deal with instruction-access latency, we propose *cooperative instruction prefetching*, a novel technique which significantly outperforms state-of-the-art instruction prefetching schemes by being able to prefetch more aggressively and much further ahead of time while at the same time substantially reducing the amount of useless prefetches.

To cope with data-access latency, we investigate three complementary techniques. First, we study how to use *compiler-inserted data prefetching* to tolerate the latency of accessing pointer-based data structures. To schedule prefetches early enough, we design three prefetching schemes to overcome the pointer-chasing problem associated with these data structures, and we automate them in an optimizing research compiler. Second, we study how to safely perform an important class of locality optimizations, namely *dynamic* *data layout optimizations*, in non-numeric codes. Specifically, we propose the use of an architectural mechanism called *memory forwarding* which can guarantee the safety of data relocation, thereby enabling many aggressive data layout optimizations (which also facilitate prefetching) that cannot be safely performed using current hardware or compiler technology. Finally, in an effort to minimize the overheads of latency tolerance techniques, we propose new cache miss prediction techniques based on *correlation profiling.* By correlating cache miss behaviors with dynamic execution contexts, these techniques can accurately isolate dynamic miss instances and so pay the latency tolerance overhead only when there would have been cache misses.

Detailed design considerations and experimental evaluations are provided for our proposed techniques, confirming them as viable solutions for coping with memory latency in non-numeric applications.

Key Words and Phrases: Cache performance, non-numeric applications, tolerating latency, instruction prefetching, data prefetching, locality optimizations, cache miss prediction.

Acknowledgments

My heart is filled with excitement at this moment, not only because I have finally finished a six-year-long project, but also because it is not until now, as I reflect over the last six years, that I realize how many people have contributed to this project in so many ways.

First of all, I must thank my family for their endless support throughout this study. My parents, Chiu-Ho Luk and Suet-Chun Chan, demonstrated their unconditional love by enduring my absence through these years and by sacrificing the living standard they have long deserved to be provided by their only son. My younger sister, Pui-Shan Luk, willingly played the role of her brother in the family and gave persistent care and encouragement over our weekly phone calls.

I am indebted to my advisor Prof. Todd C. Mowry for the tremendous time, energy, and intellect he put into my Ph.D. education. Todd taught me everything I need for doing high-quality research: how to choose and attack problems, how to write papers and give talks, and how to be a responsible part of the research community. When I compare myself to the person I was six years ago, there is no doubt how much he has influenced my abilities and attitudes. I wish that Todd will feel free to share in any success that I may have in the future.

I thank the members of my Ph.D. committee, Prof. Tarek S. Abdelrahman, Prof. Wen-Mei Hwu, Prof. Ken Sevcik, and Prof. Dave Wortman for their invaluable guidance and suggestions that have improved the completeness and readability of this dissertation. It also benefits from Edwin Chan, Yuan-Chun Chou, Robert O'Callahan, and Terri Palmer who gave insightful comments on its earlier drafts. All their efforts are greatly appreciated.

I have had the luxury of doing research at two great universities, University of Toronto and Carnegie Mellon University, where many people provided me their kind assistance. Many thanks to Kelly Chan and Kathy Yen at UofT and Sharon Burks and Karen Olack at CMU for their administrative help. Thanks also to Mark Stoodley for his help with Mable-driven simulations. In addition, I thank the members of the Stampede group for graciously sharing their computing resources.

Another luxury of my two-city study is the fellowship and teaching I received from both the Toronto Chinese Alliance Church (TCAC) and Pittsburgh Chinese Church (PCC). Special thanks go to Philip Wong, Yijen & Peter Wu, the Daniel Fellowship and Prayer Meeting members in TCAC, and others who together prayed for my graduation. I am grateful to Tammy Fung, Danny Lee, Vivian & Fish Ng, Tracy Yu, and many others in PCC for taking care of my transportation within Pittsburgh.

The friends whom I made in Toronto and Pittsburgh ensured that life as a foreign student was never lonely. Thank you Charles Chan, Chi-Lok Chan, Chakra Chennubhotla, Yuan-Chun Chou, Ronnie Fong, Cynthia Lee, Wai-Kau Lo, Jimmy Ma, Chung-Keung Poon, Jonathan Tong, Hai Wang, Mabel Wong, Mitsue & Tomoyuki Yamakami, and Pelton Yip. Your friendship will be with me all my life.

Last but not least, I thank my fiancée Juliana Wong and her family for their hearty care and encouragement. The love, stability, joy, and diversion that I received from her were vital to the quality and completion of this dissertation.

This work was financially supported by a Canadian Commonwealth Scholarship and a fellowship from the IBM Canada's Centre for Advanced Studies.

May this work be a glory to the Almighty One

Chi-Keung Luk
University of Toronto, December 1999

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The speed of microprocessors has been increased in the past decade due to ever-increasing clock rates and the exploitation of instruction-level parallelism—multiple-instruction-issue processors running at 1GHz clock rate are expected to be available by 2001 [47, 48]. However, the overall performance of a processor is not only determined by how fast computation can be performed but also by how rapidly *instructions* and *data* can be supplied by the memory subsystem. A speedy processor alone is not guaranteed to offer high overall performance unless it is matched by a similarly powerful memory subsystem.

Unfortunately, although microprocessor speed has been increasing dramatically, the speed of memory has not kept pace. As illustrated in Figure 1.1, the speed of commercial microprocessors has doubled roughly every three years while the speed of commodity DRAM has improved by only little more than 50% over the past decade. Part of the reason for this is that there is a direct tradeoff between capacity and speed in DRAM, and the highest priority in improving DRAM has been increasing capacity. The consequence is that a DRAM access typically costs a few tens to a hundred CPU cycles in modern microprocessors, which is potentially as expensive as executing several hundreds of instructions. Worse yet, this speed gap between the CPU and memory is expected to grow continually in the foreseeable future. In fact, some researchers have already suggested that the overall performance of microprocessor-based systems will ultimately be bounded by the performance of the memory subsystem, or by the so-called *memory wall* [132, 136].

To reduce effective memory access time, virtually all of today's microprocessor-based systems employ caches. While caches are the first critical step towards addressing the memory latency problem, they are not a complete solution. Recent studies [18, 54] have shown that caches are not as effective as expected because a significant fraction of cached data is not reused before it is displaced from the cache, and other studies [12, 26, 37, 86]

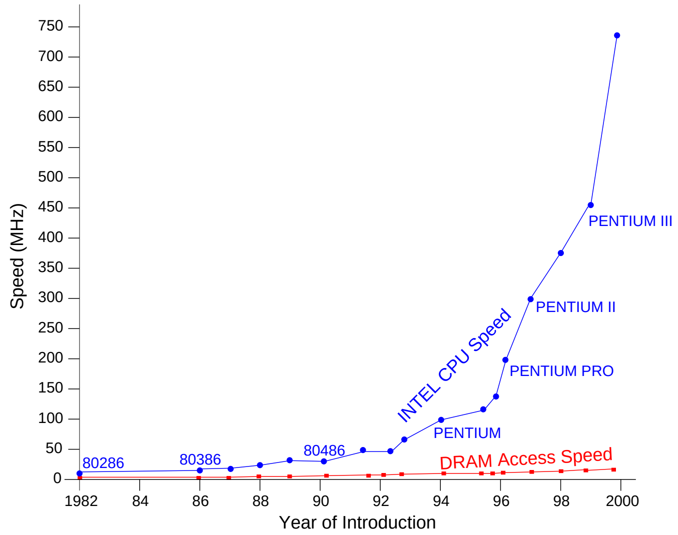


Figure 1.1: Speed of Intel microprocessors and commodity DRAM over the last 18 years.

show that cache miss penalty accounts for most of the stall time in many important applications. To improve processor utilization, techniques that reduce cache miss penalty are needed.

This dissertation investigates techniques for coping with the miss penalty of both instruction caches and data caches, thereby helping to unlock the full potential of microprocessor-based systems. In particular, we focus on a very important application domain: the class of codes known as *non-numeric*. The remainder of this chapter provides further motivations for improving the cache performance of non-numeric codes, discusses possible techniques for coping with memory latency, and presents our research goals. We conclude this chapter by listing the major contributions of this dissertation and giving an overview of the following chapters.

1.1 Cache Performance on Non-Numeric Codes

Applications are generally classified as *numeric* or *non-numeric*.[1] Numeric codes are characterized by their intensive use of floating-point data, regular loop nests, and multi-dimensional arrays. They are typically used in scientific and engineering applications. In contrast, non-numeric codes mostly operate on integer data, have irregular control

[1]There are some other common names for these two classes of applications, such as floating point vs. integer, scientific vs. general-purpose, regular vs. irregular, etc.

structures and complex data access patterns which usually involve pointer dereferences. Most commercial and personal computer applications, such as database, graphics, and spreadsheet programs, are non-numeric.

Most of the research effort on improving cache performance has concentrated on numeric codes. One reason for this focus is that numeric applications typically experience larger cache miss penalty than non-numeric applications do. Another reason is that non-numeric codes are significantly more difficult to optimize. Despite this difficulty, improving cache performance of non-numeric codes has become a very important problem because of the growing importance of non-numeric codes among today's computer applications and ever-increasing miss penalties.

The impact of cache miss penalties on the performance of non-numeric codes and commercial applications in particular has been investigated in a number of detailed performance studies [12, 37, 86]. The key finding that is common to all of these studies is that *both* instruction latency and data latency significantly degrade the performance of large commercial applications, regardless of the presence of reasonably large caches in the computer systems. For instance, a recent study conducted by Barroso *et al.* [12] has discovered that over 75% of the execution time of an online transaction processing application running on an AlphaServer 4100 (with four 300 MHz 21164 processors, each with 8KB on-chip instruction and data caches, a 96KB on-chip unified secondary cache, and a 2MB off-chip unified tertiary cache) is spent stalling for memory accesses, and those stall cycles are roughly evenly divided between instruction and data cache misses. In brief, these studies suggest that techniques are needed to reclaim the performance loss due to both instruction and data cache misses in those applications.

1.2 Techniques for Coping with Memory Latency

Techniques for coping with memory latency can be divided into two classes: those that *reduce* latency, and those that *tolerate* latency. Techniques for reducing latency include caching data[2] and improving cache effectiveness through locality optimizations. Techniques for tolerating latency include buffering and pipelining references, out-of-order execution, prefetching, and multithreading. In this section, we will briefly discuss each of these techniques to see how appropriate they are for non-numeric codes and to point out the potential challenges associated with each technique.

[2]Since both data and instructions can be cached, the term "data" here refers to instructions as well.

1.2.1 Caches

Caches reduce latency from a memory access to a cache access whenever data items are found in the cache. The likelihood of finding data in the cache depends on the cache geometry as well as the inherent *locality of reference* within the application. There are two kinds of locality: *temporal locality* is the tendency of a recently-accessed item to be accessed again soon, and *spatial locality* is the tendency of items near a recently-accessed item to be accessed soon. Since a reasonable amount of locality exists in most applications, caches are generally quite useful. For this reason, all the techniques we discuss in this section build upon caching as a foundation.

A brute-force approach to reducing cache miss penalty is to lower cache miss rates by increasing the size and associativity of caches. However, this does not appear to be a long-term solution for the following reasons: First, increasing cache size and associativity will lengthen cache access time (e.g., to provide large on-chip primary caches, the HP PA-8500 [52] processor has to increase the cache access time by an additional one to two cycles). Second, larger caches means higher hardware cost. Third, larger caches require more power, which is undesirable in power-constrained environments. Given all these disadvantages, a preferable approach to reducing latency would be to use caches as effectively as possible by better exploiting locality.

1.2.2 Locality Optimizations

Locality optimizations can significantly increase the effectiveness of both instruction and data caches. Let us begin with instruction locality optimizations.

Sequential instruction accesses enjoy spatial locality inherently. To enhance instruction locality across non-sequential accesses, on the other hand, the best-known approach is to lay out codes based *statically* on control-flow profiling information [43, 56, 87, 105, 129]. Doing so puts codes that are frequently executed together into adjacent memory, thereby improving spatial locality and reducing conflict misses. While such profile-guided code placement is a viable approach which requires little hardware support, it has the following limitations: First, an extra pass is needed to collect profiling information. Second, performance may not improve if the training data set does not truly reflect the actual data set. Third, in applications that do not have strongly-biased control flows, static layouts cannot mimic dynamic branching behaviors well.

Now we turn our attention to data locality optimizations. Data locality can potentially be improved by *restructuring computation*, by *reorganizing data layouts*, or by a combination of both. Figure 1.2 contains an example illustrating these two possible ap-

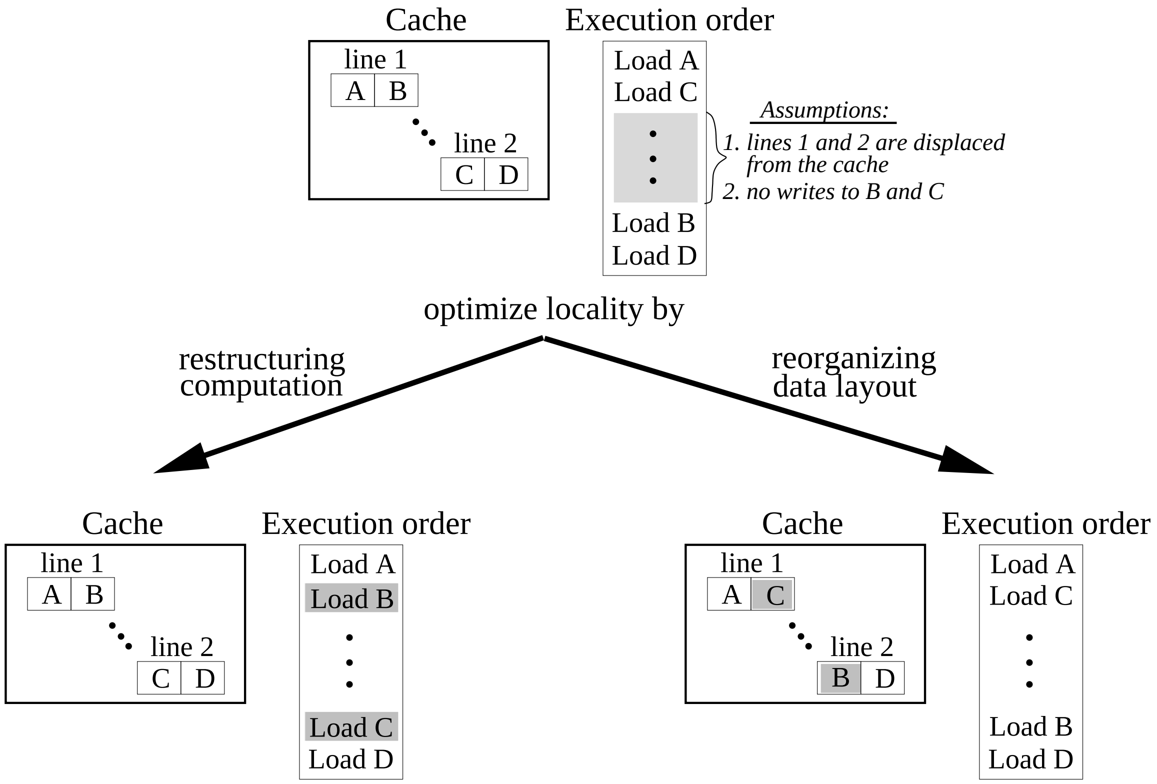


Figure 1.2: Example illustrating the two possible approaches to optimizing locality.

proaches. In this example, items `A` and `B` share the same cache line while `C` and `D` share another in the original data layout. Therefore, loading `A` and `C` at the beginning of the execution incurs two cache misses. Assume that both lines are then displaced from the cache before the loads of `B` and `D`. Then, these two loads will suffer another two cache misses. To improve locality, we can either switch `Load B` and `Load C` in the execution (assuming there are no writes to `B` and `C` in between these two loads) or switch `B` and `C` in the data layout. In both cases, there will be only two misses instead of four in the original case.

In either approach, the compiler must check a priori whether the optimization is always *legal*. If it is not certain that this is the case, the compiler must be conservative and give up the optimization. To restructure computation, the compiler needs to ensure that the order of references[3] to the same data is preserved after restructuring. To reorganize data layouts, the compiler must guarantee that all data references to the original layouts are adjusted to the new layouts after reorganization. Therefore, the key to both kinds of legality checks is to know exactly which data object a reference is pointing to.

[3]If there are only *reads* but no *writes*, the reference order does not need to be preserved to make the optimization legal.

Knowing this is relatively easy in *numerical* codes since most of the interesting references are usually direct array references, and hence fairly accurate legality tests such as dependency analysis [44, 85] have been developed for locality optimizations in these codes. Unfortunately, due to the presence of pointers (especially those that point to heap-allocated objects) in *non-numeric* codes, it is very difficult (if not impossible) for the compiler to figure out exactly which data object a reference is pointing to. This has seriously restricted compilers from applying many attractive locality optimizations to non-numeric applications. In particular, though different researchers have demonstrated the large performance potential of *dynamic* data layout optimizations such as *copying* [71, 127, 138] and *clustering* [31], being unable to check the legality has forced compilers to give up these optimization opportunities.

To cope with whatever latency that cannot be reduced through caching and locality optimizations, we need to consider techniques that *tolerate* memory latency.

1.2.3 Buffering and Pipelining References

Buffering and pipelining references is an approach which typically tolerates data latency but *not* instruction latency. The latency of *writing data* to the memory can be hidden by making use of *write buffers*. Write buffers exploit the fact the processor does not need to wait for a write to finish so long as it properly observes the future effect of the written data. Because of this, the processor can proceed immediately after a write by simply issuing the write to the write buffer, provided that future reads check the write buffer for matching addresses before bringing data from memory. An additional advantage of write buffers is that multiple writes can be overlapped to exploit pipelining.

Non-blocking loads [41] and *lockup-free caches* [69] are hardware-based techniques for tolerating latency of *reading data* through buffering and pipelining. In non-blocking loads, the processor stalls only when the data is *used* rather than when the data is loaded. A lockup-free cache, on the other hand, allows multiple outstanding cache misses. By combining these two techniques, it would be possible to buffer multiple reads and to pipeline their accesses. However, since in practice the use of a load value typically occurs shortly after the load is performed, these techniques could only hide a small fraction of relatively long cache miss latencies.

Pipelining is also applicable to instruction accesses. Nevertheless, since which instruction line needs to be fetched next depends on the instructions in the current line, subsequent instruction fetching will be stopped shortly after an instruction cache miss.

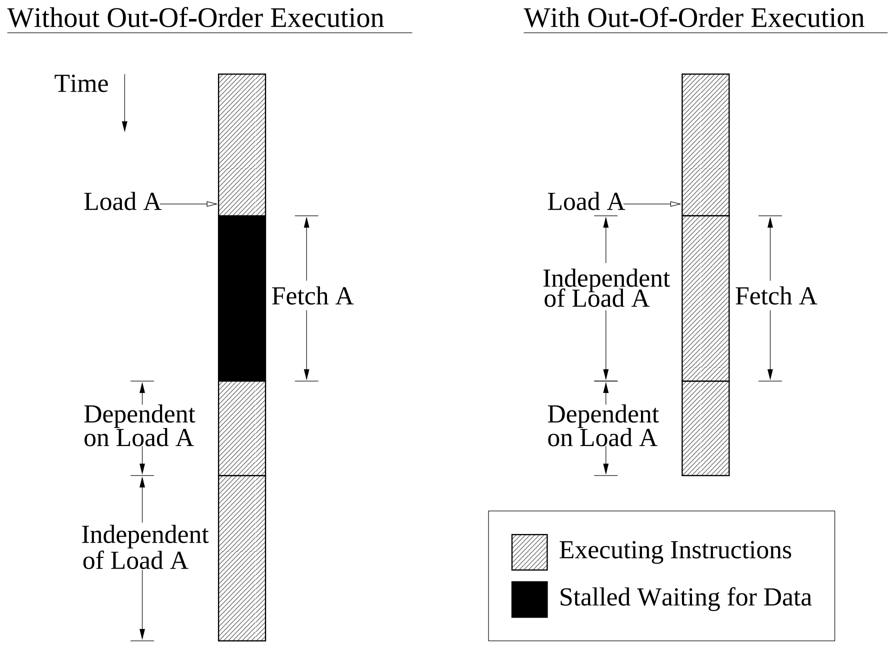


Figure 1.3: Illustration of how out-of-order execution tolerates memory latency.

Consequently, only very few instruction cache misses can be pipelined at a time.[4]

1.2.4 Out-Of-Order Execution

Many recent processors [55, 57, 74, 139] provide support for out-of-order execution so as to tolerate both pipeline and memory latencies. Rather than following the program order, the processor can execute any instructions within a hardware instruction-scheduling window that have all the input operands ready (assuming no resource conflicts). Using this capability, the processor does not have to stall at a data cache miss, provided that there are enough instructions that do not depend on the results of any uncompleted operations, including the memory access that suffers the cache miss. Figure 1.3 illustrates how read latency can be tolerated through out-of-order execution. In the case without out-of-order execution (on the left), the processor stalls while data item `A` is fetched. In contrast, with out-of-order execution, the processor can execute those instructions that are ready upon `Load A` in parallel with the fetch. Note that the decoupling of instruction fetching and execution in out-of-order machines also tolerates *instruction* latency to a certain extent by allowing the execution to continue upon an instruction cache miss.

Out-of-order execution is most beneficial to programs that have abundant *instruction-*

[4]Note that, however, more misses can be pipelined if some more aggressive instruction fetching mechanism such as instruction prefetching is used.

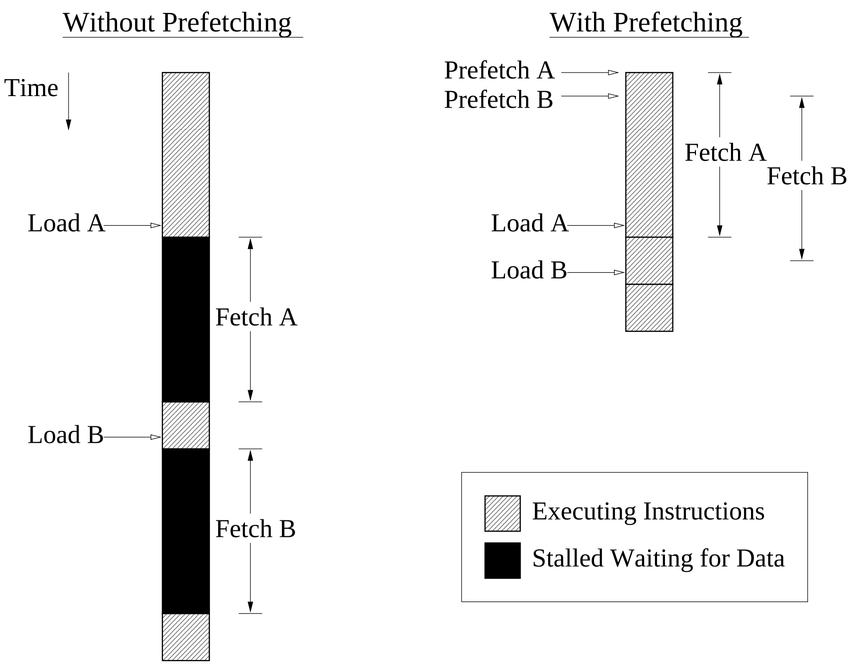


Figure 1.4: Illustration of how prefetching tolerates memory latency.

level parallelism (commonly referred to as ILP, the extent to which instructions in the *same* program thread can be executed in parallel). Unfortunately, this approach is unlikely to reduce much of the stall time caused by load misses, because loads are typically the first instructions of data dependence chains and hence many instructions in the scheduling window are in fact dependent on the loads missing in the cache. Similarly, the amount of ILP in many non-numeric programs is insufficient for tolerating much of the instruction miss latency. In addition, the hardware of the scheduling window and the dependence-tracking logic are fairly complex.

We have seen that both out-of-order execution and buffering/pipelining references are inadequate for tolerating read latency. Fortunately, the following two approaches, namely *prefetching* and *multithreading*, are more promising for tolerating read latency.

1.2.5 Prefetching

The key to tolerating read latency is to decouple the *request* for data from the *use* of that data, while finding enough useful *parallelism* to keep the processor busy in between. Prefetching tolerates latency by anticipating what data is needed and moving it to the cache ahead of time while executing other useful instructions within the *same* program thread. Figure 1.4 illustrates the way that prefetching tolerates read latency. Without prefetching, the processor stalls at the cache misses due to `Load A` and `Load B`. If the two prefetches are launched early enough before the loads, both `A` and `B` will be in the cache at the times that they are loaded. It is important to realize from this example that prefetching allows a memory access to be overlapped with computation as well as with other memory accesses (i.e. the accesses can be pipelined).

Unlike normal memory operations, prefetches are typically *non-binding* and *non-excepting*. With a *binding* prefetch, the data value is "bound" at the time the prefetch is executed by placing it in either a buffer or a register such that that is the actual value to be observed by a subsequent load. The problem with a binding prefetch is that if the data is modified during the interval between when the prefetch is issued and the bound value is used, the value delivered will be stale. On the contrary, with a *non-binding* prefetch, the data value is not bound until it is referenced by a subsequent load. On the other hand, a *non-excepting* prefetch will never generate exceptions even if the data address is invalid. When a prefetch is both non-binding and non-excepting, it can be issued speculatively without worrying about the validity of the data address and value.

Prefetching is applicable to both instructions and data and can be controlled by hardware or software. Hardware-controlled prefetching does not have any instruction overhead but is less flexible when prefetching complicated access patterns. In contrast, software-controlled prefetching is more flexible in scheduling prefetches at the expense of the associated instruction overhead. Below, we briefly discuss the existing work on instruction prefetching and on data prefetching.

Almost all of the existing instruction prefetching schemes are purely hardware controlled [60, 106, 115, 116, 118]. To exploit the inherent spatial locality of instruction streams, most of these schemes let the hardware prefetch a few sequential cache lines. To prefetch non-sequential instruction accesses, some schemes use history information to predict which non-sequential line should be prefetched next, while some prefetch the targets of conditional branches and procedure calls. However, as revealed by the detailed performance analysis of these schemes reported in Chapter 2, none of these existing schemes can schedule prefetches early enough to fully tolerate the miss latency in fast modern processors. Another potential problem with instruction prefetching is *cache pollution*. If too many sequential lines or jump targets are prefetched, they may displace some instructions that will be used soon from the cache.

On the data side, sophisticated hardware-controlled [28] and software-controlled [96] data prefetching schemes have been developed for numeric codes. These schemes rely on the fact that addresses of future data references are highly predictable (since most of them are direct array references). In contrast, prefetching non-numeric codes is significantly more difficult, and so research in this area has so far been lacking. The major challenge

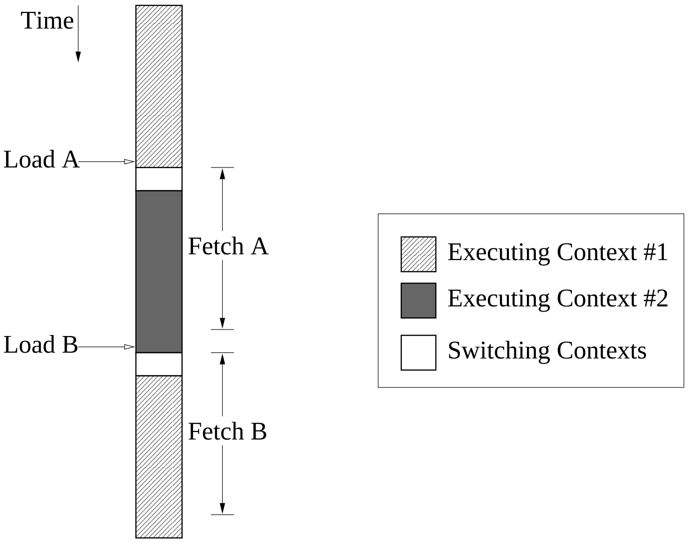


Figure 1.5: Illustration of how multithreading tolerates memory latency.

is, just as in performing locality optimizations on non-numeric codes, to anticipate the apparently chaotic addresses of future data references. However, with sophisticated compiler analyses (especially those that understand pointers), the compiler may be able to predict these complicated data addresses to a useful extent. It is not clear that hardware can achieve such prediction. An additional advantage of software-controlled prefetching is that since a number of recent processors [13, 112, 139] have already provided support for data-prefetch instructions, no extra hardware support is needed as in the case of hardware-controlled prefetching. Therefore, software-controlled data prefetching may be a feasible technique for coping with data latency in non-numeric codes, despite the need for minimizing its instruction overhead.

1.2.6 Multithreading

Similar to out-of-order execution and prefetching, multithreading tolerates read latency by executing some useful instructions between the times that the data is requested and when it is used. The most important difference is that in multithreading those overlapped useful instructions are obtained by switching the context to another thread. Figure 1.5 illustrates how multithreading tolerates read latency through context switching. When `Load A` misses in the cache, the processor switches to context #2 and continues executing that context until another cache miss occurs (i.e. `Load B`). Then, it switches back to context #1. Hopefully `Load A` has already finished by that time and so the processor will not be stalled.

Traditionally, multithreading is purely hardware-based [4, 49, 68, 73]. It can be used to hide both pipeline and memory latencies, including instruction latency and data latency. Compared to prefetching, multithreading has two main advantages: First, unlike prefetching, it does not require the ability to predict addresses in advance, which is a very challenging problem in data prefetching for non-numeric codes. Second, since it, unlike software-controlled prefetching, does not require any software support it can improve the speed of existing executables without recompilation.

However, multithreading does have the following significant drawbacks. First of all, it requires multiple concurrent threads. This additional concurrency may or may not exist, particularly in a uniprocessor environment. Automatic parallelization of non-numeric applications is still inefficient in most cases, and it is unlikely that programmers would go through the pain of hand-parallelizing their applications just for the sake of multithreading. A second drawback is the overhead of context-switching, which occurs in two forms: (i) pipeline bubbles caused by flushing the pipeline after the detection of a cache miss, and (ii) the additional time needed to save and restore context state (e.g., the register file). These switching overheads can potentially offset much of the performance gain of multithreading. Finally, multithreading requires a significant amount of hardware support to minimize context-switching overhead (e.g., replicated register files). Clearly, multithreading is a more expensive solution than prefetching, in terms of the demands on both concurrency and hardware.

1.2.7 Overall Strategy

Table 1.1 is a summary of the techniques we have just discussed. For each technique, we list the major challenges in applying the technique to cope with instruction latency and data latency in *non-numeric codes*.

After looking at the individual techniques, we can now devise an overall strategy of coping with memory latency in non-numeric codes. Obviously, caching is very important on its own and is the primary latency-hiding mechanism in most processors nowadays. Also, buffering, pipelining, and out-of-order execution are already used in many recent processors as additional means to hide latency. To cope with whatever latency that is not hidden by these techniques (read latency in most cases), we can consider multithreading, locality optimization, and prefetching. Multithreading is attractive in theory, but the long-standing challenge in automatically finding thread-level parallelism in non-numeric codes suggests that we consider multithreading as the last resort. Instead, we will concentrate on locality optimizations and prefetching.

Table 1.1: Techniques for coping with memory latency.

<table>
<tr><th rowspan="2">Technique</th><th rowspan="2">Benefit</th><th rowspan="2">Exploits</th><th colspan="2">Challenges in Coping with</th></tr>
<tr><th>Instruction Latency</th><th>Data Latency</th></tr>
<tr><td>Caching</td><td>Reduces Latency</td><td>Locality of Reference</td><td colspan="2">• Insufficient Reuse
• Tradeoff Between Size and Access Time</td></tr>
<tr><td rowspan="2">Locality Optimizations</td><td rowspan="2">Reduces Latency</td><td rowspan="2">Reordering Computation or Reorganizing Data/Instructions to Enhance Locality</td><td>(Legality Is Not a Problem)</td><td>• Legality Test</td></tr>
<tr><td colspan="2">• Need Profiling Information
• Sensitivity to Profile Quality</td></tr>
<tr><td>Buffering and Pipelining</td><td>Tolerates Latency</td><td>Non-Blocking Memory Accesses</td><td>Instructions Depending on Control Flow</td><td>Data Depending on Load Results</td></tr>
<tr><td>Out-of-Order Execution</td><td>Tolerates Latency</td><td>Instruction Level Parallelism</td><td colspan="2">• Lack of Independent Instructions to Tolerate Read and Instruction Latency
• Complex Hardware Support</td></tr>
<tr><td rowspan="2">Prefetching</td><td rowspan="2">Tolerates Latency</td><td rowspan="2">Parallelism within a Single Thread</td><td colspan="2">• Scheduling Prefetches Early Enough</td></tr>
<tr><td>• Cache Pollution</td><td>(Cache Pollution Is Less Likely a Problem)</td></tr>
<tr><td>Multithreading</td><td>Tolerates Latency</td><td>Parallelism across Multiple Threads</td><td colspan="2">• Finding Concurrency Automatically
• Possible Negative Impact on Single-Thread Performance</td></tr>
</table>

On the instruction side, locality optimizations are desirable if the user can afford an extra profiling pass and the training input is representative of the actual one. Otherwise, instruction prefetching may be preferable, provided that we can find some way to launch prefetches early enough while avoiding cache pollution. On the data side, both locality optimizations and prefetching face the same difficult problem of knowing future data addresses. However, since prefetching accepts inexact knowledge of these addresses (this will not corrupt the program since prefetches are both non-binding and non-excepting), prefetching is likely to be more widely applicable than locality optimizations. And as we have already discussed, software-controlled prefetching appears to be more promising than hardware-controlled prefetching for these types of codes because the compiler may be more capable of predicting the data addresses. Finally, to reduce both latency and bandwidth consumption, we wish to have locality optimizations as well provided that they are safe to apply.

1.3 Research Goals

The goal of this dissertation is to help improve the cache performance of non-numeric codes. Since this is a largely unexplored area and is so broad by itself, solving the whole problem in a single dissertation is an overly ambitious goal. Instead, we will focus on a few of the more important problems.

Since both instruction latency and data latency are important, we will address both. For instruction latency, despite both locality optimizations and prefetching being important, we concentrate on instruction prefetching. The reason for this choice is that techniques for optimizing instruction locality are relatively mature. In fact, many industrial compilers have already provided such tools. In contrast, existing instruction prefetching techniques do not appear to be adequate for fast modern processors and hence a new instruction prefetching technique may be warranted. The key to the success of this new technique is to launch prefetches early enough while avoiding cache pollution. Also, to avoid placing the burden on the programmer, instruction-prefetch scheduling should be done automatically, either by the hardware or the compiler.

For data latency, since access patterns in non-numeric codes can be very unpredictable, we will focus primarily on an important class of non-numeric codes: those that make intensive use of pointer-based data structures such as linked lists and trees. First, we wish to devise compiler-based prefetching algorithms for these pointer-based codes. Our goal here is to have the compiler automatically predict data addresses and insert prefetches into these codes. Next, if software-controlled prefetching can offer significant performance benefits, we would like to see if these benefits can be further increased by reducing prefetching overheads. Our approach is to devise a general technique that can reduce the overheads of any latency tolerance techniques, including but not limited to prefetching. This technique should accurately predict when cache misses happen so that a latency tolerance mechanism will be invoked only upon cache misses (but not hits) to minimize the overhead. Finally, we wish to provide a technique that can help exploit the large performance potential of dynamic data layout optimizations by *automatically* guaranteeing their legality in non-numeric programs.

1.4 Contributions of Dissertation

This dissertation makes the following primary contributions:

- The design and evaluation of an effective and fully-automatic instruction prefetching technique called *cooperative prefetching* [80, 81], whereby the hardware and compiler cooperate to hide the latency. Cooperative prefetching comprises two novel components: *compiler-inserted instruction prefetching* and an intelligent hardware *prefetch filter* for eliminating useless prefetches. The result of their cooperation is that, unlike previous instruction prefetching techniques, we are able to prefetch instructions far enough in advance without polluting the cache. Detailed experimental results demonstrate that cooperative prefetching hides 50% or more of the latency remaining with the best previous instruction prefetching techniques, while at the same time reduces the number of useless prefetches by a factor of six. Consequently, cooperative prefetching significantly improves the overall performance of all the applications we examined.

- The design and evaluation of three *compiler-based data prefetching* schemes for tolerating the latency of accessing pointer-based data structures [79, 82]. Our schemes are to date the only compiler-based techniques designed for overcoming the *pointer-chasing problem* in prefetching these data structures. They are implemented in a widely used research compiler and evaluated using detailed simulations of a dynamically-scheduled processor. Experimental results demonstrate that automatic compiler-inserted prefetching significantly improves the execution speed of pointer-based codes—by as much as more than twofold. While one of these prefetching schemes is more widely applicable, the other two can offer larger performance benefits and handle very large miss latencies by predicting the traversals of the data structures. In addition, we show that our schemes are fairly robust with respect to a number of key architectural parameters.

- The proposal and evaluation of an architectural technique called *memory forwarding* [83] which can guarantee the legality of *any* locality optimizations based on data layout reorganization. Our technique is the only one to date that can provide such a correctness guarantee, and thereby substantially help unlock the full potential of locality optimizations in non-numeric codes. We suggest a number of potential applications of memory forwarding, and address some important issues of implementing it in modern processors. Experimental results show that the aggressive locality optimizations enabled by memory forwarding greatly reduce both memory latencies and memory bandwidth consumption in a set of non-numeric applications, and hence offer significant speedups: in some cases by more than twofold. In addition, we demonstrate that these optimizations improve the effectiveness of prefetching.

- The proposal and evaluation of a cache-miss prediction technique named *correlation profiling* [97, 98] for reducing the run-time overheads of latency tolerance techniques. Correlation profiling offers significantly higher cache-miss prediction accuracy than that achieved by state-of-the-art prediction techniques by exploiting the correlation between dynamic execution contexts and cache miss behaviors. By activating latency tolerance mechanisms only in dynamic instances of a static memory reference that will miss in the cache, much of the unnecessary overheads can be saved. Experimental results show that roughly half of the 21 non-numeric applications studied can potentially enjoy significant reductions in memory stall time by exploiting correlation profiling. Moreover, we use correlation profiling to improve performance of dynamic instruction scheduling and software-controlled prefetching.

1.5 Organization of Dissertation

This dissertation comprises the following five chapters in addition to this introductory chapter:

Chapter 2 describes cooperative instruction prefetching. We start with a quantitative evaluation of state-of-the-art instruction prefetching techniques. After identifying the fundamental problem with existing techniques, we propose cooperative prefetching as a solution and explain how it can prefetch much further ahead without polluting the cache. Finally, we evaluate in detail the effectiveness of cooperative prefetching in a modern processor.

Chapter 3 studies compiler-based data prefetching schemes for pointer-based data structures. First, we identify the fundamental problem in prefetching these data structures. Second, we address this problem by devising three new prefetching schemes, and we also describe an implementation of these schemes. Finally, we present an experimental evaluation of them as well as a quantitative comparison with another compiler-inserted pointer prefetching technique.

Chapter 4 explores the use of memory forwarding as a technique to facilitate data layout optimizations. We first describe a similar form of memory forwarding that appeared in the past. We then propose how memory forwarding can be adapted to improve cache performance in modern processors, and address some implementation issues. Lastly, we evaluate the potential performance benefits of memory forwarding by using it to enable a number of aggressive optimizations whose legality cannot be guaranteed by existing hardware or compiler technology.

Chapter 5 investigates the use of correlation profiling to predict data cache misses. We begin by introducing some basic concepts and then propose three forms of correlation. Next we present a qualitative analysis of expected benefits, followed by a quantitative evaluation. To understand when and why correlation profiling succeeds, we perform detailed case studies of individual applications. Finally, we demonstrate the practicality of correlation profiling by applying it to two latency tolerance techniques.

Finally, Chapter 6 includes a summary of the primary results in this dissertation and a number of directions for future work in this area. Appendix A discusses the overall experimental methodology used in this dissertation.

Chapter 2

Cooperative Instruction Prefetching

2.1 Introduction

The rate of instruction supply is crucial to the performance of computer systems, and is significantly determined by the latency of instruction accesses. While instruction caches are helpful for reducing this latency, they are not a complete solution, especially in many non-numeric (commercial) applications that have large instruction footprints and poor instruction localities. For example, a study conducted by Maynard *et al.* [86] demonstrates that many commercial applications suffer from relatively large instruction cache miss rates (e.g., over 20% in an 8KB cache). To further tolerate instruction cache miss latency, one attractive approach is to automatically *prefetch* instructions into the cache before they are needed.

2.1.1 Previous Work on Instruction Prefetching

Several researchers have considered instruction prefetching in the past. We will begin by discussing and then quantitatively evaluating four of the most promising techniques that have been proposed to date, all of which are purely hardware-based: *next-N-line* prefetching [115, 116], *target-line* prefetching [118], *wrong-path* prefetching [106], and *Markov* prefetching [60].

Before we begin our discussion, we briefly introduce some prefetching terminology. The *coverage factor* is the fraction of original cache misses that are prefetched. A prefetch is *unnecessary* if the line is already in the cache (or is currently being fetched), and is *useless* if it brings a line into the cache which will not be used before it is displaced. An ideal prefetching scheme would provide a coverage factor of 100% and would generate no unnecessary or useless prefetches. In addition, the *timeliness* of prefetches is also crucial. The *prefetching distance* (i.e. the elapsed time between initiating and consuming the result of a prefetch) should be large enough to fully hide the miss latency, but not so large that the line is likely to be displaced by other accesses before it can be used (i.e. a useless prefetch).

The idea behind *next-N-line prefetching* [115, 116] is to prefetch the N sequential lines following the one currently being fetched by the CPU. A larger value of N tends to increase the prefetching distance, but also increases the likelihood of polluting the cache with useless prefetches. The optimal value of N depends on the line size, the cache size, and the behavior of the application itself. Next-N-line prefetching captures sequential execution as well as control transfers where the target falls within the next N lines. It is usually included as part of other more complex instruction prefetching schemes, and based on our experiments, it accounts for most of the performance benefit of these previously existing schemes.

To further expand the scope of prefetching to capture more control transfer targets, Smith and Hsu [118] proposed *target-line prefetching* which uses a prediction table to record the address of the line which most recently followed a given instruction line, thus enabling hardware to prefetch targets whenever an entry is found in this table. They observed that combining target-line prefetching with next-1-line prefetching produced significantly better results than either technique alone.

Rather than relying on history tables, Pierce and Mudge [106] proposed *wrong-path prefetching* which combines next-N-line prefetching with always prefetching the target of control transfers with static target addresses. Hence for conditional branches, both the target and fall-through lines will always be prefetched. However, since target addresses cannot be determined early, this scheme only outperforms next-N-line prefetching when a conditional branch is initially untaken but later taken (assuming that enough time has passed to hide the latency but not so much that the line has been displaced). Their results indicated that wrong-path prefetching performed slightly better than next-1-line prefetching on average.

Joseph and Grunwald [60] proposed *Markov prefetching*, which correlates consecutive miss addresses. These correlations are stored in a *miss-address prediction table* which is indexed using the current miss address, and which can return multiple predicted addresses. The Joseph and Grunwald study focused primarily on data cache misses, and did not compare Markov prefetching with techniques designed specifically for prefetching instructions.

Finally, we note that while Xia and Torrellas [137] considered instruction prefetching for codes where the layout has already been optimized using profiling information, we focus only on techniques which do not require changes to the instruction layout in this

Table 2.1: Parameters used in the evaluation of existing instruction prefetching techniques.

Technique	# of Sequential Lines Prefetched	Target Prefetching Parameters		
		# of Targets	Table Size	Table Indexing Method
Next-N-Line	$N = 2, 4, 8$	0	0	N/A
Target-Line	2	1	64 entries	direct-mapped with tags
Wrong-Path	2	1	0	N/A
Markov	2	2	512 KB	direct-mapped with tags

study.

Performance of Existing Instruction Prefetching Techniques

To quantify the performance benefits and limitations of the four prefetching techniques described above, we implemented each of them within a detailed, cycle-by-cycle simulator which models an out-of-order four-issue superscalar processor based on the MIPS R10000 [139]. We model a two-level cache hierarchy with split 32 KB, two-way set-associative primary instruction and data caches and a unified 1 MB, four-way set-associative secondary cache. Both levels use 32 byte lines. The penalty of a primary cache miss that hits in the secondary cache is at least 12 cycles, and the total penalty of a miss that goes all the way to memory is at least 75 cycles (plus any delays due to contention, which is modeled in detail). To provide better support for instruction prefetching, we further enhanced the primary instruction cache relative to the R10000 as follows: we divide it into four separate banks, and we add an eight-entry victim cache [61] and a 16-entry prefetch buffer [60]. Further details on our experiments will be presented later in Section 2.5.

Table 2.1 summarizes the prefetching parameters used throughout our experiments. These parameters were chosen through experimentation in an effort to maximize the performance of each scheme. All schemes effectively include next-2-line prefetching. (Although next-2-line prefetching was not in the original Markov prefetching design [60], we added it since we found that it improves performance.) When a target is to be prefetched, we prefetch two consecutive lines starting at the target address.

Figure 2.1 shows the performance impact of each prefetching scheme on a collection of seven non-numeric applications (which are discussed more in Section 2.5). We show three different versions of next-N-line prefetching (where $N = 2$, 4, and 8) in Figure 2.1, along with the original case without prefetching (**O**) and the case with a perfect instruction cache (**P**). Each bar represents execution time normalized to the case without prefetching,

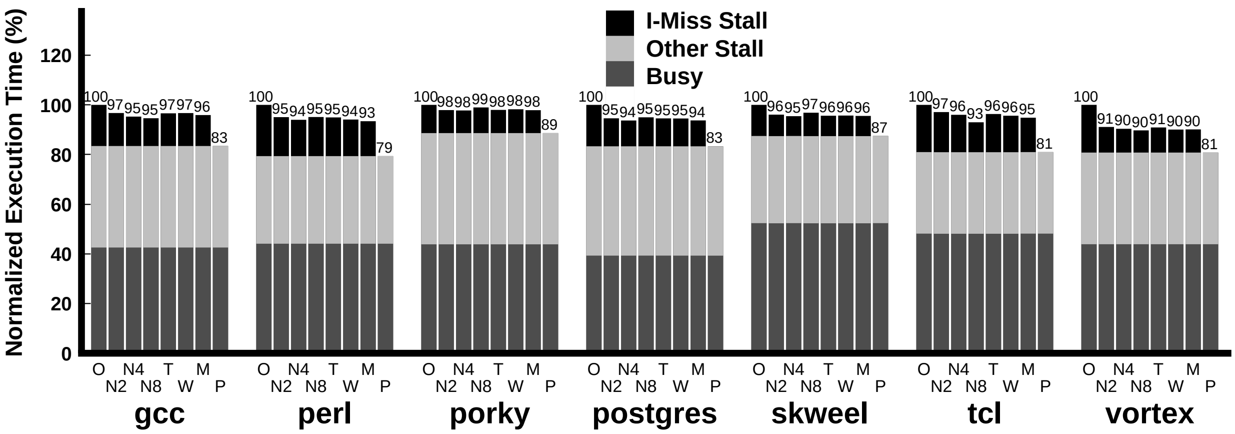


Figure 2.1: Performance of existing instruction prefetching techniques (**O** = original, **N**x = next-x-line prefetching, **T** = target-line prefetching, **W** = wrong-path prefetching, **M** = Markov prefetching, **P** = perfect instruction cache).

and is broken down into three categories corresponding to all potential graduation slots.[1] The bottom section (*Busy*) is the number of slots when instructions actually graduate, the top section (*I-Miss Stall*) is any non-graduating slots that would not occur with a perfect instruction cache, and the middle section (*Other Stall*) is all other slots where instructions do not graduate.

We observe from Figure 2.1 that despite significant differences in complexity and hardware cost, the various prefetching schemes offer remarkably similar performance, with no single scheme clearly dominating. Perhaps surprisingly, the best performance is achieved by either next-4-line or next-8-line prefetching in all cases except `perl`; even in `perl`, next-4-line prefetching is still within 1% of the best case. The reason for this is that the bulk of the benefit offered by each of these schemes is due to prefetching sequential accesses.

Finally, we see in Figure 2.1 that these schemes are hiding no more than half of the stall time due to instruction cache misses. Through a detailed analysis of why these schemes are not more successful (further details are presented later in Section 2.6.1), we observe that although the coverage is generally quite high, the real problem is the *timeliness* of the prefetches—i.e. prefetches are not being launched early enough to hide the latency. Hence there is significant room for improvement over these existing schemes.

[1] The number of graduation slots is the issue width (4 in this case) multiplied by the number of cycles. We focus on graduation rather than issue slots to avoid counting speculative operations that are squashed.

2.1.2 Our Solution

To hide instruction cache miss latency more effectively in modern microprocessors, we propose and evaluate a new fully-automatic instruction prefetching scheme whereby the compiler and the hardware cooperate to launch prefetches earlier (therefore hiding more latency) while at the same time maintaining high coverage and actually *reducing* the impact of useless prefetches relative to today's schemes. Our approach involves two novel components. First, to enable more aggressive sequential prefetching without polluting the cache with useless prefetches, we introduce a new *prefetch filtering* hardware mechanism. Second, to enable more effective prefetching of non-sequential accesses, we introduce a novel compiler algorithm which inserts explicit *instruction-prefetch instructions* into the executable to prefetch the targets of control transfers far enough in advance. Our experimental results demonstrate that our scheme provides significant performance improvements over existing schemes, eliminating roughly 50% or more of the latency that had remained with the best existing scheme.

This chapter is organized as follows. We begin in Section 2.2 with an overview of our approach, and then present further details on the architectural and compiler support in Sections 2.3 and 2.4. Sections 2.5 and 2.6 present our experimental methodology and our experimental results, and finally we conclude in Section 2.7.

2.2 Cooperative Instruction Prefetching

We begin this section with a high-level overview of our prefetching scheme. To make our approach concrete, we also present an example illustrating prefetch insertion.

2.2.1 Overview of the Prefetching Algorithm

As we mentioned earlier, the key challenge in designing a better instruction prefetching scheme is to be able to launch prefetches earlier—i.e. to achieve a larger *prefetching distance*. Let us consider the sequential and non-sequential portions of instruction streams separately.

Prefetching Sequential Accesses

Since the addresses within sequential access patterns are trivial to predict, they are well-suited to a purely hardware-based mechanism such as next-N-line prefetching. To get far enough ahead to fully hide the latency, we would like to choose a fairly large value for N (e.g., $N = 8$ in our experiments). However, the problem with this is that larger values of N increase the probability of overshooting the end of the sequence and polluting the cache with useless prefetches. For example, next-8-line prefetching performs worse than next-4-line prefetching for four cases in Figure 2.1 (`perl`, `porky`, `postgres`, and `skweel`) due to this effect.

The ideal solution would be to prefetch ahead aggressively (i.e. with a large N) but to stop upon reaching the end of the sequence. Xia and Torrellas [137] proposed a mechanism for doing this which uses software to explicitly mark the likely end of a sequence with a special bit. In contrast, we achieve a similar effect using a more general *prefetch filtering* mechanism which automatically detects and discards useless prefetches before they can pollute the instruction cache[2]. We will explain how the prefetch filter works in detail later in Section 2.3.3, but the basic idea is to use two-bit saturating counters stored in the secondary cache tags to dynamically detect cases where lines have been repeatedly prefetched into the primary instruction cache but were not accessed before they were displaced (i.e. *useless* prefetches). When prefetches for such lines subsequently arrive at the secondary cache, they are simply dropped. One advantage of our approach is that it adapts to the dynamic branching behavior of the program, rather than relying on static predictions of likely control flow paths. In addition, our filtering mechanism is equally applicable to *non-sequential* as well as sequential prefetches.

Prefetching Non-Sequential Accesses

In contrast with sequential access patterns, purely hardware-based prefetching schemes are far less successful at prefetching *non-sequential* instruction accesses early enough. Wrong-path prefetching does not attempt to predict the target address of a given branch early, but instead hopes that the same branch will be revisited sometime in the not-too-distant future with a different branch outcome. Both target-line and Markov prefetching rely on building up history tables to predict addresses to prefetch along control targets. However, if a control transfer is encountered for the first time or if its entry has been displaced from the finite history table, then its target will not be prefetched.[3] Perhaps more importantly, even if a valid entry is found in the history table, it is often too late to fully hide the latency of prefetching the target since the processor is already accessing the line containing the branch.

[2]Interestingly, our prefetch filter works in a similar fashion as the hardware mechanism recently proposed by Johnson *et al.* [59] for controlling the fetch size upon data cache misses according to the detected spatial locality.

[3]Note that although our prefetch filtering mechanism can also potentially suffer from the limitations of learning within a finite table, we find that it is far more important to prefetch target addresses early enough rather than filtering out all useless prefetches.

To overcome these limitations, we rely on *software* rather than hardware to launch non-sequential instruction prefetches early enough. To avoid placing any burden on the programmer, we use the compiler to insert these new instruction-prefetching instructions automatically. As we describe in further detail later in Section 2.4, our compiler algorithm moves prefetches back by a specified prefetching distance while being careful not to insert prefetches that would be redundant with either next-N-line prefetching or other software instruction prefetches. Since many control transfers within procedures have targets within the N lines covered by our next-N-line prefetcher, the bulk of the instructions inserted by our compiler algorithm are for prefetching *across* procedure boundaries (as we show later in Section 2.5). Hence, although it is an oversimplification, one could think of our scheme as being primarily hardware-based for *intraprocedural* prefetching, and primarily software-based for *interprocedural* prefetching.

While direct control transfers (i.e. ones where the target address is statically known) are handled in a straightforward way by our algorithm, *indirect jumps* require some additional support in order for software to generate the target addresses early. We consider two separate cases of indirect jumps: procedure returns, and all other indirect jumps. Since procedure return addresses can be easily predicted through the use of a *return address stack* [62], we simply use a special prefetch instruction which implicitly uses the top of the return address stack as its argument.[4] To predict the target addresses of other indirect jumps, we use a hardware structure called an *indirect-target table* which records past target addresses of individual indirect jump instructions, and which is indexed using the instruction addresses of indirect jumps themselves. A prefetch instruction designed to prefetch the target of an indirect jump i conceptually stores the instruction address of i, which is then used to index the indirect-target table to retrieve the actual target addresses to prefetch. (Note that an indirect-target table is considerably smaller than the tables used by either target-line or Markov prefetching since it only contains entries for active indirect jumps other than procedure returns.)

While the advantage of software-controlled instruction prefetching is that it gives us greater control over issuing prefetches early, the potential drawbacks are that it increases the code size and effectively reduces the instruction fetch bandwidth (since the prefetch instructions themselves consume part of the instruction stream). Fortunately, our experimental results demonstrate that this advantage outweighs any disadvantages.

[4]Although one could also imagine using the return address register as an explicit argument to the prefetch instruction, this may complicate the processor by creating a new datapath from the register file to the instruction fetcher. In general, we would like to avoid instruction-prefetch instructions which have register arguments.

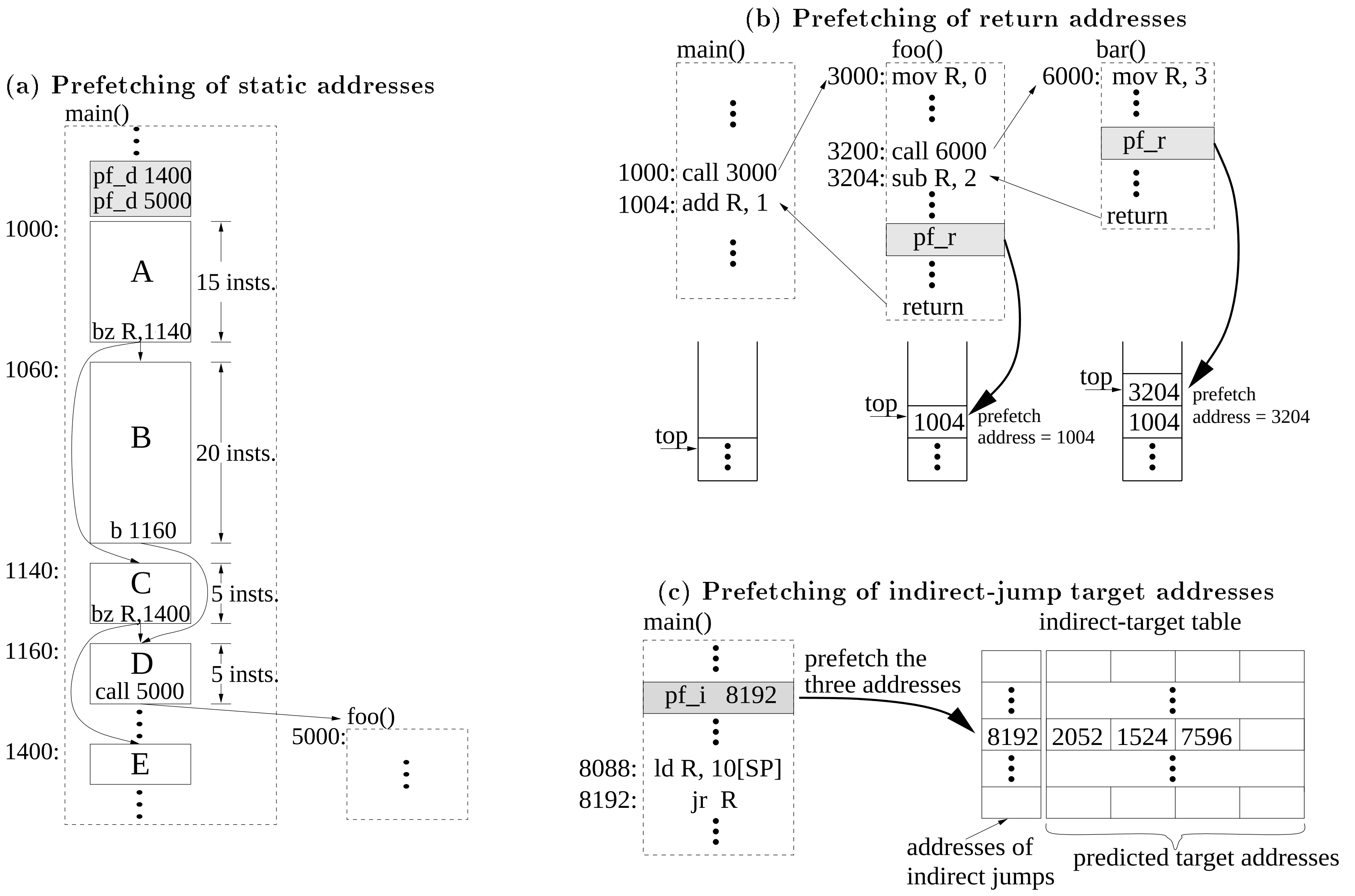


Figure 2.2: Examples of prefetch insertion for different types of target addresses (`pf_d` = prefetch a direct address, `pf_r` = prefetch a return address, `pf_i` = prefetch an indirect-jump target address).

2.2.2 Example of Prefetch Insertion

To make our discussion more concrete, Figure 2.2 contains three examples of how different types of prefetches are inserted. We assume the following in these examples: a cache line is 32 bytes long; an instruction is four bytes long (hence one cache line contains eight instructions); hardware next-8 line prefetching is enabled; and the prefetching distance is 20 instructions.

Figure 2.2(a) shows two procedures, `main()` and `foo()`, where `main()` contains five basic blocks (labeled `A` through `E`). Two prefetches have been inserted at the beginning of basic block `A`: one targeting block `E`, and the other targeting procedure `foo()`. There is no need to insert software prefetches for blocks `B`, `C` or `D` at `A` since they will already be handled by next-8-line prefetching. The prefetch targeting `E` is inserted in block `A` rather than in block `C` in order to guarantee a prefetching distance of at least 20 instructions. Although there are two possible paths from `A` to `foo()` (i.e. `A`→`B`→`D`→`foo()` and `A`→`C`→`D`→`foo()`), the compiler inserts only a single prefetch of `foo()` in `A` (rather than inserting one in `A` and one in `B`) because (i) `A` *dominates*[5] both paths, and (ii) the compiler determines that these prefetched instructions are not likely to be displaced by other instructions fetched along the path `A`→`B`→`D`→`foo()`.

Figure 2.2(b) shows an example of prefetching return addresses. The prefetches in procedures `bar()` and `foo()` get their addresses from the top of the return address stack—i.e. `3204` and `1004`, respectively. Finally, Figure 2.2(c) shows an example where a prefetch is inserted to prefetch the target address of the indirect jump at address `8192` before the actual target address is known (i.e. the value register R has not been determined yet). Hence the prefetch has `8192` as its address operand to serve as an index into the indirect-target table. Three target addresses are predicted for this indirect jump, and all of them will be prefetched.

2.3 Architectural Support

Our prefetching scheme requires new architectural support. In this section, we describe our extensions to the instruction set, how these new instructions affect the pipeline, and the new hardware that we add to the memory system (including the prefetch filter).

[5] Node d of a flow graph dominates node n if every path from the initial node of the flow graph to n goes through d [6].

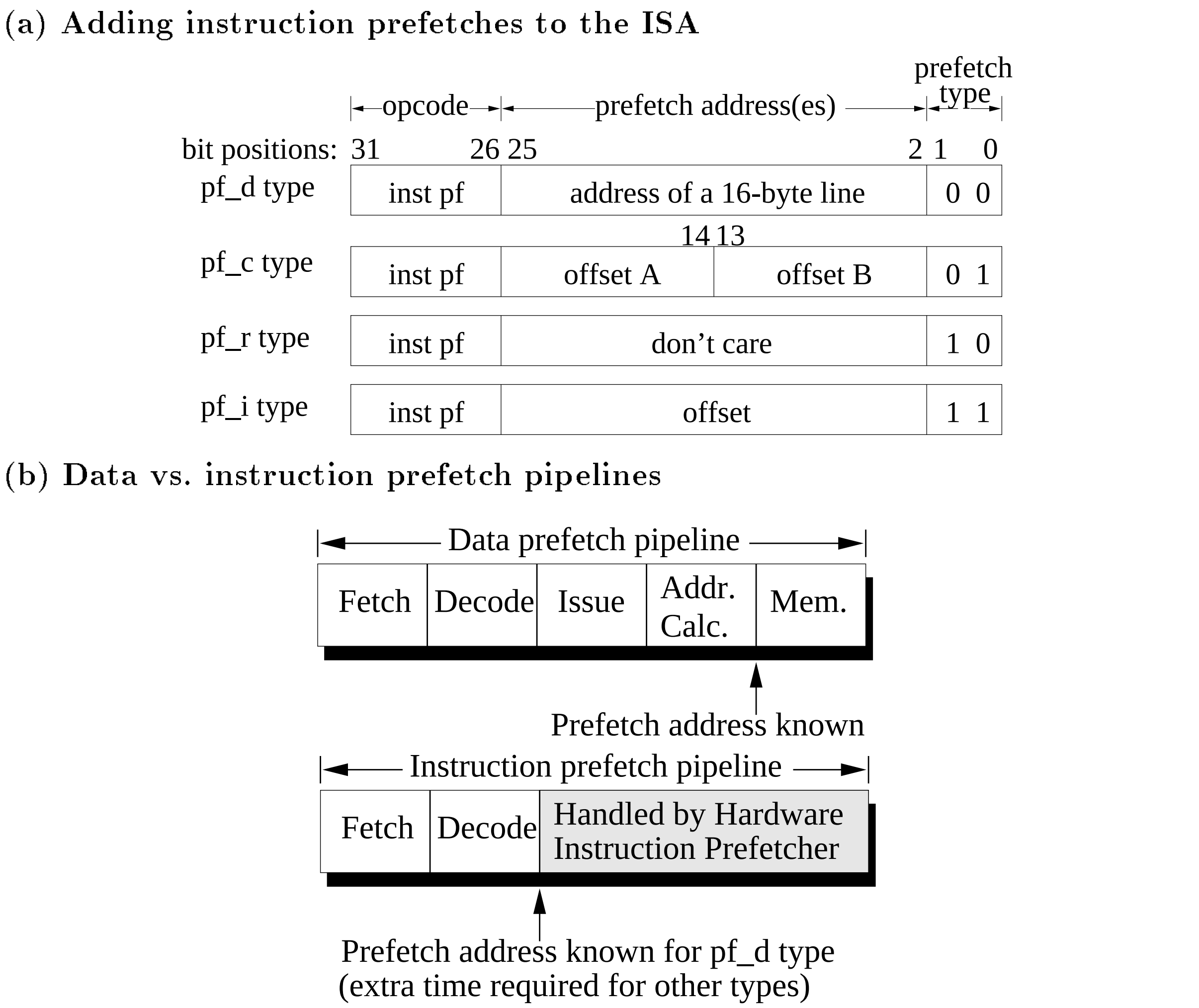


Figure 2.3: Possible extensions to the ISA and the CPU pipeline for instruction prefetches.

2.3.1 Extensions to the Instruction Set Architecture

Without loss of generality, we assume a base instruction set architecture (ISA) similar to the MIPS ISA [63]. Within a 32-bit MIPS instruction, the high-order six bits contain the opcode. For the jump-type instructions which implement static procedure calls, the remaining 26 bits contain the low-order bits of the target word address. We will use this same instruction format as our starting point.

There are many ways to encode our new instruction-prefetch instructions, and Figure 2.3(a) shows just one of the possibilities. An opcode is designated to identify instruction-prefetch instructions. In contrast with the standard jump-type instruction format, we assume that 24 bits (bits 2 through 25) contain information for computing the prefetch address(es), bits 1 and 0 indicate one of the four prefetch types. The prefetch type `pf_d` stores a single prefetch address in a format similar to a MIPS jump address. The only difference is that since the lower two bits are ignored, it effectively encodes a 16-byte-aligned address.[6] The `pf_c` type is a *compact* format which encodes two target addresses within the 24-bit field in the form of offsets between the target address lines and the prefetch instruction line itself (again, a single offset bit represents 16 bytes); each offset is 12 bits wide. The remaining two types are for prefetching indirect targets—`pf_r` is for procedure returns, and `pf_i` is for general indirect-jump targets. A `pf_r` prefetch does not require an argument since it implicitly uses the top of the return address stack as its address. A `pf_i` prefetch encodes the word offset between itself and the indirect-jump instruction that it is prefetching. To look up the prefetch address(es), this offset is added to the current program counter to create an index into the indirect-target table.

2.3.2 Impact on the Processor Pipeline

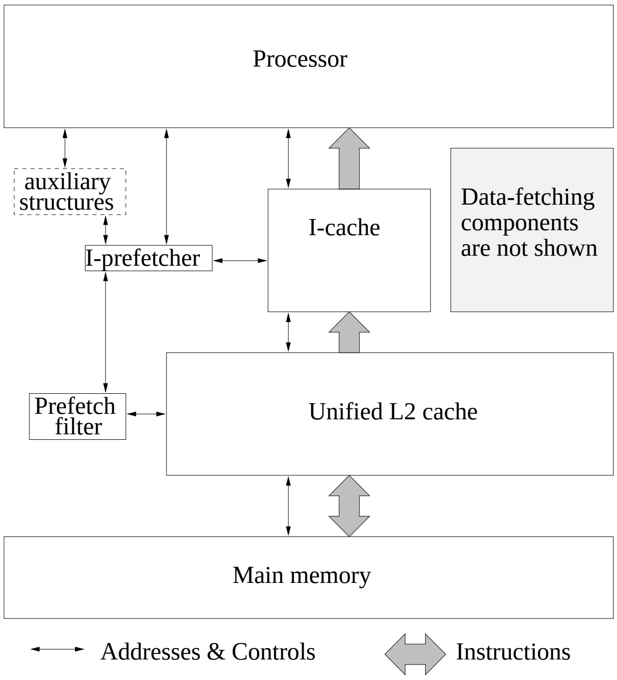
Many recent processors have implemented instructions for data prefetching [13, 112, 139]. With respect to pipelining, our *instruction* prefetches differ in two important ways from data prefetches: (i) the pipeline stage in which the prefetch address is known, and (ii) the computational resources consumed by the prefetches. Figure 2.3(b) contrasts the pipeline for data prefetches in the MIPS R10000 [139] with the pipeline for our instruction prefetches in an equivalent machine. As we see in Figure 2.3(b), the prefetch address of a `pf_d` instruction prefetch is known immediately after the *Decode* stage (the other three prefetch types would require some additional time), while the address for a data prefetch is not known until it is computed in the *Address Calculate* stage. Hence a `pf_d` instruction prefetch can be initiated two cycles earlier than a data prefetch. In addition, since instruction prefetches do not go through the latter three pipeline stages of a data prefetch (instead they are handled directly by the hardware instruction prefetcher after they are decoded), they do not contend for processor resources including functional units, the reorder buffer, register file, etc. In effect, the instruction prefetches are removed from the instruction stream as soon as they are decoded, thereby having minimal impact on most computational resources.

2.3.3 Extensions to the Memory Subsystem

Figure 2.4(a) shows our memory subsystem (only the instruction fetching components are displayed). The *I-prefetcher* is responsible for generating prefetch addresses and

[6] Since most machines have at least 16 byte instruction lines, this is not a limitation.

(a) Memory subsystem



(b) Example of prefetch filtering

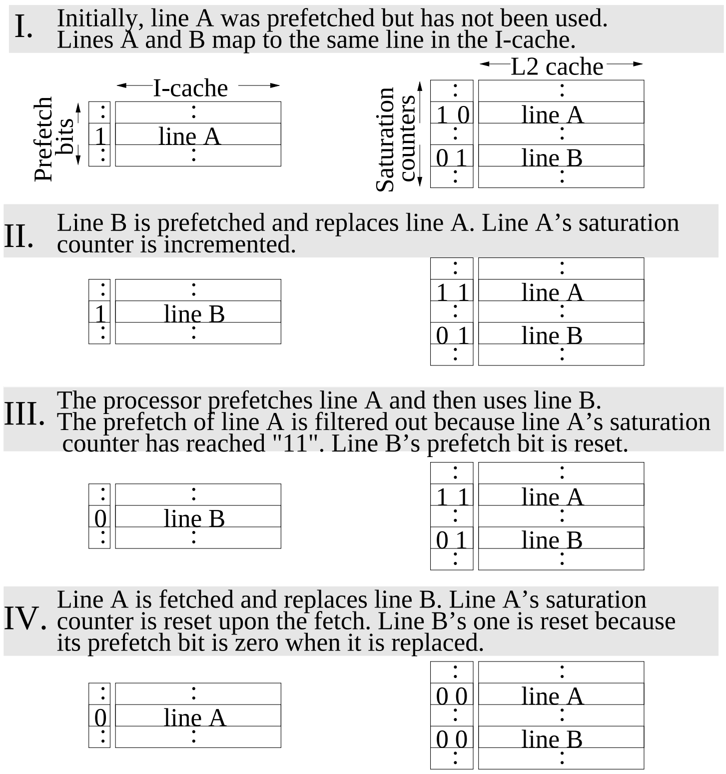


Figure 2.4: The memory subsystem and an example of the prefetch filtering mechanism.

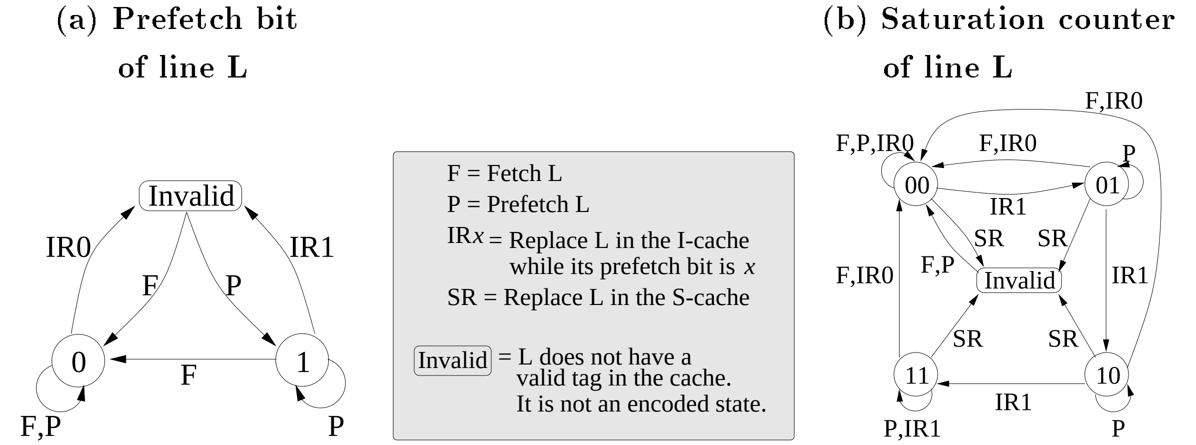


Figure 2.5: The states and transitions of (a) prefetch bits and (b) saturation counters under prefetch filtering.

launching prefetches to the unified L2 cache for both hardware and software initiated prefetching. Prefetch-address generation involves simple extraction of prefetch addresses from `pf_d` prefetches, adding constant offsets to the current program counter (for next-N line prefetching and `pf_c` prefetches), or retrieving prefetch targets from some hardware structures (for `pf_r` and `pf_i` prefetches). The I-prefetcher will not launch a prefetch to the L2 cache if the line being prefetched is already in the primary instruction cache (*I-cache*) or has an outstanding fetch or prefetch for the same line address. The *auxiliary structures* shown in Figure 2.4(a) include the return address stack and the indirect-target table used by `pf_r` and `pf_i` prefetches, respectively. These structures are not necessary if these two types of prefetches are not implemented.

Prefetch Filtering Mechanism

The *prefetch filter* sits between the I-prefetcher and the L2 cache to reduce the number of useless prefetches. In addition, a *prefetch bit* is associated with each line in the I-cache to remember whether the line was prefetched but not yet used, and a two-bit saturating counter value is associated with each line in the L2 cache to record the number of *consecutive* times that the line was prefetched but not used before it was replaced. The width of these saturating counters was selected through experimentation.

The prefetch filtering mechanism works as follows. When a line is *fetched* from the L2 cache to the I-cache, both the prefetch bit and the saturating counter value are reset to zero. When a line is *prefetched* from the L2 cache to the I-cache, its prefetch bit is *set* to one and its saturation counter does not change. When a prefetched line is actually used by a fetch, its prefetch bit is *reset* to zero. When a prefetched line l in the I-cache is

```
// This algorithm schedules prefetches for every basic block in the input executable.
algorithm Schedule_Prefetches
        (E: executable)  // the input executable
    return ();           // nothing to return

  foreach B in E do
    Schedule_Direct_Prefetches(B, B, 0, {}); // schedule direct prefetches for B
  end foreach;
  foreach B in E do
    if (B ends with an indirect jump or a procedure return) then
      // Schedule_Indirect_Prefetches() (code shown in Figure 2.7) inserts prefetches
      // for the target of the indirect control transfer located in B.
      Schedule_Indirect_Prefetches(B, B, Instruction_Count(B.instructions), {});
    end if;
  end foreach;
end algorithm;

// This algorithm inserts direct prefetches targeting a given basic block T.
algorithm Schedule_Direct_Prefetches
        (B: basic block,          // current basic block
         T: basic block,          // prefetch-target basic block
         D: integer,              // the prefetching distance between B and T
         S: set of basic blocks)  // basic blocks considered so far
    return (); // nothing to return

  if (B ∉ S) then // continue only if B hasn't been considered
    S := S ∪ {B}; // B is now being considered
    // First, determine if a prefetch for T, either launched by hardware or software, is already present in B.
    prefetched: boolean := Hardware_Prefetched(B, T) or Software_Prefetched(B, T);
    if (not prefetched) then
      // not prefetched yet, attempt to insert a prefetch at B if it is sufficiently early and necessary
      if (D ≥ PF_DIST and not Locality_Likely(B, T)) then
        // PF_DIST is the desirable prefetching distance.
        // Locality_Likely() determines the likelihood that B and T coexist in the cache.
        Attach_Direct_Prefetch(B, T); // attach a direct prefetch for T onto B
        prefetched := True; // a prefetch for T is just inserted into B
      end if;
    end if;
    if (not prefetched) then
      // still not prefetched yet, attempt to insert prefetches at the predecessors of B
      foreach B's predecessor block P do
        increment: integer;
        if (B is the fall-through block of P and P ends with a procedure call (static or dynamic)) then
          // also count the length of the called procedure
          increment := Shortest_Path(P.instructions);
        else // P reaches B via a static control transfer or a fall-through path
           // that does not end with a procedure call
          increment := Instruction_Count(P.instructions);
        end if;
        D' := D + increment; // update prefetching distance conservatively
        Schedule_Direct_Prefetches(P, T, D', S); // consider P next
      end foreach;
    end if;
  end if;
end algorithm;
```

Figure 2.6: Main prefetch scheduling algorithm and the algorithm for scheduling direct prefetches.

replaced by another line, then if the prefetch bit of line l is set, its saturation counter is incremented (unless it has already saturated, of course); otherwise, the counter is reset to zero. When the prefetch filter receives a prefetch request for line l, it will either respond normally if the counter value is below a threshold T, or else it will drop the prefetch and send a "prefetch canceled" signal to the processor if the counter has reached T (in our experiments, $T = 3$). Figure 2.4(b) shows an example of how the prefetch filtering mechanism works, and Figure 2.5 summarizes the states and transitions of the prefetch bit and the saturation counter for a particular cache line.

2.4 Compiler Support

The compiler is responsible for automatically inserting prefetch instructions into the executable. Since prefetch insertion is most effective if it begins after the code is otherwise in its final form, this new pass occurs fairly late in the compilation: perhaps at link time, or as in our case, implemented as a binary rewrite tool. Further, the compiler should schedule prefetches so as to achieve high coverage and satisfactory prefetching distances while minimizing the static and dynamic instruction overhead. Hence our compiler algorithm has two major phases, *prefetch scheduling* and *prefetch optimization*, which are described in the following subsections. In addition, the compiler also determines how many lines should be brought into the cache by a prefetch. A complete implementation of this algorithm was used throughout our experiments.

2.4.1 Preprocessing

This step prepares the information that will be used by the scheduling and the optimization phases. One important task here is to identify the interesting control structures in the input executable, in particular *loops*. We used the algorithm given in Section 10.4 of the Aho *et al* book [6] to find all the natural loops. This algorithm requires us to first compute the *dominators*.[7] In addition to this use, dominator information is also needed in other parts of our compiler algorithm.

2.4.2 Prefetch Scheduling

Figure 2.6 shows our main prefetch scheduling algorithm `Schedule_Prefetches`, which inserts direct and indirect prefetches into a given executable using two similar algorithms: `Schedule_Direct_Prefetches` and `Schedule_Indirect_Prefetches`.

[7] We say node d of a flow graph *dominates* node n if every path from the initial node of the flow graph to n goes through d [6].

Schedule_Direct_Prefetches

Algorithm `Schedule_Direct_Prefetches` (also shown in Figure 2.6) attempts to move prefetches for a given target basic block back by a distance of at least *PF_DIST* instructions. `Schedule_Direct_Prefetches` also tries to avoid inserting unnecessary prefetches by checking whether another prefetch with the same target already exists. This check is made by algorithm `Hardware_Prefetched`, which determines if the target basic block is covered by hardware sequential prefetching, and by algorithm `Software_Prefetched`, which tests whether an identical software prefetch is present in the same block. Algorithm `Locality_Likely`, which we will discuss shortly, checks whether the target basic block is likely to be already in the I-cache, in which case no prefetch is required. Also, as implied by the control flow of `Schedule_Direct_Prefetches`, if `prefetched` is *False* but `Locality_Likely` is *True*, we will keep attempting to insert prefetches further back, since, even though locality exists *within* a particular loop, we still need to prefetch *before* entering that loop. Note that when we consider inserting prefetches into a predecessor block, the prefetching distance must be updated. If there is a procedure invocation between the predecessor block and the current block, the new prefetching distance should include the length of the called procedure, which is estimated by `Shortest_Path`; otherwise, the new prefetching distance is simply obtained by adding the instruction count of the predecessor block to the old one. Finally, `Schedule_Direct_Prefetches` *terminates* along a particular path in one of the following four situations: (i) a prefetch is successfully inserted on that path; (ii) prefetching is discovered to be unnecessary for that path; (iii) there are no more predecessors to move further back; or (iv) a basic block is considered again to insert a prefetch (i.e. the path constitutes a cycle).

Schedule_Indirect_Prefetches

Working in a similar fashion to `Schedule_Direct_Prefetches`, `Schedule_Indirect_Prefetches` (shown in Figure 2.7) inserts prefetches for indirect jumps and procedure returns. However, unlike `Schedule_Direct_Prefetches`, the second input parameter of `Schedule_Indirect_Prefetc` is the basic block containing the indirect control transfer instead of the target basic block itself since the target basic block is unknown statically. Therefore, when we pass this basic block as the first basic block to consider (i.e. how `Schedule_Indirect_Prefetches` is invoked in `Schedule_Prefetches`), the initial prefetching distance is not zero but the length of this basic block. Another difference between `Schedule_Direct_Prefetches` and `Schedule_Indirect_Prefetches` is that `Hardware_Prefetched` and `Locality_Likely` are not called in `Schedule_Indirect_Prefetches` since they cannot get much informa-

```
// This algorithm inserts prefetches targeting the indirect control transfer located in
// a given basic block T.
algorithm Schedule_Indirect_Prefetches
        (B: basic block,          // current basic block
         T: basic block,          // the basic block that contains the indirect control transfer
         D: integer,              // the prefetching distance between B and
                                  // the target of the indirect control transfer
         S: set of basic blocks)  // basic blocks considered so far
    return (); // nothing to return

  if (B ∉ S) then // continue only if B hasn't been considered
    S := S ∪ {B}; // B is now being considered
    // First, determine if there is another indirect software prefetch for the same target in B.
    prefetched: boolean := Software_Prefetched(B, T);
    if (not prefetched) then
      // not prefetched yet, attempt to insert a prefetch at B if it is sufficiently early
      if (D ≥ PF_DIST) then
        // PF_DIST is the desirable prefetching distance.
        Attach_Indirect_Prefetch(B, T); // attach a prefetch targeting the indirect control transfer
        prefetched := True; // a prefetch is just inserted into B
      end if;
    end if;
    if (not prefetched) then
      // still not prefetched yet, attempt to insert prefetches at the predecessors of B
      foreach B's predecessor basic block P do
        increment: integer;
        if (B is the fall-through block of P and P ends with a procedure call (static or dynamic)) then
          increment := Shortest_Path(P.instructions);
        else // P reaches B via a static control transfer or a fall-through that does not
             // end with a procedure call.
          increment := Instruction_Count(P.instructions);
        end if;
        D' := D + increment; // update prefetching distance conservatively
        Schedule_Indirect_Prefetches(P, T, D', S); // consider P then
      end foreach;
    end if;
  end if;
end algorithm;
```

Figure 2.7: Algorithm for scheduling indirect prefetches.

tion without knowing the exact prefetch target address. While `Schedule_Indirect_Prefetches` handles both indirect jumps and procedure returns in mostly the same way, it has to make sure that prefetches for a return located in a procedure—say, P—will not be issued prior to the invocation of P. This is accomplished by the `Attach_Indirect_Prefetch` routine.

Locality_Likely

Given two basic blocks, the algorithm shown in Figure 2.8 estimates how much the presence of one block in the I-cache implies the other's. The likelihood of both appearing

```
// This algorithm determines the likelihood that two given basic blocks A and B coexist in the I-cache.
algorithm Locality_Likely
        (A: basic block,
         B: basic block)
    return (boolean); // true if A and B are likely to exist in the I-cache simultaneously

  L_A: set of loops := My_Loops(A);    // find the set of loops containing A
  L_B: set of loops := My_Loops(B);    // find the set of loops containing B
  L_common: set of loops := L_A ∩ L_B; // find the set of loops that are common to A and B
  foreach l ∈ L_common do
     if (Largest_Volume(l.body, True) < Effective_CacheSize) then
        // Largest_Volume() computes the largest possible volume of instructions to be fetched.
        // Effective_CacheSize is the effective I-cache size, after taking other factors such as
        // cache conflicts into account.
        // A and B are likely to coexist in the I-cache if their common loop can be entirely held
        // into the cache.
        return True;
     end if;
  end foreach;
  return False;
end algorithm;
```

Figure 2.8: Algorithm for determining the likelihood of two basic blocks existing in the cache simultaneously.

would be high if all the following three conditions are true: (i) both blocks belong to a common loop; (ii) both blocks are fetched or prefetched before entering the loop; and (iii) the entire loop (of course including the two blocks in question) will stay in the I-cache from the first to the last loop iteration. Condition (ii) is ensured automatically by the caller of `Locality_Likely` (e.g., `Schedule_Direct_Prefetches`), and therefore we only need to check conditions (i) and (iii) in `Locality_Likely`. Condition (i) is checked by finding the sets of loops that contain the two blocks and then intersecting them. For condition (iii), we estimate it to be true if the largest possible volume of instructions to be fetched by the loop is smaller than the "effective" size of the I-cache (the effective I-cache size is only a fraction of the actual I-cache capacity in order to account for effects due to cache conflicts and instruction alignment, etc.). In our implementation, the effective cache size is one-eighth of the actual size. Similar to this test for locality within loops, our implementation also includes a test for locality within *recursive procedure calls*.

Shortest_Path

Algorithm `Shortest_Path` in Figure 2.9 estimates the minimum number of instructions to be executed for a given control-flow structure. The interesting cases in this algorithm are: (i) conditional branches, where we choose the shorter of the "*then*" and "*else*" paths;

```
// This algorithm computes the shortest path through an instruction sequence.
algorithm Shortest_Path
        (I: list of instructions) // instructions at a given control-flow graph level
    return (shortestLength: integer);

  shortestLength: integer := 0;
  foreach i ∈ I do
    if (i is a conditional branch) then
      // pick the one with shorter path length
      shortestLength := shortestLength + min(Shortest_Path(i.then_part), Shortest_Path(i.else_part));
    else if (i is an unconditional branch) then
      shortestLength := shortestLength + Shortest_Path(i.target_part);
      // instructions following i won't be executed after the branch and hence they shouldn't be counted then
      return (shortestLength);
    else if (i is a loop) then
      if (isConstant(i.num_iterations)) then
        shortestLength := shortestLength + i.num_iterations * Shortest_Path(i.loop_body);
      else // assume at least one iteration when iteration count is unknown
        shortestLength := shortestLength + Shortest_Path(i.loop_body);
      end if;
    else if (i is a static, non-recursive procedure call) then
      shortestLength := shortestLength + Shortest_Path(i.procedure_body);
    else if (i is a dynamic procedure call) then
      // assume small length for dynamic procedure calls
      shortestLength := shortestLength + Small_ProcedureLength;
    end if;
    shortestLength := shortestLength + 1;
  end foreach;
  return (shortestLength);
end algorithm;
```

Figure 2.9: Algorithm for computing the shortest path through an instruction sequence.

(ii) unconditional branches, where we return the length of the target paths; (iii) loops, where we use the iteration count if it is known and otherwise assume that at least a single iteration is executed; (iv) static procedure calls, where we use the length of the procedure body, unless there is recursion; and (v) dynamic procedure calls, where we conservatively assume unknown procedure length to be small ($Small\_ProcedureLength = 10$ in our implementation).

Largest_Volume

This algorithm estimates the largest possible volume of instructions to be fetched in a given instruction sequence. As shown in Figure 2.10, `Largest_Volume` has a similar structure to `Shortest_Path`. The major difference of them is that while `Shortest_Path` computes in a *minimal* notion, `Largest_Volume` does in a *maximal* one. As a result, `Largest_Volume` needs an additional parameter `combine` to indicate whether we should

```
// This algorithm computes the largest possible volume of instructions fetched in an
// given instruction sequence.
algorithm Largest_Volume
          (I: list of instructions // instructions at a given control-flow graph level
           combine: boolean)        // whether need to accumulate the volume of
                                    // instructions on disjoint paths
    return (largestVolume: integer);

  largestVolume: integer := 0;
  foreach i ∈ I do
    if (i is a conditional branch) then
      if (combine) then
        // accumulate the volume of both paths
        largestVolume := largestVolume + Largest_Volume(i.then_part, combine)
                         + Largest_Volume(i.else_part, combine);
      else
        // pick the one with larger volume
        largestVolume := largestVolume + max(Largest_Volume(i.then_part, combine),
                                             Largest_Volume(i.else_part, combine));
      end if;
    else if (i is an unconditional branch) then
      largestVolume := largestVolume + Largest_Volume(i.target_part, combine);
      // instructions following i won't be executed after the branch and hence they shouldn't be counted then
      return (largestVolume);
    else if (i is a loop) then
      // conservatively assume that all instructions in the loop are used (in different iterations)
      largestVolume := largestVolume + Largest_Volume(i.loop_body, True);
    else if (i is a static, non-recursive procedure call) then
      largestVolume := largestVolume + Largest_Volume(i.procedure_body, combine);
    else if (i is a dynamic procedure call) then
      // assume large volume for dynamic procedure calls
      largestVolume := largestVolume + Large_ProcedureVolume;
    end if;
    largestVolume := largestVolume + 1;
  end foreach;
  return (largestVolume);
end algorithm;
```

Figure 2.10: Algorithm for computing the largest possible volume of instructions accessed.

sum up instructions accessed in *disjoint* paths or simply select the path accessing more instructions; the former case is used to conservatively count all possible instructions accessed by a loop. Note that when we encounter a dynamic procedure call, we assume a fairly large procedure body ($Large\_ProcedureVolume = 1024$ in our implementation).

2.4.3 Prefetch Optimization

After generating an initial prefetch schedule, the compiler then performs a number of optimizations attempting to minimize both static and dynamic prefetching overheads.

We devised our optimizations in light of existing compiler optimizations [6], including *code motion*, *common subexpression elimination*, and *extraction of loop invariants*. However, our optimizations differ from them in two aspects. First, we need to take *locality* information into account to make sure that these optimizations would not degrade the effectiveness of prefetching. Second, since prefetches are mainly performance hints, we can simply ignore the information that we do not know statically (e.g., indirect jump targets, mapping to cache sets); the worst case is to affect performance but does not break the program, as it would in classical compiler optimizations. Our optimization algorithm is composed of the following four passes. To make our discussion more concrete, we use the example in Figure 2.12 to help explain these passes.

Pass 1: Combining Prefetches at Dominators

This pass boosts prefetches that have been attached to a basic block b in the prefetch scheduling phase to b's nearest dominator (other than b itself) if the boosting is not harmful (it is harmful when the boosted prefetches may displace other useful instructions from the cache before b is referenced). After this boosting process, the compiler could combine redundant prefetches at dominators. For example, Figure 2.12(b) shows the result of combining the two prefetches of line `y` into one after boosting prefetches from basic blocks `D`, `E`, and `F` into their dominator `C`.

Pass 2: Eliminating Unnecessary Prefetches

A prefetch instruction u targeting a line l is *unnecessary* if l resides in the I-cache on *all* possible paths reaching u. To eliminate unnecessary prefetch instructions, we devise a data-flow analysis algorithm to estimate which instruction lines reside in the I-cache at a particular program point. This algorithm, called `Compute_Cached_Instructions`, resembles the one that computes *available expressions* in classical compiler optimizations [6]. It is shown in Figure 2.11.

Algorithm `Compute_Cached_Instructions` performs a *forward* data-flow analysis. For a given block B other than the initial block B_0, $in[B]$ obtains its value by intersecting $out[P]$'s for all predecessors P of B. We can then compute $out[B]$ by uniting $use[B]$, $pf[B]$, and ($in[B]$ - $displace[B]$). This union represents the fact that the instruction lines reside upon exiting a block is equal to the lines reside upon entering the block plus those that are newly brought into the cache by the block (i.e. $use[B] \cup pf[B]$) but exclude those that are displaced. The set $displace[B]$ is initialized using `Conflicting_Instructions`, which estimates the instruction lines that could possibly conflict with $use[B] \cup pf[B]$.

```
// This algorithm estimates the instruction lines that reside in the I-cache at each basic block of
// the input executable.
// Explanation for the data-flow analysis variables used in this algorithms:
//          in[B] = the set of instruction lines residing in the I-cache upon entering block B
//          out[B] = the set of instruction lines residing in the I-cache upon exiting block B
//          use[B] = the set of instruction lines used in block B
//          pf[B] = the set of instruction lines prefetched in block B
//          displace[B] = the set of instruction lines displaced from the I-cache in block B

algorithm Compute_Cached_Instructions
          (E: executable)    // the input executable
          (B0: basic block) // the initial block
    return (in); // in[B] for each block B

  in, out, use, pf, displace: array of set of instruction lines;
  // some initialization follows
  in[B0] := {};
  use[B0] := the set of instruction lines used in B0;
  pf[B0] := the set of instruction lines prefetched in B0;
  out[B0] := use[B0] ∪ pf[B0]; // in[B0] and out[B0] will never change then
  foreach B ≠ B0 in E do
    use[B] := the set of instruction lines used in B;
    pf[B] := the set of instruction lines prefetched in B;
    // Find out which instruction lines in U could be conflicting with those that are used or prefetched
    // in B, where U includes all instruction lines in E.
    displace[B] := Conflicting_Instructions(U, use[B] ∪ pf[B]);
    // assume that all instruction lines other than those that are displaced are cached initially
    out[B] := U - displace[B];
  end foreach;
  change: boolean := True;
  while change do // iterate until converged
    change := False;
    foreach B ≠ B0 in E do
      // We are sure that an instruction line resides in the cache upon entering B if it does upon
      // leaving every predecessor of B.
      in[B] :=           ⋂              out[P];
               P a predecessor of B
      oldout: set of instruction lines := out[B];
      out[B] := use[B] ∪ pf[B] ∪ (in[B] − displace[B]);
      if (out[B] ≠ oldout) then
        change := True;
      end if;
    end foreach;
  end while;
  return (in);
end algorithm;
```

Figure 2.11: Our data-flow analysis algorithm for estimating which instruction lines reside in the I-cache.

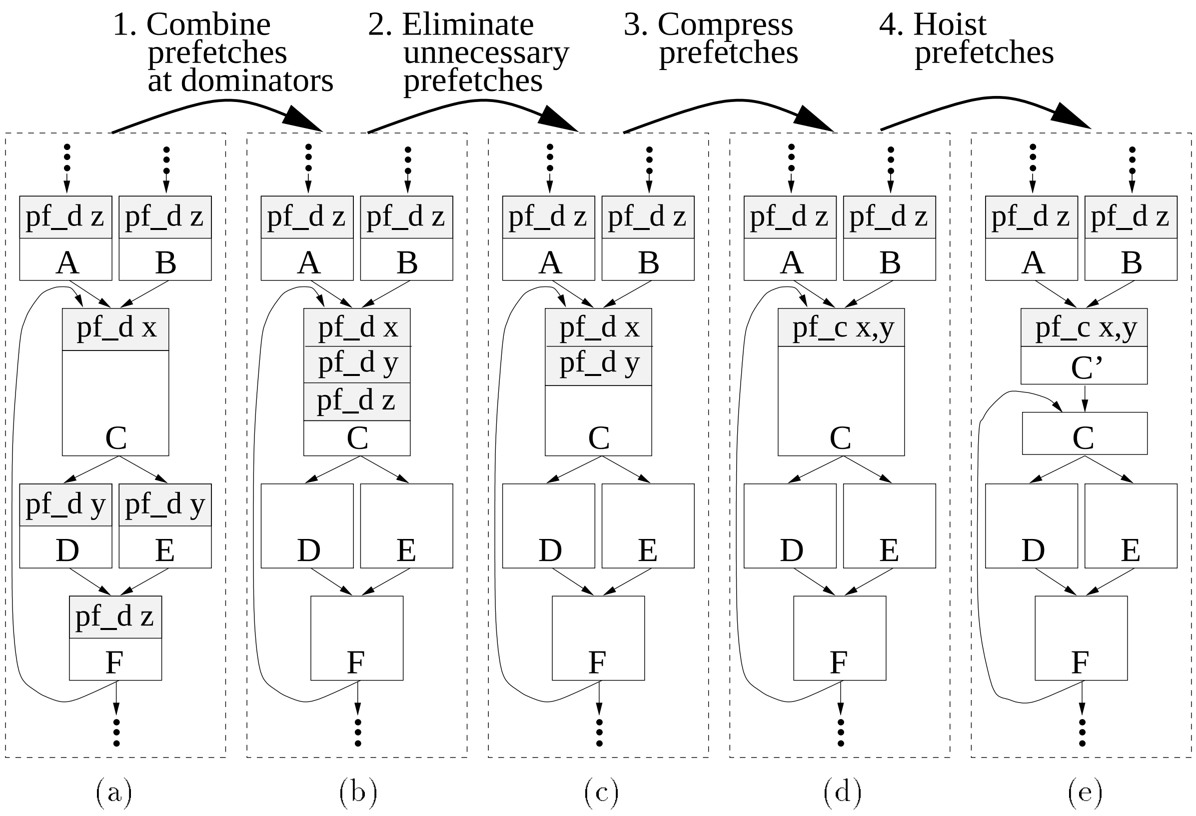


Figure 2.12: Example of prefetch optimization. `A` to `F` are basic blocks; `x`, `y` and `z` are cache line addresses. `C` is a dominator of `D`, `E`, `F`, and `C` itself. Part (a) is the initial schedule, and part (e) is the final optimized schedule.

Like many other iterative data-flow analyses, `Compute_Cached_Instructions` repeats computing *in*[] and *out*[] until *out*[] remains the same in two consecutive iterations.

In the example shown in Figure 2.12(b), we assume that once line `z` is prefetched before entering the loop, it will stay in the cache for the whole execution of the loop. Then line `z` will definitely be in the I-cache when we enter basic block `C` regardless of whether we came from `A`, `B`, or `F`. Therefore, the prefetch of line `z` in `C` is unnecessary and can be eliminated, as shown in Figure 2.12(c).

Pass 3: Compressing Prefetches

The compiler checks whether multiple `pf_d` prefetches in the same basic block can be compressed into a single compact prefetch. For each basic block b, the compiler needs to compute the offsets between the starting address of b and the target addresses of all `pf_d` prefetches scheduled in b. It then attempts to fit these offsets into a minimum number of compact prefetch instructions. Our example assumes that the address offsets of both lines `x` and `y` are representable within 12 bits, and therefore the two `pf_d` prefetches in `C` are compressed into a single `pf_c` prefetch, as shown in Figure 2.12(d).

Pass 4: Hoisting Prefetches

Finally, the compiler hoists prefetches scheduled inside a loop up to the nearest basic block that dominates but is not part of the loop, if the prefetches do not need to be re-executed at every iteration (which may not be the case if each iteration can access a large volume of instructions). If such a block is not found (most probably because these outside-loop dominators are too far away from the loop), a *pre-header* block will be created for the loop to hold the hoisted prefetches. For example, in Figure 2.12(e), a pre-header `C'` is created to immediately precede the header (i.e. `C`) of the loop containing `C`, `D`, `E`, and `F` to hold the hoisted `pf_c` prefetch. While this optimization does not reduce the code size, it can reduce the number of *dynamic* prefetches.

2.4.4 Determining the Software Prefetch Size

Since instruction accesses are fairly sequential, it is usually helpful to bring in more than one cache line by a software prefetch. Currently, our compiler decides a constant prefetch size for all software prefetches and passes it to the hardware prior to program execution. If hardware prefetch filtering is enabled, the software prefetch size is set to four; otherwise, it is set to two. Alternatively, the compiler can use variable prefetch sizes. However, this would require extra operands or more complex encoding in instruction-prefetch instructions in order to specify the prefetch size.

2.5 Experimental Framework

We performed our experiments on seven non-numeric applications which were chosen because their relatively large instruction footprints result in poor instruction cache performance. These applications are described Table 2.2, and all of them were run to completion. Table 2.3 shows the number of software prefetches inserted into the executable, broken down into the interprocedural and intraprocedural cases. As we see in Table 2.3, the software component of our scheme mainly targets interprocedural prefetching.

We performed detailed simulations of our applications on a dynamically-scheduled, superscalar processor similar to the MIPS R10000 [139]. Table 2.4 shows the parameters used in our model for the bulk of our experiments (we vary the latency and bandwidth later in Section 2.6.7). As shown in Table 2.4, we enhanced the memory subsystem in a

Table 2.2: Application characteristics. Note: the "combined" miss rate is the fraction of instruction fetches which suffer misses in both the 32KB I-cache *and* the 1MB L2 cache.

Name	Description	Input Data Set	Instructions Graduated	Miss Rate I-Cache	Miss Rate Combined
Gcc	The GNU C compiler drawn from SPEC92	The stmt.i in the reference input set	136.0M	2.63%	0.10%
Perl	The interpreter of the Perl language drawn from SPEC95	A Perl script called a2ps.pl which converts ascii to postscript	41.4M	5.03%	0.06%
Porky	A SUIF compiler pass for simplifying and rearranging codes	The compress95.c program in SPEC95 (default optimizations)	86.8M	2.38%	0.06%
Postgres	The PostgreSQL database management system [142]	A subset of queries in the Postgres Wisconsin benchmark	46.0M	3.76%	0.16%
Skweel	A SUIF compiler pass for loop parallelization	A program that computes Simplex (with all optimizations)	68.1M	2.22%	0.06%
Tcl	An interpreter of the script language Tcl version 7.6	Tcltags.tcl which makes Emacs-style TAGS file for Tcl source	37.5M	2.78%	0.02%
Vortex	The Vortex object-oriented database program in SPEC95	A reduced SPEC95 input set	193.0M	6.48%	0.08%

Table 2.3: Number of software prefetches inserted into the executable. Note: The "static prefetch count" is normalized to the size of the original executable. Prefetches are classified as either *interprocedural* or *intraprocedural*, depending on whether the prefetch target and the prefetch itself are in the same procedure.

Name	Static Prefetch Count (% of Original Executable Size) Interprocedural	Intraprocedural
Gcc	6.3%	1.6%
Perl	8.3 %	1.7 %
Porky	7.1%	0.4%
Postgres	8.3%	0.6%
Skweel	7.5 %	0.6%
Tcl	7.2%	1.0%
Vortex	10.3%	0.7%

few ways relative to the R10000 to provide better support for instruction prefetching—e.g., we added an eight-entry victim cache [61] and a 16-entry prefetch buffer [60].

We compiled each application as a "nonshared" executable with `-O2` optimization using the standard MIPS C compilers under IRIX 5.3. We implemented our compiler algorithm as a standalone pass which reads in the MIPS executable and modifies the

Table 2.4: Simulation parameters for the baseline architecture.

Pipeline Parameters	
Fetch & Decode Width	8 aligned sequential instructions
Issue & Graduate Width	4
Functional Units	2 Integer, 2 FP, 2 Memory, 2 Branch
Reorder Buffer Size	32
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction Scheme	2-bit Counters

Memory Parameters	
Line Size	32B
I-Cache	32KB, 2-way set-associative, 4 banks
Inst. Prefetch Buffer	16 entries
D-Cache	32KB, 2-way set-associative, 4 banks
Victim Buffers	8 entries each for data and inst.
Miss Handlers (MSHRS)	32 each for data and inst.
Unified S-Cache	1MB, 4-way set-associative
Primary-to-Secondary Miss Latency	12 cycles (plus any delays due to contention)
Primary-to-Memory Miss Latency	75 cycles (plus any delays due to contention)
Primary-to-Secondary Bandwidth	32 bytes/cycle
Secondary-to-Memory Bandwidth	8 bytes/cycle

binary. However, since we did not have access to a complete set of binary rewrite utilities, we tightly integrated our compiler pass with our simulator so that rather than physically generating a new executable, we instead pass a logical representation of the new binary to the simulator which it can then model accurately. For example, the simulator fetches and executes all of the new instruction prefetches as though they were in a real binary, and it remaps all instruction layouts and addresses to correspond to what they would be in the modified binary. Hence we truly emulate the physical insertion of prefetches at the expense of decreased simulation speed.

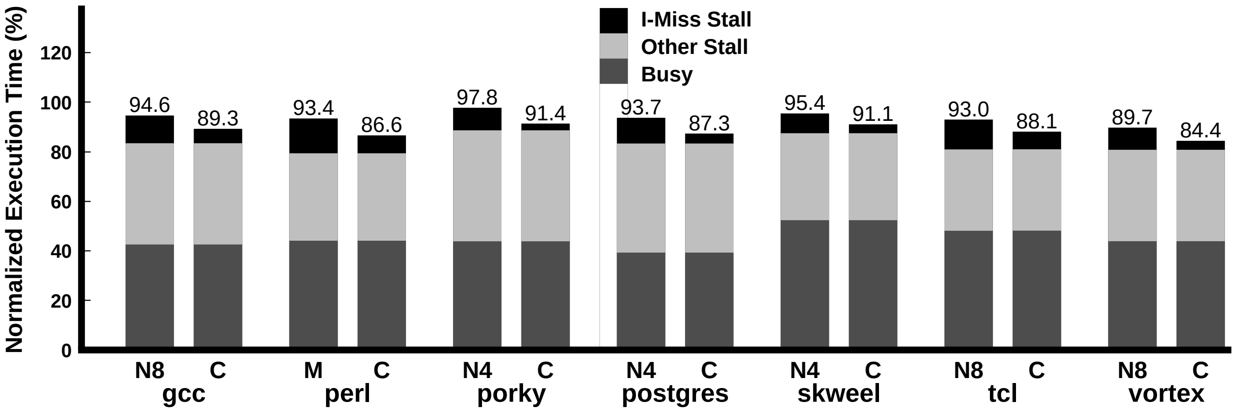


Figure 2.13: Performance comparison of our basic cooperative prefetching and the best performing existing schemes of individual applications (**N**x = next-x-line prefetching, **M** = Markov prefetching, **C** = cooperative prefetching).

2.6 Experimental Results

We now present results from our simulation studies. We start by evaluating the overall performance of our basic cooperative prefetching scheme(with only direct prefetches), and then evaluate the benefit of also adding indirect prefetches (i.e. `pf_r` and `pf_i`). Next, we examine the relative importance of the two key components of our scheme: prefetch filtering and compiler-inserted prefetching. We also quantify the impact of our compiler's prefetch optimizations, and of varying the prefetching distance parameter, on the code size and performance. We then investigate if cooperative prefetching could benefit from profiling information. After that, we explore the impact of varying cache latencies and bandwidth on the performance of our scheme. Finally, we evaluate whether cooperative prefetching is cost effective.

2.6.1 Performance of Basic Cooperative Prefetching

Our basic cooperative prefetching scheme includes compiler-inserted `pf_d` and `pf_c` prefetches, hardware-based next-8-line prefetching, and prefetch filtering. No `pf_r` or `pf_i` prefetches (and hence the required hardware structures) are used. A prefetching distance of 20 instructions is used for all applications. (We will discuss the impact of the prefetching distance more later in Section 2.6.5.)

Figure 2.13 shows the performance impact of cooperative instruction prefetching. For each application, we show two cases: the bar on the left is the best previously-existing prefetching scheme (seen earlier in Figure 2.1), and the bar on the right is cooperative

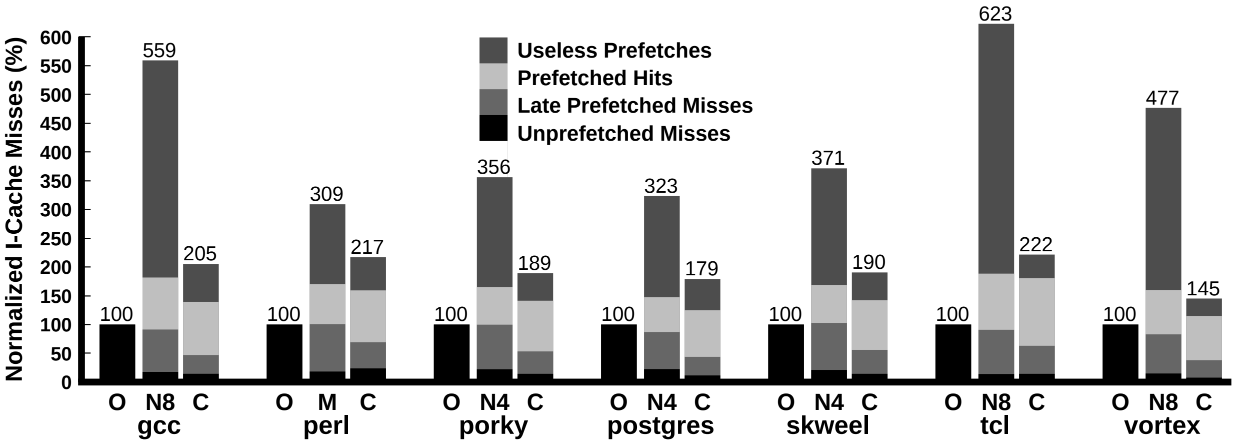


Figure 2.14: Breakdown of all I-cache misses. (**O** = original, **N**x = next-x-line prefetching, **M** = Markov prefetching, **C** = cooperative prefetching).

prefetching (**C**). Note that the number of instructions that actually graduate (i.e. the *busy* section) is equal in both cases because instruction prefetches are removed from the instruction stream once they are decoded (see Section 2.3.2). As we see in Figure 2.13, our cooperative prefetching scheme offers significant speedups over existing schemes (6.4% on average) by hiding a substantially larger fraction of the original instruction cache miss stall times (71% on average, as opposed to an average reduction of 36% for the best existing schemes).

To understand the performance results in greater depth, Figure 2.14 shows a metric which allows us to evaluate the coverage, timeliness, and usefulness of prefetches all on a single axis. This figure shows the total I-cache misses (including both fetch and prefetch misses) normalized to the original case (i.e. without prefetching) and broken down into the following four categories. The bottom section is the number of fetch misses that were not prefetched (this accounts for 100% of the misses in the original case, of course). The next section (*Late Prefetched Misses*) is where a miss has been prefetched, but the prefetched line has not returned in time to fully hide the miss (in which case the instruction fetcher stalls until the prefetched line returns, rather than generating a new miss request). The *Prefetched Hits* section is the most desirable case, where a prefetch fully hides the latency of what would normally have been a fetch miss, converting it into a hit. Finally, the top section is useless prefetches which bring lines into the cache that are not accessed before they are replaced.

Figure 2.14 shows that both cooperative prefetching and the best existing prefetching schemes achieve large coverage factors, as indicated by the small number of unprefetched misses. The main advantage of our scheme is that it is more effective at launching

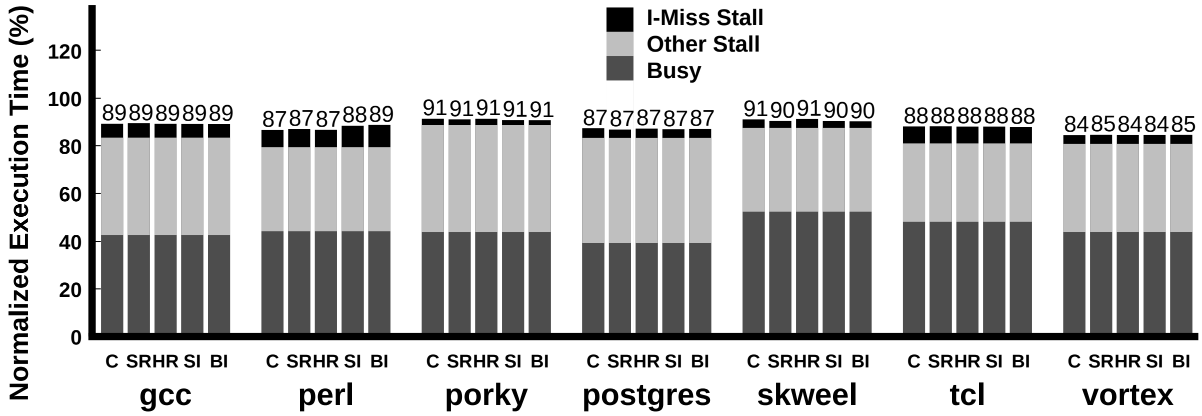


Figure 2.15: Impact of adding prefetches for procedure returns and indirect jumps (**C** = basic cooperative prefetching, **SR** = basic plus `pf_r` prefetches, **HR** = basic plus using hardware to prefetch the next three return addresses at each return, **SI** = basic plus `pf_i` prefetches with a smaller indirect-target table, **BI** = basic plus `pf_i` prefetches with a bigger indirect-target table).

prefetches early enough. This is demonstrated in Figure 2.14 by the significant reduction in *late prefetched misses*, the bulk of which have been converted into *prefetched hits*. We also observe in Figure 2.14 that both cooperative prefetching and existing schemes experience a certain amount of *cache pollution* since the sum of the bottom three sections of the bars adds up to over 100%. However, the *prefetch filtering* mechanism used by cooperative prefetching helps to reduce this problem, thereby resulting in a smaller total for the bottom three sections than the best existing scheme in all of our applications. In addition, Figure 2.14 shows another benefit of prefetch filtering: it dramatically reduces the number of useless prefetches. The reduction in total useless prefetches ranges from 2.4 in `perl` to 10.6 in `tcl`—on average, cooperative prefetching has achieved a sixfold reduction in useless prefetching.

2.6.2 Adding Prefetches for Procedure Returns and Indirect Jumps

Having seen the success of our basic cooperative prefetching scheme, we now evaluate the performance benefit of extending it to include the *indirect* prefetches—i.e. `pf_r` and `pf_i` prefetches for procedure returns and indirect jumps, respectively. Figure 2.15 shows the performance of five variations of cooperative prefetching: the basic scheme (**C**); the basic scheme plus `pf_r` prefetches (**SR**); the basic scheme plus using hardware to prefetch the top three addresses on the stack at each procedure return (**HR**); and two cases which include the basic scheme plus `pf_i` prefetches (**SI** and **BI**). Both schemes **SR** and **HR** use a 12-entry return address stack. While scheme **HR** has no instruction overhead, scheme **SR** has a better control over the prefetching distance via compiler scheduling. Scheme **SI** uses a 1 KB, 2-way set-associative indirect-target table where each entry holds up to four target addresses; scheme **BI** uses a 16 KB, 4-way set-associative indirect-target table with 16 targets per entry.

As we can see in Figure 2.15, the marginal benefit of supporting indirect prefetches is quite small for these applications. Part of the limitation is that only a relatively small fraction (roughly 15%) of the remaining misses which are not handled by our basic scheme are due to either procedure returns or indirect jumps, and therefore the potential for improvement is small. In addition, since some indirect jumps can have a fairly large number of possible targets—e.g., more than eight, as we observe in `perl` and `gcc`—prefetching all of these targets could result in cache pollution. Prefetching indirect jump targets may become more important in applications where they occur more frequently—e.g., object-oriented programs that make heavy use of virtual functions, or applications that use shared libraries. Although two of our applications are written in C++ (`porky` and `skweel`), they rarely use virtual functions. Since our applications show little benefit from `pf_r` and `pf_i` prefetches, we do not use them in the remainder of our experiments.

2.6.3 Importance of Prefetch Filtering and Software Prefetching

Two components of the cooperative prefetching design contribute to its performance advantages: prefetch filtering and compiler-inserted software prefetching. To isolate the contributions of each component, Figure 2.16 shows their performance individually as well as in combination. The relative importance of prefetch filtering versus compiler-inserted prefetching varies across the applications: in `tcl`, prefetch filtering is more important, and in `postgres`, compiler-inserted prefetching is more important. In all cases, the best performance is achieved when both techniques are combined, and in all but one case this results in a significant speedup over either technique alone. Intuitively, the reason for this is that the benefits of prefetch filtering (i.e. avoiding cache pollution) and software prefetching (i.e. issuing non-sequential prefetches early enough) are *orthogonal*. Hence both components of our design are clearly important for performance and are complementary in nature.

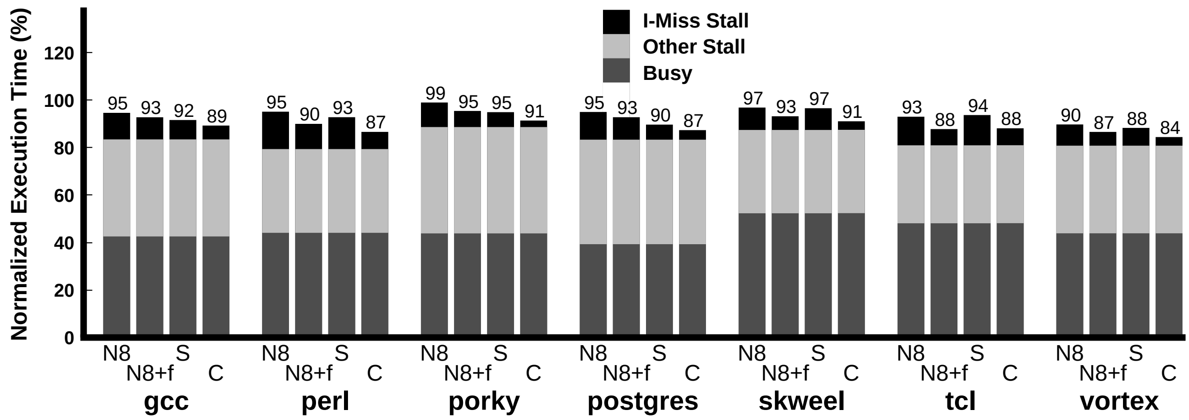


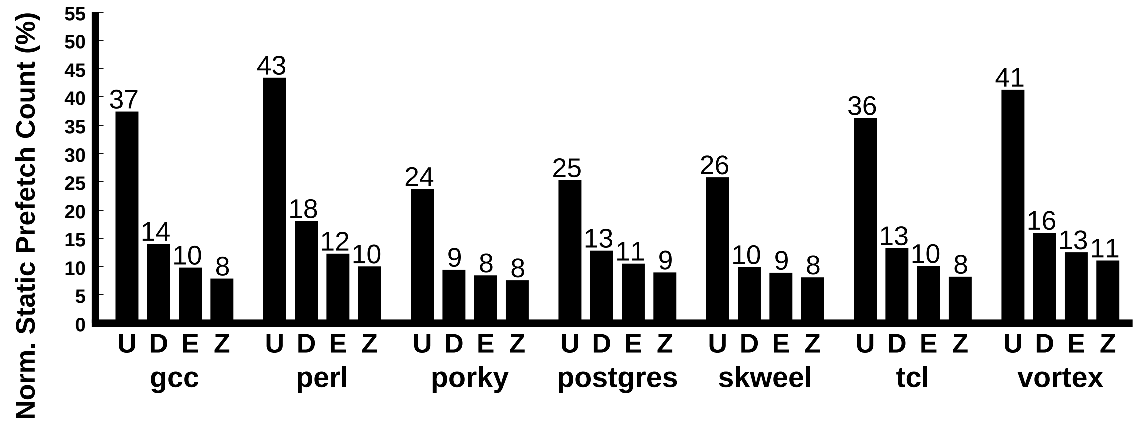
Figure 2.16: Performance of four different combinations of prefetch filtering and compiler-inserted prefetching (**N8** = next-8-line prefetching alone, **N8+f** = next-8-line prefetching with prefetch filtering, **S** = compiler-inserted prefetching alone without prefetch filtering, **C** = cooperative prefetching).

2.6.4 Impact of Prefetching Optimizations

To evaluate the effectiveness of the compiler optimizations discussed earlier in Section 2.4 in reducing the number of prefetches, we measured their impact both on code size and performance. Figure 2.17(a) shows the number of static prefetches remaining as each optimization pass is applied incrementally, normalized to the original code size. Without any optimization (**U**), the code size can increase by over 40%. Combining prefetches at dominators (**D**) dramatically reduces the prefetch count by more than half in all applications except `postgres`. Eliminating unnecessary prefetches and compressing prefetches further reduces the prefetch count by a moderate amount. (Prefetch hoisting has no effect on the static prefetch count, and therefore is not shown in Figure 2.17(a).) Altogether, the prefetch optimizations limit the prefetch count to only 9% of the original code size on average.

Figure 2.17(b) shows the impact of these optimizations on performance. As we see in this figure, combining prefetches at dominators results in a noticeable performance improvement in several cases (e.g., `gcc`, `perl`, and `tcl`). The other optimizations have a negligible performance impact. In fact, prefetch compression and hoisting sometimes degrade performance by a very small amount by changing the order in which prefetches are launched.

(a) Static prefetch count



(b) Performance

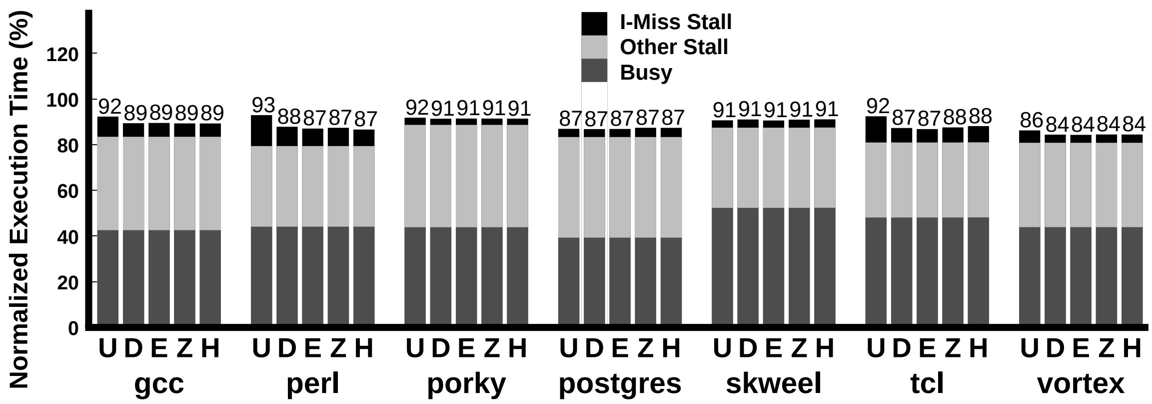


Figure 2.17: Impact of prefetch optimization on (a) the static prefetch count and (b) the performance of cooperative prefetching. (**U** = unoptimized, **D** = combining prefetches at dominators, **E** = case **D** plus eliminating unnecessary prefetches, **Z** = case **E** plus compressing prefetches, **H** = case **Z** plus hoisting prefetches). The y-axis of (a) is normalized to the number of instructions in the original executable.

2.6.5 Varying the Prefetching Distance

A key parameter in our prefetch scheduling compiler algorithm is the *prefetching distance* (i.e. PF_DIST in Figure 2.6). When choosing a value for this parameter, we must consider the following tradeoffs: we would like the parameter to be large enough to hide the expected miss latency, but setting the parameter too high can increase the code size (since more prefetches must be inserted to cover a larger number of unique incoming paths) and increase the likelihood of polluting the cache. In our experiments so far, we have used a prefetching distance of 20 instructions, which is roughly equal to the product

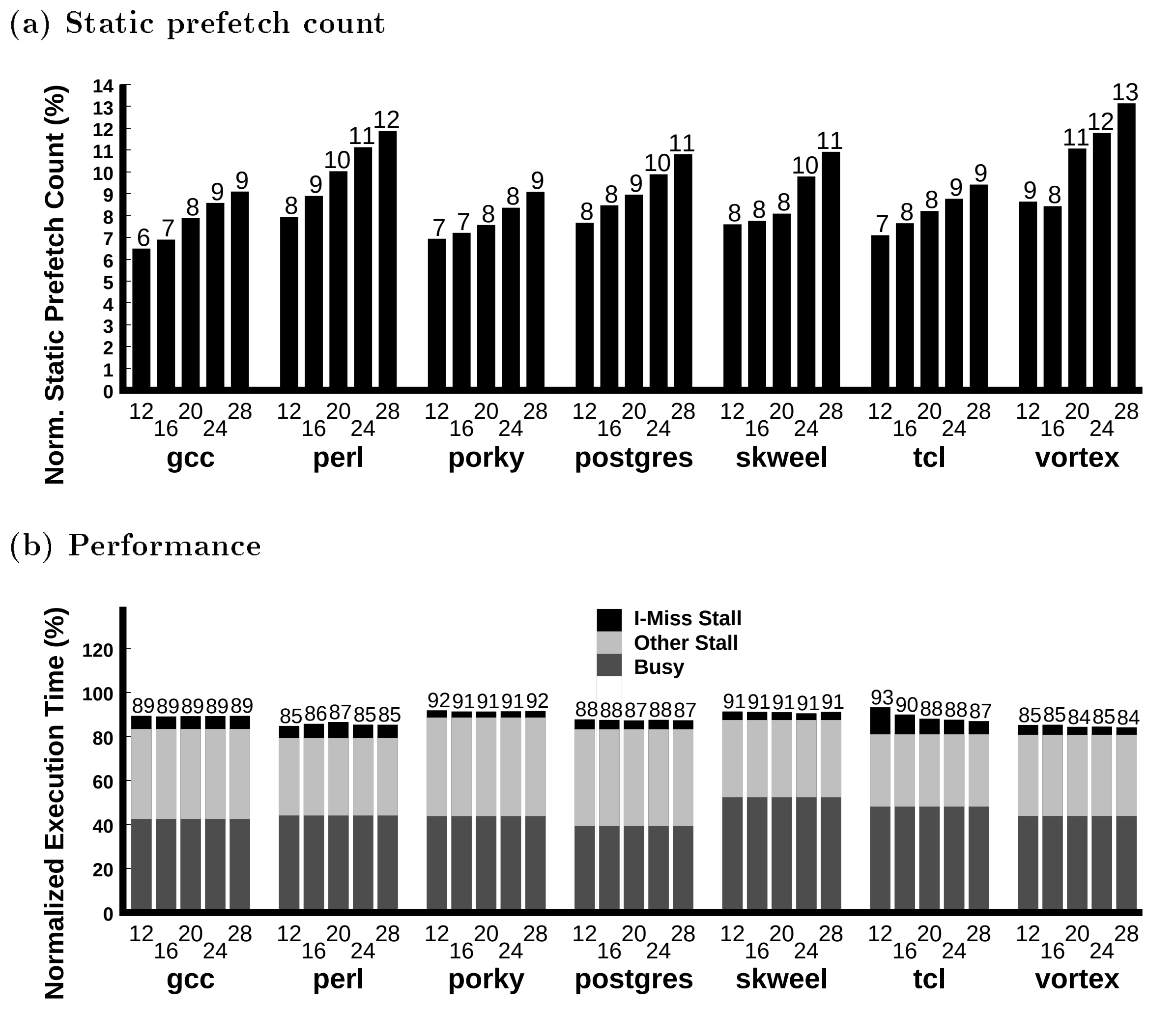


Figure 2.18: Impact of the prefetching distance on (a) the static prefetch count and (b) the performance of cooperative prefetching. (x = a prefetching distance of x instructions is used in the compiler scheduling; the case **20** is the default for our basic cooperative prefetching). The y-axis of (a) is normalized to the number of instructions in the original executable.

of the expected IPC (~1.6) and the primary-to-secondary miss latency ($\geq$ 12 cycles). To determine the sensitivity of cooperative prefetching to this parameter, we varied the prefetching distance across a range of five values from 12 to 28 instructions, and measured the resulting impact on both code size and performance (shown in Figures 2.18(a) and 2.18(b), respectively).

As we observe in Figure 2.18(a), increasing the prefetching distance can result in a noticeable increase in the code size. Fortunately, even with a prefetching distance as large as 28 instructions, the compiler is still able to limit the code expansion to less that 11% on average, due to the optimizations discussed in the previous section. In contrast,

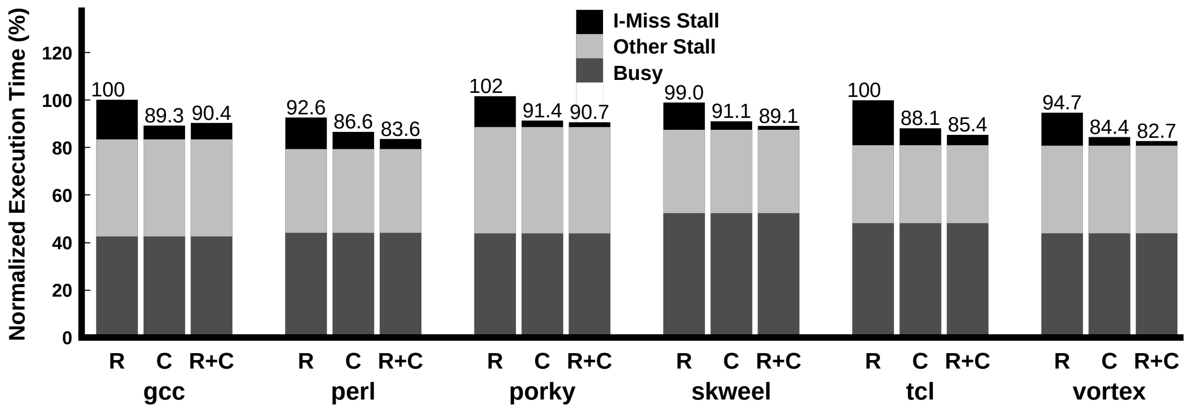


Figure 2.19: Performance impact of code reordering guided by profiling information (**R** = code reordered, **C** = cooperative prefetching, **R+C** = code reordered plus cooperative prefetching).

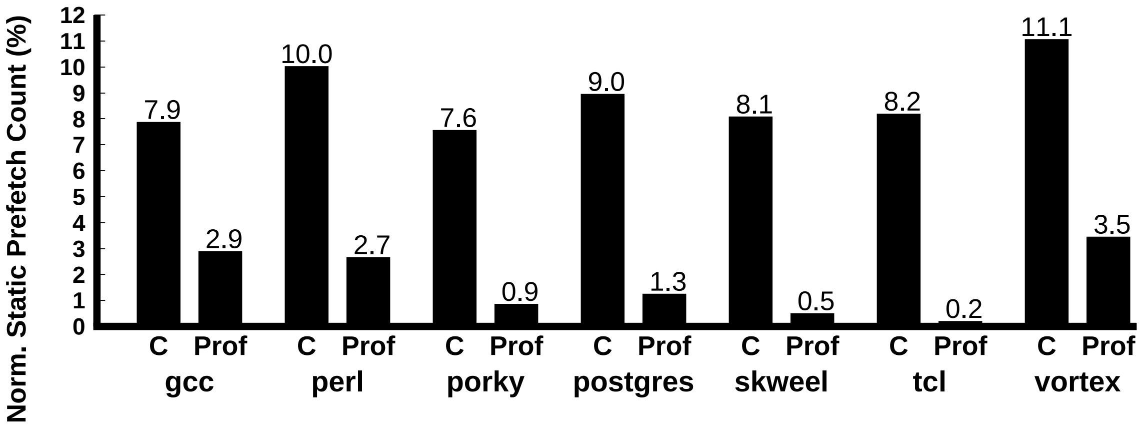
the *performance* offered by cooperative prefetching is less sensitive to the prefetching distance, as we see in Figure 2.18(b). While `tcl` enjoys a 6% speedup as we increase this parameter from 12 to 28 cycles, the other applications experience no more than a 2% fluctuation in performance across this range of values. Hence we observe that performance is not overly sensitive to this parameter.

2.6.6 Impact of Profiling Information

One advantageous feature of cooperative prefetching is that it requires no profiling information. In this section, we study how well our technique performs relative to some profiling-driven techniques. In addition, we investigate whether cooperative prefetching could be further improved by using profiling information. We present below the results of two experiments in which profiling information is used in different ways to improve instruction cache performance.

In the first experiment, control-flow profiling information is used to guide *code reordering* for better instruction locality. We used the IRIX utility `cord` to rearrange procedures in an executable according to an order determined by another IRIX utility, `prof`. The results are shown in Figure 2.19, where the three cases in each application are: code reordered (**R**), cooperative prefetching (**C**), and code reordered plus cooperative prefetching (**R+C**). For cases **R** and **R+C**, we experimented with two different code orders, one based on a profile where both the actual and training runs use the same input and the other based on a profile where the two runs use different inputs. The better performing code order is reported in Figure 2.19. (`Postgres` is not included in Figure 2.19

(a) Static prefetch count



(b) Performance

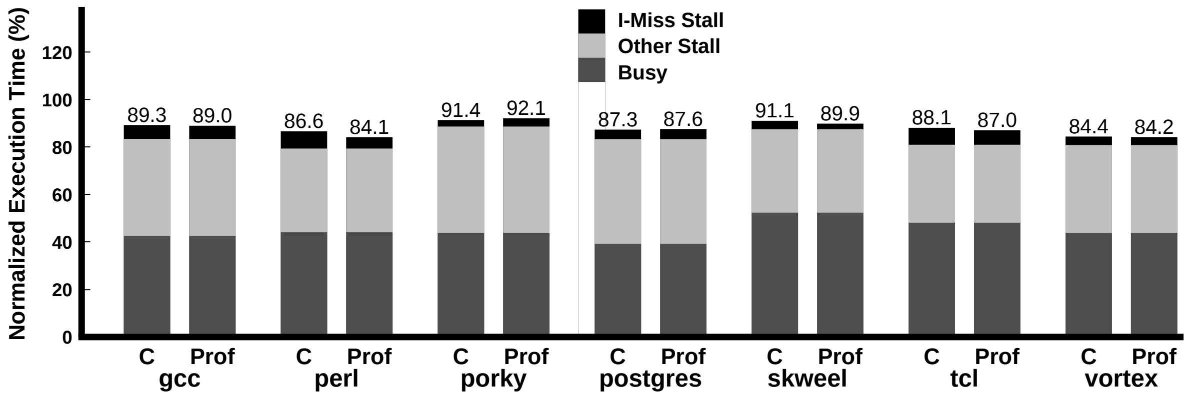


Figure 2.20: Impact of profiling-guided prefetch selection on (a) the static prefetch count and (b) the performance of cooperative prefetching. (**C** = original cooperative prefetching, **Prof** = cooperative prefetching with prefetches selected using profiling information.) The y-axis of (a) is normalized to the number of instructions in the original executable.

because `prof` is unable to handle this application.) We observe from Figure 2.19 that the performance of the code-reordered cases (**R**) is somewhat disappointing—it performs even worse than the original case in `porky`. The problem is that since the code order determined by `prof` mainly reduces the cache misses caused by *conflicts* in procedure mapping, there is little improvement in *compulsory* or *capacity* misses (they may get even worse in the new code order). Fortunately, by applying cooperative prefetching to the reordered code (**R+C**), these compulsory or capacity misses are largely eliminated, while at the same time the new code order helps reduce the conflict misses that are not handled by cooperative prefetching alone. As a result, the **R+C** cases have better performance than the **C** cases in all applications except `gcc`.

In the second experiment, we attempted to reduce software prefetching overheads by using profiling information. We first recorded the average I-cache miss rates of individual prefetch instructions in a training run. Then we decided to insert a prefetch instruction in the actual run whenever the average miss rate of that prefetch was higher than a given threshold—that is, we wish to eliminate prefetches that are *unnecessary* most of the time. The impact of this profiling-guided prefetch selection on both the static prefetch count and the performance is shown in Figure 2.20. We varied the threshold miss rates from 10^{-6} to 10^{-3}, and the **Prof** case of each application in Figure 2.20 used the threshold that resulted in the best performance. To be optimistic, we used the same inputs for both training and actual runs. It is obvious from Figure 2.20(a) that this profiling information is very effective at reducing the static prefetch count even after our prefetch optimizations are applied. However, as shown in Figure 2.20(b), the performance impact is less clear. While the **Prof** cases achieve 1%-3% speedups over the **C** cases in `perl`, `skweel`, and `tcl`, they also perform a little worse than the **C** cases in two applications because some discarded static prefetches turn out to be useful occasionally.

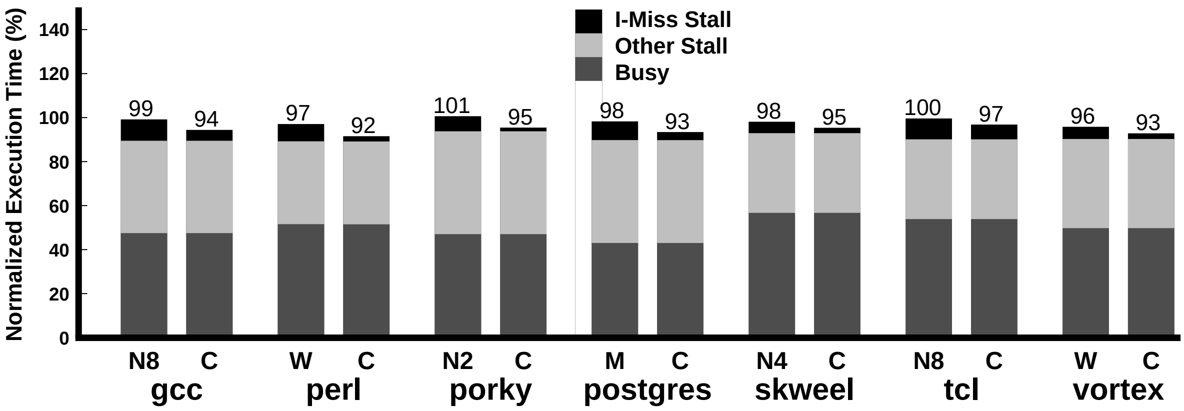
Overall, while the two kinds of profiling information studied in this section could further improve the performance of cooperative prefetching, they do not appear to be indispensable since cooperative prefetching alone already offers most of the same performance advantages.

2.6.7 Impact of Latency and Bandwidth Variations

We now consider the impact of varying miss latencies and available bandwidth between the primary and secondary caches on the performance of cooperative prefetching. Recall that in our experiments so far, the primary-to-secondary miss latency has been 12 cycles (plus any delays due to contention). Figure 2.21 shows the performance of the best performing existing schemes and cooperative prefetching when the primary-to-secondary latency is decreased to 6 cycles and increased to 24 cycles. (Note that the compiler's prefetching distance was set to 12 and 28 instructions, respectively, for the 6-cycle and 24-cycle cases.) As we see in Figure 2.21, cooperative prefetching still performs well under both latencies, and results in even larger improvements as the latency grows. In the 24-cycle case, cooperative prefetching results in an average speedup of 24.4%, which is significantly larger than the 14.2% speedup offered by the best existing scheme.

Turning our attention to bandwidth, recall that our experiments so far have assumed a bandwidth of 32 bytes/cycle between the primary instruction cache and the secondary

(a) Miss latency = 6 cycles



(b) Miss latency = 24 cycles

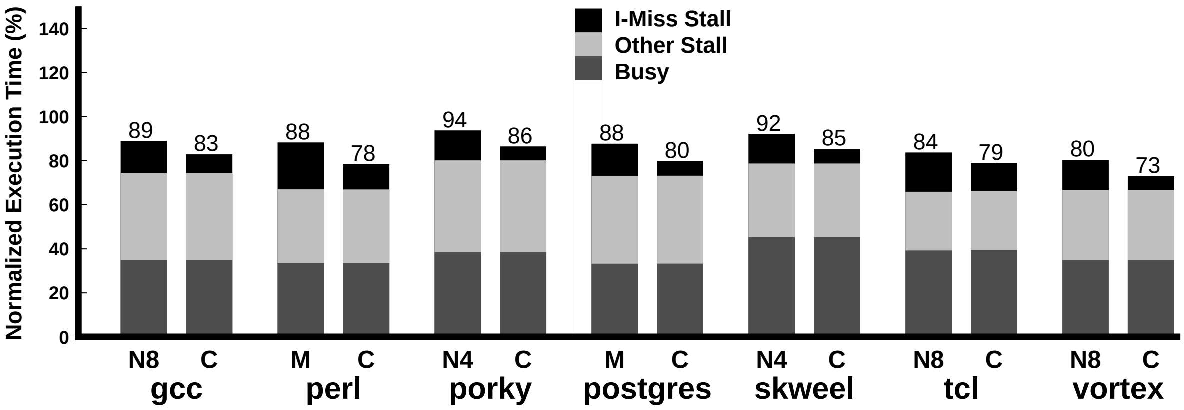
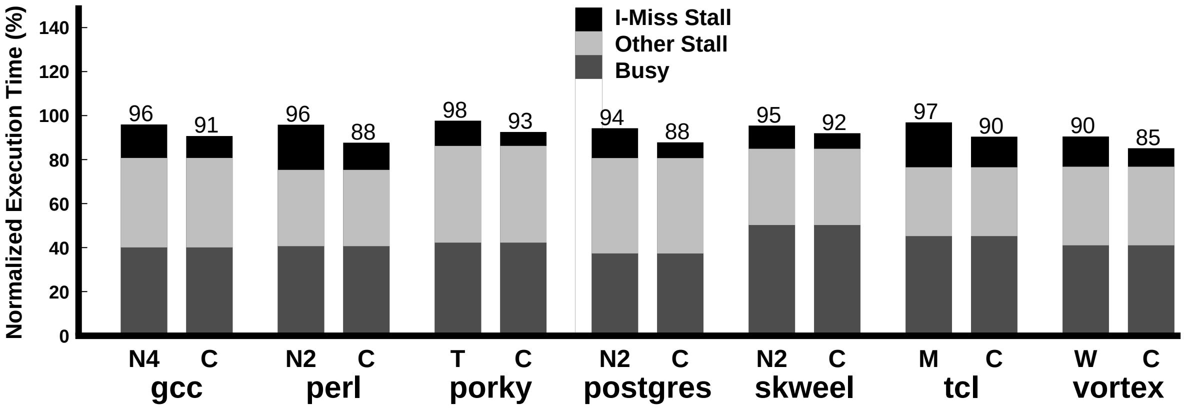


Figure 2.21: Impact of varying the cache miss latency. (**C** = cooperative prefetching, best performing existing schemes: **N**x = next-x-line prefetching, **T** = target-line prefetching, **W** = wrong-path prefetching, **M** = Markov prefetching).

cache. Figure 2.22 shows the impact of decreasing this bandwidth to 8 bytes/cycle, and of increasing it to unlimited bandwidth. There are two things to note from these results. First, we see in Figure 2.22(a) that while reducing the bandwidth does degrade the performance of cooperative prefetching somewhat—from an average speedup of 13.3% to 11.9%—the overall performance gain still remains high. Hence cooperative prefetching can achieve good performance with the range of bandwidth that is common for recent processors. (Note that this bandwidth includes servicing data cache misses as well.) Second, we observe in Figure 2.22(b) that *increasing* the bandwidth beyond 32 bytes/cycle does not significantly improve the performance of cooperative prefetching (the average speedup only increases from 13.3% to 13.7%). Therefore cooperative prefetching is not

(a) **Bandwidth = 8 bytes/cycle**



(b) **Bandwidth = infinite**

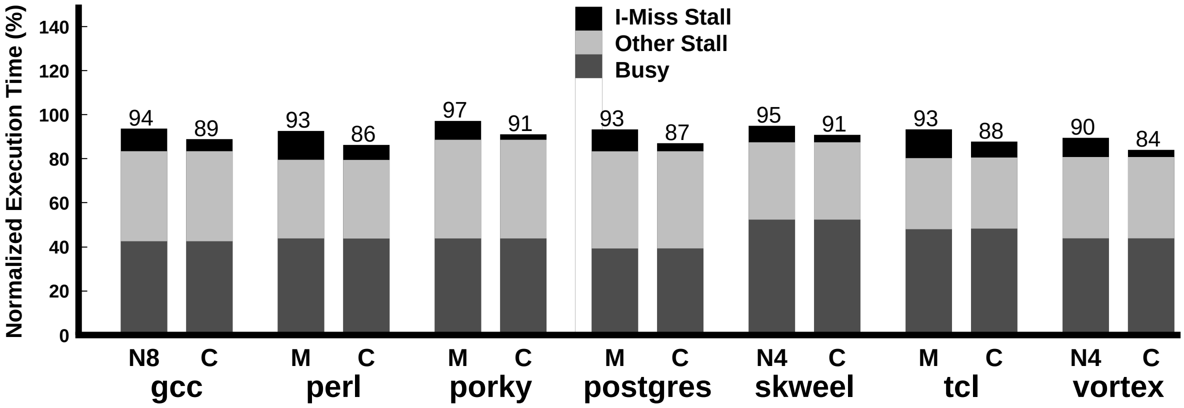


Figure 2.22: Impact of varying the bandwidth between the I-cache and L2 cache (**C** = cooperative prefetching, best performing existing schemes: **N**x = next-x-line prefetching, **T** = target-line prefetching, **W** = wrong-path prefetching, **M** = Markov prefetching).

bandwidth-limited, and it is more likely that it is limited by other factors (e.g., cache pollution, achieving a sufficient prefetching distance, etc.).

2.6.8 Cost Effectiveness

Having demonstrated the performance advantages of cooperative prefetching, we now focus on whether the additional hardware support is cost effective. One alternative to cooperative prefetching would be to simply increase the cache sizes by a comparable amount. (Note that this is overly simplistic since the primary cache sizes are often limited more by access time than the amount of silicon area available.) For our baseline

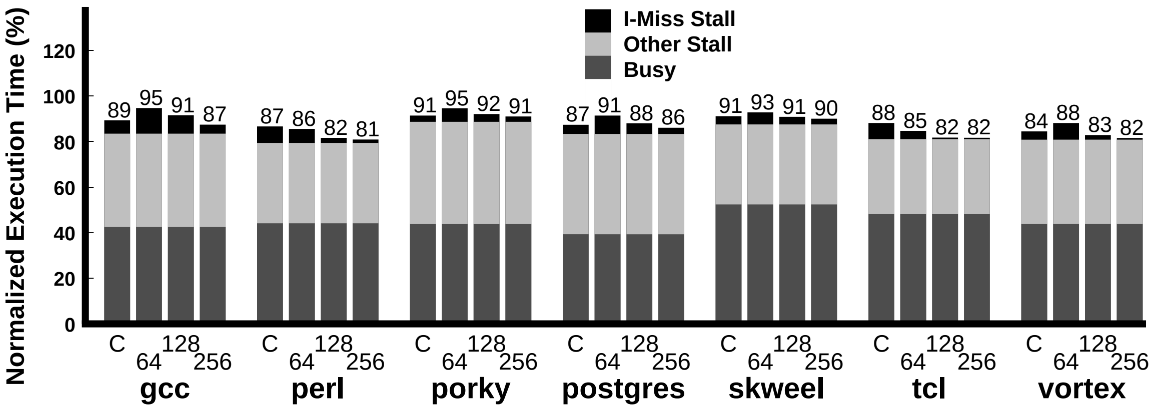


Figure 2.23: Performance comparison of cooperative prefetching and larger I-caches (**C** = a 32 KB I-cache with cooperative prefetching, x = an x KB I-cache without prefetching). The y-axis is normalized to the execution time of a 32 KB I-cache without prefetching.

architecture, the additional storage necessary to support basic cooperative prefetching is 640 bytes at the level of the primary I-cache (128 bytes for the prefetch bits used by prefetch filtering, and 512 bytes for the prefetch buffer), and 8 KB for the 2-bit saturating counters added to the L2 cache.

Figure 2.23 compares the performance of a 32 KB I-cache with cooperative prefetching with that of three larger I-caches, ranging from 64 KB to 256 KB, without prefetching. It is encouraging that the average speedup achieved by cooperative prefetching (13.3%) is greater than that obtained by doubling the cache size from 32 KB to 64 KB (10.8%) despite of the substantially higher hardware cost of the larger cache. In addition, cooperative prefetching outperforms the 128 KB I-cache in three of the seven applications, and is within 2% of the performance with a 256 KB I-cache in five cases. Overall, cooperative prefetching appears to be a more cost-effective method of improving performance than simply increasing the I-cache size.

2.7 Chapter Summary

To overcome the disappointing performance of existing instruction prefetching schemes on modern microprocessors, we have proposed and evaluated a new prefetching scheme whereby the hardware and software cooperate as follows: the hardware performs aggressive next-N-line prefetching combined with a novel *prefetch filtering* mechanism to get far ahead on sequential accesses without polluting the cache, and the compiler uses a novel algorithm to insert explicit *instruction-prefetch instructions* into the executable to prefetch non-sequential accesses. Our experimental results demonstrate that our scheme significantly outperforms existing schemes, eliminating 50% or more of the latency that had remained with the best existing scheme. This reduction in latency translates into a 13.3% average speedup over the original execution time on a state-of-the-art superscalar processor, which is more than double the 6.5% speedup achieved by the best existing scheme, and much closer to the maximum 20% speedup (for these applications and this architecture) in the ideal instruction prefetching case. These improvements are the result of launching prefetches earlier (thereby hiding more latency), while at the same time reducing the cache-polluting effects of useless prefetches dramatically.

Chapter 3

Compiler-Based Prefetching for Recursive Data Structures

3.1 Introduction

We begin our study on improving *data* cache performance by investigating software-controlled prefetching in this chapter. Specifically, we focus on an important class of non-numeric codes: those containing pointer-based data structures (also known as "recursive" data structures).

Recursive **D**ata **S**tructures (**RDS**s) include familiar objects such as linked lists, trees, graphs, etc., where individual nodes are dynamically allocated from the heap, and nodes are linked together through pointers to form the overall structure. For our purposes, "recursive data structures" can be broadly interpreted to include most pointer-linked data structures (e.g., mutually-recursive data structures, or even a graph of heterogeneous objects). From a memory performance perspective, these pointer-based data structures are expected to be an important concern for the following reasons. For an application to suffer a large memory penalty due to data replacement misses, it typically must have a large data set relative to the cache size. Aside from multi-dimensional arrays, recursive data structures are one of the most common and convenient methods of building large data structures (e.g., B-trees in database applications, octrees in graphics applications, etc.). As we traverse a large RDS, we may potentially visit enough intervening nodes to displace a given node from the cache before it is revisited; hence temporal locality may be poor. Finally, in contrast with arrays, where consecutive elements are at contiguous addresses and therefore stride-one accesses can exploit long cache lines, there is little inherent spatial locality between consecutively-accessed nodes in an RDS since they are dynamically allocated from the heap and can have arbitrary addresses. Therefore, techniques for coping with the latency of accessing these pointer-based data structures are essential. One possible approach to tackling this latency problem is prefetching. In the following subsection, we briefly discuss some previous work on prefetching for non-numeric codes.

3.1.1 Previous Work on Data Prefetching for Non-Numeric Codes

Although prefetching has been studied extensively for array-based numeric codes [10, 13, 20, 27, 45, 67, 95, 96, 112], relatively little work has been done on non-numeric applications. Chen *et al.* [29] used global instruction scheduling techniques to move address generation back as early as possible to hide a small cache miss latency (10 cycles), and found mixed results. Zhang and Torrellas [143] proposed a hardware-assisted scheme for prefetching irregular applications in shared-memory multiprocessors. Under their scheme, programs are annotated to bind together groups of data (e.g., fields in a record or two records linked by a pointer), which are then prefetched under hardware control. Their scheme has two shortcomings:(i) annotations are inserted manually, and (ii) their hardware extensions are not likely to be applicable in uniprocessors. Joseph and Grunwald [60] proposed a hardware-based Markov prefetching scheme which prefetches multiple predicted addresses upon a primary cache miss. While Markov prefetching can potentially handle chaotic miss patterns, it requires considerable amount of hardware support and is quite limited in selecting what to prefetch and controlling the prefetching distance. We are also aware of a number of hardware prefetching techniques [64, 110, 111] for pointer-based codes that have been proposed since our prefetching techniques were published [79]. Roth *et al.* [110] studied a dependence-based mechanism that dynamically identifies the RDS accesses in a program and prefetches by speculatively pre-executing these accesses. Roth and Sohi [111] then investigated jump-pointer-based prefetching schemes in hardware, which were derived from one of our prefetching techniques (history-pointer prefetching). Most recently, Karlsson *et al.* [64] suggested storing previously-seen RDS addresses into a so-called prefetch array and proposed a hardware scheme to prefetch all addresses in this array in a single instruction. As mentioned in their paper, their technique is essentially an extended integration of two of our prefetching techniques (greedy prefetching and history-pointer prefetching).

To our knowledge, the only compiler-based pointer prefetching scheme in the literature is the SPAID scheme proposed by Lipasti *et al.* [77]. Based on an observation that procedures are likely to dereference any pointers passed to them as arguments, SPAID inserts prefetches for the objects pointed to by these pointer arguments at the call sites. Therefore this scheme is only effective if (i) dereferencing these pointer arguments causes significant cache miss penalties and (ii) the interval between the start of a procedure call and its dereference of a pointer is comparable to the cache miss latency. We will present a performance comparison of SPAID and our proposed prefetching techniques later in Section 3.5.7.

Up till now, the compiler support for exploiting software-controlled prefetching for RDSs has remained an open question. In this chapter, we address this open research question by designing and evaluating compiler-based prefetching schemes which successfully tolerate the latency of accessing recursive data structures in modern microprocessor-based systems.

3.1.2 An Overview

This chapter is organized as follows. We begin in Section 3.2 by identifying the fundamental problem that makes prefetching RDSs difficult, and proposing a guideline for devising successful prefetching schemes. Based on this guideline, we design three different prefetching algorithms. In Section 3.3, we describe how these algorithms are implemented in an optimizing research compiler (SUIF). Section 3.4 describes our experimental framework, and Section 3.5 presents our experimental results where we evaluate all three prefetching algorithms on the Olden benchmark suite through detailed simulations of a modern superscalar processor. Finally, we summarize our findings of this chapter in Section 3.6.

3.2 Software-Controlled Prefetching for RDSs

In this section we discuss the major issues and challenges involved in software-controlled prefetching for RDSs, we present guidelines for overcoming these challenges, and we describe three prefetching algorithms based on these guidelines.

3.2.1 Challenges in Prefetching RDSs

Any software-controlled prefetching scheme can be viewed as having two major phases. First, an *analysis* phase predicts which dynamic memory references are likely to suffer caches misses, and hence should be prefetched. Second, a *scheduling* phase attempts to insert prefetches sufficiently far in advance such that latency is effectively hidden, while introducing minimal runtime overhead. For array-based applications, the compiler can use *locality analysis* to predict which dynamic references to prefetch, and *loop splitting* and *software pipelining* to schedule prefetches [95].

A fundamental difference between array references and pointer dereferences is the

(a) Example Code

```
for (i=1; i<=n; i++) {
  listNode *p = listHead[i];
  while(p) {
    work(p→data);
    p = p→next;
  }
}
```

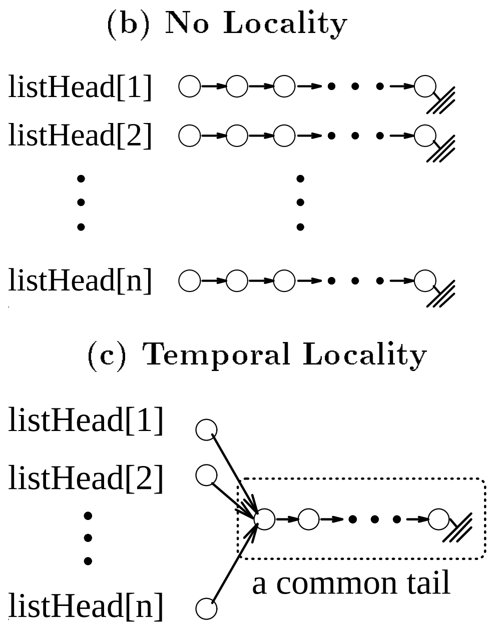


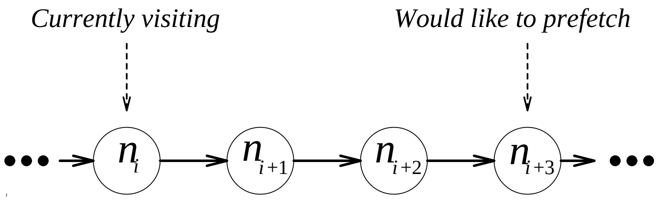
Figure 3.1: Example of list traversals, both with and without temporal data locality.

way addresses are generated. The address of an array reference $A[i]$ can always be computed once a value of i is chosen. In contrast, the address of a pointer dereference $*p$ is unknown unless the value stored in p is read. This difference makes both the analysis and scheduling phases significantly more challenging for RDSs than for arrays.

Analysis

To illustrate the difficulty of analyzing data locality in RDSs, consider the code in Figure 3.1(a), where we are traversing `n` different linked lists. In one extreme, the nodes may be entirely disjoint (as illustrated in Figure 3.1(b)), in which case we would want to prefetch every list node. Another possibility might be that each list shares a long common "tail" starting with the second list node (as illustrated in Figure 3.1(c)). In this latter case, there would be significant temporal locality (assuming the cache is large enough to contain the common tail), and ideally we would only want to prefetch the nodes in the common tail during the first list traversal (i.e. when `i`=1). Unfortunately, despite the significant progress that has been made recently in pointer analysis techniques for heap-allocated objects [40, 42, 51], compilers are still not sophisticated enough to differentiate these two cases automatically. In general, analyzing the addresses of heap-allocated objects is a very difficult problem for the compiler.

(a) Traversing an RDS



(b) Prefetching 3 Nodes Ahead

```
while (p) {
   prefetch(p→next→next→next);
   work(p→data);
   p = p→next;
}
```

(c) Prefetching 1 Node Ahead

```
while (p) {
   prefetch(p→next);
   work(p→data);
   p = p→next;
}
```

Figure 3.2: Illustration of the pointer-chasing problem.

Scheduling

Our ability to schedule prefetches for an RDS is also constrained by the fact that nodes are linked together through pointers. For example, consider the case shown in Figure 3.2(a), where assuming that three nodes worth of computation is needed to hide the latency, we would like to initiate a prefetch for node n_{i+3} while we are visiting node n_i. The problem is that to compute the address of node n_{i+3}, we must first dereference a pointer in node n_{i+2}, and to do that, we must first dereference a pointer in node n_{i+1}, etc. As a result, one cannot prefetch (or fetch) a future node until all nodes between it and the current node have been fetched. However, the very act of touching these intermediate nodes means that we cannot tolerate the latency of fetching more than one node ahead. For example, the prefetching code shown in Figure 3.2(b) will not hide any more latency than the code in Figure 3.2(c).[1] In fact, the code in Figure 3.2(c) is likely to run faster since it has less instruction overhead. This example illustrates what we refer to as the *pointer-chasing problem.*

Since scheduling RDS prefetches is such a difficult problem, we make it the primary focus of this study. Improvements in analysis tend to reduce prefetching overhead by eliminating unnecessary prefetches. However, without sufficient scheduling techniques, there will be no upside to prefetching and hence reducing overhead will be irrelevant. Fortunately, as we discuss in the next subsection, there are techniques for scheduling

[1]Assuming that nodes are not larger than cache lines; if they are, then prefetching further ahead can potentially result in a pipelining benefit.

prefetches that avoid the pointer-chasing problem.

3.2.2 Overcoming the Pointer-Chasing Problem

Let us formalize the pointer-chasing problem as follows. At a given RDS node n_i with address A_i, we wish to prefetch the node n_{i+d} that will be visited d nodes after n_i. We choose d (the *prefetching distance*) to be just large enough to hide the cache miss latency: $d = \lceil \frac{L}{W} \rceil$, where L is the expected miss latency and W is the estimated amount of computation between node accesses in cycles. To prefetch n_{i+d}, we must compute its address A_{i+d} based on the information available at n_i. The relationship between A_i and A_{i+d} can be expressed as:

$$A_{i+d} = \mathcal{F}(d, A_i)$$

where $\mathcal{F}$ is an address generating function.

A key factor in whether prefetch scheduling is effective is the *number of pointer-chain dereferences* required within the RDS to evaluate $\mathcal{F}(d, A_i)$, which we denote as $\|\mathcal{F}\|$. To overcome the pointer-chasing problem, we would like $\|\mathcal{F}\|$ to be as small as possible. If $\mathcal{F}$ is implemented by following the pointer chain from n_i to n_{i+d}, then $\|\mathcal{F}\| = d$. Instead, we will consider the cases where $\|\mathcal{F}\| = 1$ and $\|\mathcal{F}\| = 0$ (other values of $\|\mathcal{F}\|$ are possible, but do not appear to be interesting).

The case where $\|\mathcal{F}\| = 1$ means that only one pointer dereference is needed within the RDS to compute A_{i+d} at node n_i. This implies that n_i needs a direct pointer to n_{i+d}—we call this pointer a *jump-pointer*.[2] Jump-pointers can occur either *naturally* or *artificially* with respect to the RDS: a natural jump-pointer is a pointer that already exists in n_i, whereas an artificial jump-pointer is added to n_i for the purpose of prefetching. With natural jump-pointers, we are using one of the pointers at n_i to *approximate* A_{i+d}. The advantage is that no extra storage or computation is needed to create a natural jump-pointer, but unfortunately the effectiveness of prefetching may be limited by the accuracy of this approximation. In contrast, we require additional storage and computation to add artificial jump-pointers to an RDS, but hopefully these pointers will yield A_{i+d} more precisely (particularly if the structure of the RDS does not change rapidly between times when the artificial jump-pointers are set).

The case where $\|\mathcal{F}\| = 0$ means that *no* pointer dereferences are required to compute A_{i+d} at n_i. This is obviously a good case, but how can one compute the address of a heap-allocated object (which normally can reside at an arbitrary address) without

[2] A similar data structure called *skip lists* [108] has been used to accelerate searching and facilitate parallel processing on linked lists.

(a) Code with Greedy Prefetching

```
preorder(treeNode * t) {
  if (t != NULL) {
    prefetch(t→left);
    prefetch(t→right);
    process(t→data);
    preorder(t→left);
    preorder(t→right);
  }
}
```

(b) Cache Miss Behavior

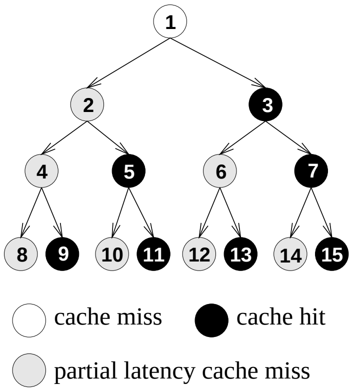


Figure 3.3: Illustration of greedy prefetching.

dereferencing any pointers? The answer is that we must have special knowledge of an RDS's layout in memory such that A_{i+d} can be directly implied from A_i and d.[3] There are many ways to accomplish this. For example, one could map a tree into an array structure such that there was a one-to-one mapping between the tree position and an array index. We will discuss the details of the approach we take later in Section 3.2.2.

In the remainder of this section, we propose three prefetching schemes with various $\|\mathcal{F}\|$ which avoid the pointer-chasing problem: *greedy prefetching* corresponds to $\|\mathcal{F}\| = 1$ using natural jump-pointers; *history-pointer prefetching* corresponds to $\|\mathcal{F}\| = 1$ using artificial jump-pointers; and *data-linearization prefetching* is a case where $\|\mathcal{F}\| = 0$.

Greedy Prefetching

In a k-ary RDS, each node contains k pointers to other nodes. Greedy prefetching exploits the fact that when $k > 1$, only one of these k pointers can be immediately followed by control flow as the next node in the traversal. Hence the remaining $k - 1$ pointers serve as natural jump-pointers, and can be prefetched immediately upon first visiting a node. Although none of these jump-pointers may actually point to n_{i+d}, hopefully each of them points to $n_{i+d'}$ for some $d' > 0$. If $d' < d$, then the latency may be partially hidden; if $d' \geq d$, then we expect the latency to be fully hidden, provided that the node is not displaced from the cache before it is referenced (which may occur if $d' \gg d$).

To illustrate how greedy prefetching works, consider the pre-order traversal of a binary tree (i.e. $k = 2$), where Figure 3.3(a) shows the code with greedy prefetching added. Assuming that the computation in `process()` takes half as long as the cache miss la-

[3]We may also need to take other information into account, such as the traversal order, but nothing can involve dereferencing a pointer within n_i.

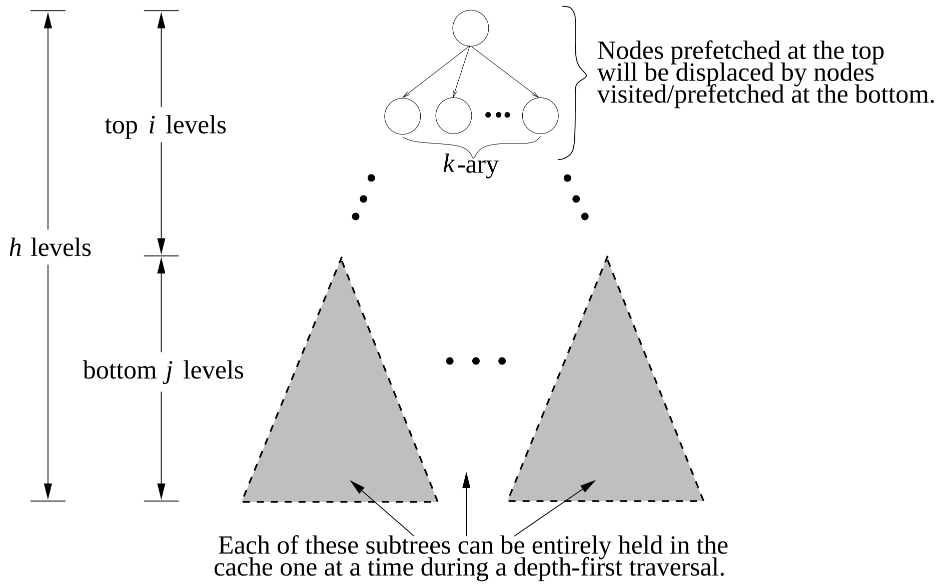


Figure 3.4: A complete k-ary tree with h levels.

tency, we would want to prefetch two nodes ahead (i.e. $d = 2$) to fully hide the latency. Figure 3.3(b) shows the caching behavior of each node. We obviously suffer a full cache miss at the root node (node 1), since there was no opportunity to fetch it ahead of time. However, we would only suffer half of the miss penalty ($\frac{L}{2}$) when we visit node 2, and no miss penalty when we eventually visit node 3 (since the time to visit the subtree rooted at node 2 is greater than L). In this example, the latency is fully hidden for roughly half of the nodes, and reduced by 50% for the other half (minus the root node). If we generalize this example to a k-ary tree, we would expect the fraction of nodes where latency is fully hidden to be roughly $\frac{k-1}{k}$ (assuming that prefetched nodes are generally not displaced from the cache before they are referenced). Hence a larger value of k is likely to improve the performance of greedy prefetching, since more natural jump-pointers are available.

The validity of the above example relies on three assumptions: (i) no conflict misses, (ii) the cache is large enough to hold the whole tree, and (iii) every node of the tree will be visited. Assumption (i) can be realized by having an associative cache and/or a victim buffer. Assumption (ii) is in fact not necessary for greedy prefetching performing well. As long as the cache size is significantly larger than the node size (not the size of the whole RDS), capacity misses would not be a problem. To support this argument, consider the complete k-ary ($k > 1$) tree T shown in Figure 3.4 which has t nodes and h levels ($t = \frac{k^h-1}{k-1}$). Suppose that the cache is just large enough to hold a complete k-ary tree with c nodes and j levels ($c = \frac{k^j-1}{k-1}$). If we traverse T in a depth-first order, then every node in the bottom j levels of T that we prefetch along with the traversal can be kept in the cache until it is actually used. In contrast, nodes in the top $i = h - j$ levels of T that we prefetched before visiting the bottom of T will be displaced by the time that they are actually used. Since there are $m = \frac{k^i-1}{k-1}$ such nodes, the proportion of prefetched nodes of T that are *not* displaced before their use is p where

$$
\begin{aligned}
p &= 1 - \frac{m}{t} \\
&= 1 - \frac{k^i - 1}{(k-1)t} && (\text{since } m = \tfrac{k^i-1}{k-1}) \\
&= 1 - \frac{k^{h-j} - 1}{(k-1)t} && (\text{since } i = h - j) \\
&= 1 - \frac{\frac{t(k-1)+1}{c(k-1)+1} - 1}{(k-1)t} && (\text{since } k^h = t(k-1)+1,\ k^j = c(k-1)+1) \\
&= 1 - \frac{1 - \frac{c}{t}}{1 + c(k-1)} && (\text{algebraic simplification}) \\
&= \left[\frac{1 + t(k-1)}{1 + c(k-1)}\right]\frac{c}{t} && (\text{algebraic simplification}) \qquad (3.1)
\end{aligned}
$$

Therefore, for any reasonable c (say $c > 100$), p would be close to 1 meaning that for most parts of the tree, any greedily prefetched items would be able to stay in the cache long enough for future use. Assumption (iii) determines whether the prefetches launched at a node is useful. The worst case for a node is that none of the k pointers prefetched is followed. In this situation, the k prefetches are wasteful. Thus, the effectiveness of greedy prefetching depends on the ratio of the number of nodes visited to the number of nodes prefetched.

So far, we have been assuming depth-first traversals for RDSs, which appear to be more common than breath-first traversals in general. Compared against depth-first traversals, breath-first traversals would have fewer intervening nodes between the current node and those that are greedily prefetched. This has two implications to the performance of greedy prefetching. The upside is that it is highly unlikely that prefetched nodes will be displaced by others before they are used. However, the downside is that there would be less computation to overlap with the prefetch latency.

In general, greedy prefetching offers the following advantages: (i) it requires no additional storage or computation to construct the natural jump-pointers; (ii) it is applicable to a wide variety of RDSs, regardless of how they are accessed or whether their structure is modified frequently; and (iii) it is relatively straightforward to implement in a compiler. A potential shortcoming of greedy prefetching is that it does not offer precise control over the prefetching distance, which is the motivation for our next algorithm.

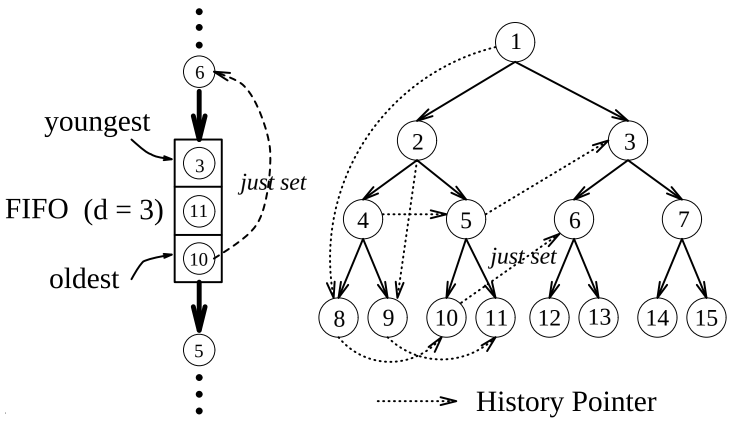


Figure 3.5: Example showing the update of history-pointers.

History-Pointer Prefetching

Rather than relying on natural jump-pointers to approximate A_{i+d}, we can potentially synthesize more accurate jump-pointers based on the actual RDS traversal patterns, while still achieving $\|\mathcal{F}\| = 1$. The idea behind the *history-pointer prefetching* scheme is that we create a new jump-pointer (called a *history-pointer*) in n_i which contains the observed value of A_{i+d} during a recent traversal of the RDS. (Note that we could potentially store multiple artificial jump-pointers in n_i to account for multiple traversal orderings.) On subsequent traversals of the RDS, we prefetch the nodes pointed to by these history-pointers. This scheme is most effective when the traversal pattern does not change rapidly over time, in which case the history-pointer in n_i is likely to point to either n_{i+d} or else hopefully a node that will be visited soon. On the other hand, if the structure of the RDS changes radically between traversals, the history-pointers might not be effective.

To construct the history-pointers, we maintain a FIFO queue of length d which contains pointers to the last d nodes that have just been visited. When we visit a new node n_i, the oldest node in the queue will be n_{i-d} (i.e. the node visited d nodes earlier), and hence we update the history-pointer of n_{i-d} to point to n_i. After the first complete traversal of the RDS, all of the history-pointers will be set. Figure 3.5 illustrates a snapshot of this bookkeeping process for the tree shown earlier in Figure 3.3. Assuming that $d = 3$ and that we have just reached node 6, we would now update the history-pointer of the oldest node in the 3-entry queue (node 10) to point to node 6.

Comparing the performance of this scheme with greedy prefetching, history-pointer prefetching offers no improvement on the first traversal of an RDS, since the history-

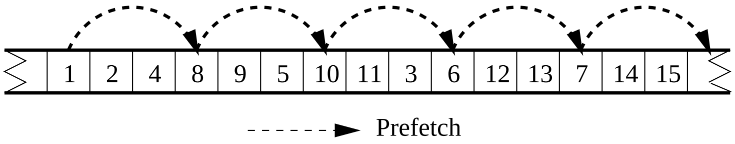


Figure 3.6: Illustration of data-linearization prefetching.

pointers have yet to be set (greedy prefetching would hide some fraction of the latency).[4] However, on subsequent traversals of the RDS, history-pointer prefetching will hide nearly all of the latency, whereas greedy prefetching will continue to hide only a fraction of the latency.

While history-pointer prefetching offers the potential advantage of improved latency tolerance, this comes at the expense of two additional forms of overhead: (i) execution overhead to construct the history-pointers, and (ii) space overhead for storing these new pointers. To minimize execution overhead, we can potentially update the history-pointers less frequently, depending on how rapidly the structure of the RDS changes. In one extreme, if the RDS never changes, we only need to set the history-pointers once. The problem with space overhead is that it potentially worsens the caching behavior. The desire to eliminate this space overhead altogether is the motivation for our next prefetching scheme.

Data-Linearization Prefetching

The goal of *data-linearization prefetching* is to achieve an $\mathcal{F}$ such that A_{i+d} can be predicted precisely, but without requiring any pointer dereferences (i.e. $\|\mathcal{F}\| = 0$). Another advantage of this scheme is that it improves spatial locality. The basic idea is to map heap-allocated nodes that are likely to be accessed close together in time into contiguous memory locations. With this mapping, one can easily predict A_{i+d} and hence prefetch it early enough.

Recall that the address of an array element $x[i+d]$ can be computed relative to $x[i]$ as follows (using C-like syntax):

$$\&x[i+d] = \&x[i] + d \times \texttt{sizeof}(x[0]) \tag{3.2}$$

Therefore, if we can map the RDS onto an array x of nodes such that $A_i = \&x[i]$, no pointer dereference is needed to compute A_{i+d}—we simply need two arithmetic operations per prefetch address.

[4]Hence we may want to use greedy prefetching for the first traversal of an RDS when the history-pointers are being initialized.

The question is how and when can this mapping (which we call *data linearization*) be performed? In theory, one could dynamically remap the data even after the RDS has been initially constructed, but doing so may violate program semantics.[5] Unfortunately, current compiler technology is very limited in ensuring program correctness across dynamic relocation of RDSs. In Chapter 4, we will study a hardware technique called *memory forwarding* as a means to provide this correctness guarantee. But without such a technique, the best time to map the nodes would be at creation time, which is appropriate if either the creation order already matches the traversal order, or if it can be safely reordered to do so. Since dynamic remapping is expensive (even if correctness is not an issue), this scheme obviously works best if the structure of the RDS changes only slowly (or not at all). If the RDS does change radically, the program will still behave correctly, but prefetching will not improve (or may even degrade) performance.

Figure 3.6 illustrates how data-linearization prefetching works for the tree shown earlier in Figures 3.3 and 3.5. The ordering of nodes in the array corresponds to the pre-order traversal order. To prefetch d nodes ahead, one simply uses equation (3.2) to compute A_{i+d}. In addition, if a single cache line can hold $m > 1$ nodes, we can exploit this spatial locality by only issuing a prefetch once every m nodes. If we are traversing the RDS inside a loop, we can accomplish this by unrolling the loop by a factor of m (similar to what is done in array-based prefetching [95]). For a traversal through recursion, one could potentially keep track of the number of nodes visited between prefetches, but the overhead of doing so may be comparable to simply issuing a prefetch for every node. The arrows in Figure 3.6 indicate the desired prefetches when $d = 3$ and $m = 3$.

3.2.3 Summary

The nature of recursive data structures makes both the analysis and scheduling of prefetches quite challenging. Before attempting to minimize prefetching overhead through improved analysis, we must first maximize the latency-hiding gain through effective prefetch scheduling techniques. The fundamental problem in scheduling prefetches for RDSs is the *pointer-chasing problem*, which we formalize as the number of pointer-chain dereferences required to compute a prefetch address ($\|\mathcal{F}\|$). Based on our desire to minimize $\|\mathcal{F}\|$, we have identified three promising prefetching schemes: *greedy prefetching* ($\|\mathcal{F}\| = 1$ with natural jump-pointers), *history-pointer prefetching* ($\|\mathcal{F}\| = 1$ with artificial jump-pointers), and *data-linearization prefetching* ($\|\mathcal{F}\| = 0$). Table 3.1 compares them from different perspectives.

[5] All pointers to these objects would also need to be updated, and understanding pointer aliasing for heap-allocated objects is quite difficult for the compiler.

Table 3.1: Summary of our three RDS prefetching schemes.

	Greedy	History-Pointer	Data-Linearization
Control over prefetching distance	Little	More precise	
Applicability to RDSs	Any	Those that are revisited and change only slowly	Those that have a major traversal order and change only slowly
Overhead in preparing prefetch addresses	None	Space and time for constructing history pointers	Time for dynamic remapping (if any)

Of these three schemes, greedy prefetching is perhaps the most widely applicable since it does not rely on traversal history information, and it requires no additional storage or computation to construct prefetch addresses. However, the other two schemes could outperform greedy prefetching when their favorable conditions are met. To evaluate them quantitatively, we have implemented these schemes as an automatic compiler pass and applied them to a set of RDS-intensive programs. In the next section, we describe the implementation details of our RDS prefetching compiler pass.

3.3 Implementation of RDS Prefetching Schemes

Our implementation of the three prefetching schemes within the SUIF compiler [133] consists of an *analysis* phase to recognize RDS accesses, and a *scheduling* phase to insert prefetches. The same analysis phase is shared by the three schemes while each of them has a slightly different scheduling phase corresponding to what should be prefetched in that scheme.

3.3.1 Analysis: Recognizing RDS Accesses

To recognize RDS accesses, the compiler uses both *type declaration* information to recognize which data objects are RDSs, and *control structure* information to recognize when these objects are being traversed. The algorithm for identifying RDS types is shown in Figure 3.7. In this algorithm, an RDS type is a record type r containing at least one pointer or an array of pointers that point either directly or indirectly to a record type s. (Note that r and s are not restricted to be the same type, since RDSs may be comprised of heterogeneous nodes.) For example, the type declarations in Figure 3.8(a) and

```
// This algorithm returns whether the given type T is an RDS type.
algorithm RDS_Type
        (T: type_definition)
    return (boolean); // True if T is an RDS type.

  if (T is a structure or union type) then // T can't be an RDS type if it isn't a structure or union.
    foreach field type F of T do // consider each field of T
      if ((F is a pointer) or (F is an array of pointers)) then
        // T is an RDS type only if F is a pointer or an array of pointers.
        do // loop over to find the non-pointer base type of F
          if ((F is a pointer to a type S) or (F is an array of type S)) then
            F := S; // consider the next level of indirection
          end if;
        while (F is a pointer type);
        if (F is a structure or union type) then
          // T is an RDS since it contains at least one pointer that points
          // directly or indirectly to a structure or union.
          return True;
        end if;
      end if;
    end foreach;
  end if;
  // not recognized as an RDS type if we reach here
  return False;
end algorithm;
```

Figure 3.7: Algorithm for identifying RDS types.

(a) RDS type

```
struct T {
  int data;
  struct T *left;
  struct T *right;
}
```

(b) RDS type

```
struct A {
  int i;
  struct B **kids[8];
}
```

(c) Not RDS type

```
struct C {
  int j;
  double f;
}
```

Figure 3.8: Examples of whether type declarations are recognized as being RDS types.

Figure 3.8(b) would be recognized as RDS types, whereas Figure 3.8(c) would not.[6]

After discovering data structures with the appropriate types, the compiler then looks for control structures that are used to traverse the RDSs, using the algorithm `recognize_RDS_accesses` shown in Figure 3.9. In particular, the compiler looks for *loops* or *recursive procedure calls* such that during each new loop iteration or procedure invocation, a pointer p to an RDS is assigned a value resulting from a dereference of p—we refer to this as a *recurrent pointer update*. This heuristic corresponds to how RDS codes are typically written.

[6] The compiler may fail to recognize cases with explicit type casting—e.g., casting `j` to be of type `(struct C*)` in Figure 3.8(c)—but such cases do not appear to be common.

```
// This algorithm recognizes all RDS accesses within the given control-flow construct C.
// When such an access is found, schedule_prefetches() is invoked to insert prefetches.
algorithm recognize_RDS_accesses
        (C: control_flow_construct)
    return (); // nothing to return

  switch (C.kind)
    case assignment: // C is an assignment statement
      l: loop := C.enclosing_loop; // l is the innermost loop that encloses C (if any)
      if (l) then // C is inside a loop.
        if (RDS_Ptr(C.lhs)) then // the left hand side of the assignment C is an RDS pointer
          if (is_recurrent(C.lhs, C.rhs)) then // C.rhs is recurrent w.r.t C.lhs
            // Insert prefetches for C.lhs inside loop l according to the
            // prefetching scheme PF_SCHEME.
            schedule_prefetches(PF_SCHEME, C.lhs, l);
          end if;
        end if;
      end if;
    end case;
    case functionCall: // C is a function call
      f: function := C.called_function; // f is the function called in C
      g: function := C.enclosing_function; // g is the function that contains C
      if (f == g) then // C is within a recursive function
        foreach formal argument a of g do // consider each of the formal arguments of g
          if (RDS_Ptr(a)) then // a is an RDS pointer
            foreach actual argument b of f do // consider each of the actual arguments of f
              if (is_recurrent(a, b)) then // b is recurrent w.r.t a
                // Insert prefetches for a inside function g according to the
                // prefetching scheme PF_SCHEME.
                schedule_prefetches(PF_SCHEME, a, g);
              end if;
            end foreach;
          end if;
        end foreach;
      end if;
    end case;
    case if: // C is an "if" statement
      recognize_RDS_accesses(C.if_condition);
      recognize_RDS_accesses(C.if_then);
      recognize_RDS_accesses(C.if_else);
    end case;
    case loop: // C is a "loop" statement
      recognize_RDS_accesses(C.loop_condition);
      recognize_RDS_accesses(C.loop_body);
    end case;
    case block: // C is a list of statement
      foreach statement i of C do
        recognize_RDS_accesses(i);
      end foreach;
    end case;
  end switch;
end algorithm;
```

Figure 3.9: Algorithm for recognizing RDS accesses.

```
// This algorithm propagates values of pointers to RDSs across the given control-flow construct C.
// The most recent pointer values are kept in the table T of pointer_value_table_type, which is a
// table of tuples. Each tuple <p, {v1, ..., vn}> denotes the fact that the pointer variable p has
// a set of possible values v1, ..., vn. In practice, propagate_RDS_pointers is incorporated
// into recognize_RDS_accesses so that we only need to keep track of the most recent pointer values.
// If propagate_RDS_pointers and recognize_RDS_accesses are performed separately, we would
// need to bookkeep pointer values at all possible program locations. However, they are presented here as two
// separate passes for the sake of clarity.
algorithm propagate_RDS_pointers
        (C: control_flow_construct)
        (T: pointer_value_table_type)
    return (); // nothing to return

  switch (C.kind)
    case assignment: // C is an assignment statement
      if (RDS_Ptr(C.lhs)) then // the left hand side of the assignment C is an RDS pointer
        assign_pointer(C, T); // update the value of the assigned variable in T
      end if;
    end case;
    case if: // C is an "if" statement
      propagate_RDS_pointers(C.if_condition, T);
      // initialize pointer values for both paths
      T_then: pointer_value_table_type := T;
      T_else: pointer_value_table_type := T;
      propagate_RDS_pointers(C.if_then, T_then);
      propagate_RDS_pointers(C.if_else, T_else);
      T := union(T_then, T_else); // combine the pointer values of both paths
    end case;
    case loop: // C is a "loop" statement
      propagate_RDS_pointers(C.loop_condition);
      propagate_RDS_pointers(C.loop_body);
      // Propagate one more time to account for any pointer updates of current iteration
      // that could affect the pointer values of next iteration.
      propagate_RDS_pointers(C.loop_condition);
      propagate_RDS_pointers(C.loop_body);
    end case;
    case block: // C is a list of statement
      foreach statement i of C do
        propagate_RDS_pointers(i);
      end foreach;
    end case;
  end switch;
end algorithm;
```

Figure 3.10: Algorithm for propagating RDS pointer values.

In `recognize_RDS_accesses`, the `assignment` and `functionCall` cases check for such pointer updates. If any of them is found, prefetches will be inserted into the loop or procedure body according to the given prefetching scheme. The remaining three cases in `recognize_RDS_accesses` are simply for covering other control-flow constructs.

To detect recurrent pointer updates, the compiler propagates pointer values using

```
// This algorithm updates in T the pointer on the left hand side (A.lhs) of assignment A by the
// value of its right hand side (A.rhs). Symbolic values are created when necessarily and the
// program locations where these values are first available (within the enclosing function) are also
// recorded in T. This information helps insert prefetches at the earliest place that the pointer
// value is available.
algorithm assign_pointer
            (A: assignment_statement)
            (T: pointer_value_table_type)
      return (); // nothing to return

   if (A.rhs is a variable p) then
      V: set of pointer_values := T.lookup(p); // V is the set of possible current values of p
      if (V ≠ {}) then
         // p already has some possible values recorded in T
         T.replace(A.lhs, V); // V becomes the set of possible current values of A.lhs.
      else // p's possible values haven't been recorded in T
         // Create a symbolic value of p to denote the very first value of p within the enclosing function.
         // A.enclosing_function.first_instruction.location is the location of the first instruction
         // of the enclosing function, the earliest place that the value is available.
         u := T.create_symbolic_value(p, A.enclosing_function.first_instruction.location);
         T.replace(A.lhs, {u}); // {u} becomes the set of possible current values of A.lhs
      end if;
   else if (A.rhs is a pointer dereference p → f) then
      V: set of pointer_values := T.lookup(p); // V is the set of possible current values of p
      if (V ≠ {}) then
         // p already has some possible values recorded in T
         W: set of pointer_values := {}; // W will be the set of possible current values of p → f.
         foreach u ∈ V do
            // construct W from V
            W := W ∪ {u → f};
         end foreach;
      else // p's possible values haven't been recorded in T
         u := T.create_symbolic_value(p, A.enclosing_function.first_instruction.location);
         T.replace(A.lhs, {u → f});
      end if;
   else if (A.rhs is a constant pointer value k) then
      // k is most likely the NULL pointer
      T.replace(A.lhs, k);
   else
      // cover all other cases including something like p = f(...)
      u := T.create_symbolic_value(A.lhs, A.location); // u is available only after A in the program
      T.replace(A.lhs, {u});
   end if;
end algorithm;
```

Figure 3.11: Algorithm for assigning new values to RDS pointers.

the algorithms `propagate_RDS_pointers` and `assign_pointer` shown in Figure 3.10 and Figure 3.11, respectively. The data structure central to these two algorithms is a table of `pointer_value_table_type` that keeps track of the most recent possible values of each RDS pointer variable. Algorithm `propagate_RDS_pointers` propagates RDS pointer val-

```
(a)  while (l) {                 (b)  for (...)  {
       listNode *m;                     listNode *n;
       ...                              ...
       m = l→next;                      n = g(n);
       l = m→next;                      ...
       ...                            }
     }

(c)  f(treeNode *t) {            (d)  k(treeNode tn) {
       ...                              ...
       f(t→left);                       k(*(tn.left));
       f(t→right);                      k(*(tn.right));
       ...                              ...
     }                                }
```

Figure 3.12: Examples of recognizable control structures for RDS traversals.

ues intra-procedurally for each function in the program as follows: it propagates pointer values along both paths of an "if" statement and combines them when the two paths rejoin; it propagates pointer values over loop iterations by essentially unrolling the loop once; when it encounters an assignment to an RDS pointer, it invokes `assign_pointer` to perform the update in the `pointer_value_table_type` table.

To update RDS pointers, algorithm `assign_pointer` first looks for any pointer variable p or pointer dereference $p \rightarrow f$ appearing on the right hand side of the given assignment statement. Then it checks whether p already has a set of possible values recorded in the table. If such a set is found, it will be used in place of p on the right hand side to update the left hand side of the assignment; otherwise a new symbolic value for p will be created by calling `create_symbolic_value`. In any case, the updated value of the left hand side is recorded in the table so that it will be observed by later assignments. In addition to pointer values, program locations where new pointer values first become available in procedures are also remembered (the program location is passed as the second argument of `create_symbolic_value`). This piece of information facilitates the insertion of prefetches at the earliest place that a pointer value is known.

To make our discussion more concrete, Figure 3.12 shows some example program fragments that our compiler treats as RDS accesses. In Figure 3.12(a), `l` is updated to `l→next→next` inside the while-loop. In Figure 3.12(b), `n` is assigned the result of the function call `g(n)` inside the for-loop. (Since our implementation does not perform interprocedural analysis, it assumes that `g(n)` results in a value `n→...→next`.) In Figure 3.12(c), two dereferences of the function argument `t` are passed as the parameters to two recursive calls. Figure 3.12(d) is similar to Figure 3.12(c), except that a record (rather than a pointer) is passed as the function argument.

Ideally, the next step would be to analyze data locality across RDS nodes—e.g., to distinguish the two cases shown in Figure 3.1—to eliminate any unnecessary prefetches. Although we have not automated this step in our compiler, we will evaluate its potential benefit later in Section 3.5.5.

3.3.2 Scheduling Prefetches

At the point where an RDS object is being traversed (i.e. where the recurrent pointer update occurs), the compiler invokes `schedule_prefetches` (refer back to Figure 3.9) to insert prefetches as follows, according to the given prefetching scheme:

Greedy Prefetches

The compiler inserts prefetches of all pointers within the RDS object that point to RDS-type objects (these are the natural jump-pointers[7]). In most situations, prefetches are inserted at the earliest points where these addresses are available within the surrounding loop or procedure body. However, if the compiler estimates that the overhead of greedy prefetching is relatively high, it will insert prefetches at later points where the prefetches launched are more likely to be useful. Our simple heuristic is that if there are more than 10 prefetch instructions at each site, the compiler will insert them *after* the conditional test (if any) that decides whether the recursion or iteration will continue. Three examples of greedy prefetch scheduling are shown in Figure 3.13.

History-Pointer Prefetches

Prefetches of all history-pointers within the RDS object are inserted at the earliest possible points. Besides, the compiler attaches codes for updating history-pointers to the surrounding loop or procedure body. Recall from Section 3.2.2 that a FIFO queue of length d (the prefetching distance) is required to record the traversal order of the RDS. Our compiler provides two implementations for this FIFO. In the what we call *array-FIFO* implementation, the FIFO is implemented as an array of d pointers with the oldest element indexed by a variable named `oldest` (the index to the youngest element can be deduced from `oldest`). Wrap-around increment of `oldest` can be done either through the modulus operator or through a conditional branch that tests for the array

[7]Note that we do not prefetch *all* pointers within an RDS object—only the ones that point to other RDS nodes (potentially of different types than the given object).

(a) Loop (*prefetches inserted at the earliest point*)

```
while (l) {                    while (l) {
  work(l→data);       ⟹         prefetch(l→next);
  l = l→next;                    work(l→data);
}                                l = l→next;
                               }
```

(b) Procedure (*prefetches inserted at the earliest point*)

```
f(treeNode *t) {               f(treeNode *t) {
  treeNode *q;        ⟹         treeNode *q;
  if (test(t→data))              prefetch(t→left);
    q = t→left;                  prefetch(t→right);
  else                           if (test(t→data))
    q = t→right;                   q = t→left;
  if (q != NULL)                 else
    f(q);                          q = t→right;
}                                if (q != NULL)
                                   f(q);
                               }
```

(c) Procedure (*prefetches inserted at a later point where the recursion is known to continue*)

```
g(bigTreeNode *t) {            g(bigTreeNode *t) {
  if (test(t→data)) {   ⟹       /* prefetches are not inserted here */
    for (i = 0 .. 15) {          if (test(t→data)) {
      g(t→child[i]);               prefetch(t→child[0]);
    }                              ...
  }                                prefetch(t→child[15]);
}                                  for (i = 0 .. 15) {
                                     g(t→child[i]);
                                   }
                                 }
                               }
```

Figure 3.13: Examples of greedy prefetch scheduling.

(a) Original loop

```
while (p) {
  work(p→data);
  p = p→next;
}
```

(b) The loop in (a) with history-pointer prefetching code added, using the *array-FIFO* implementation.

```
while (p) {
  prefetch(p→history_pointer);
  work(p→data);
  p = p→next;
  /* Q is the array that implements the FIFO queue */
  Q[oldest]→history_pointer = p; /* set the history pointer of */
                                 /* the oldest node to point to p */
  Q[oldest] = p; /* free up the oldest location for p, the */
                 /* most recently visited node */
  if (oldest == d-1) {
    oldest = 0; /* wrap around */
  } else {
    ++oldest;
  }
}
```

(c) The loop in (a) with history-pointer prefetching code added, using the *scalar-FIFO* implementation.

```
while (p) {
  prefetch(p→history_pointer);
  work(p→data);
  p = p→next;
  /* Q_0, ..., Q_dminus1 are the d pointers that implement the FIFO queue */
  Q_0→history_pointer = p; /* set the history pointer of */
                           /* the oldest node to point to p */
  Q_0 = Q_1; /* retire the oldest node to accommodate the youngest one */
  Q_1 = Q_2;
  ...
  Q_dminus2 = Q_dminus1;
  Q_dminus1 = p; /* Q_dminus1 always hold the youngest node */
}
```

Figure 3.14: Examples illustrating two possible implementations of history-pointer prefetching.

boundary. The latter approach was chosen in our implementation since the branch is very predictable with our branch predictor[8], whereas modulus is an expensive operation (which lasts for 76 cycles) for the processor model used in our experiments (details will be provided later in Section 3.4). In the what we call *scalar-FIFO* implementation, the FIFO is programmed as d individual pointer variables, say Q_0, Q_1, ... Q_{d-1}. The oldest element of the FIFO is always stored in Q_0 while the youngest is always stored in Q_{d-1} and so on. Therefore we need to copy Q_i to Q_{i-1} for i from 1 to $d-1$ upon accepting a new node. Examples of these two implementations including prefetches of history-pointers are shown in Figure 3.14. Comparing these two implementations, the instruction overhead of maintaining the FIFO is constant in the array-FIFO implementation but increases with d in the scalar-FIFO implementation. For this reason, the compiler picks either one of these implementations based on the size of d: when d is smaller than a threshold d_t, it uses the scalar-FIFO implementation for its lower overhead; but it uses the array-FIFO implementation when $d \geq d_t$. In our experimental environment, we found that $d_t = 8$ is a good choice for most cases. Relevant experimental results will be shown later in Section 3.5.3.

Data-Linearization Prefetches

Our current implementation of data-linearization prefetching does not perform data relocation because the compiler cannot guarantee that it is always safe to do so. Instead, we will perform data-linearization prefetching along with dynamic data relocation later in Chapter 4 after introducing memory forwarding. For now, the compiler simply inserts data-linearization prefetches with a hope that the creation order of the RDS mimics the major traversal order.

Prefetch Size

For all the three prefetching schemes, the compiler prefetches two cache lines by default. Since block prefetch instructions are not assumed in our underlying architecture, the compiler actually inserts multiple prefetch instructions to bring in multiple lines.

3.3.3 A SUIF Implementation

We implemented our prefetching schemes in the SUIF compiler system [133] version 1.1.0, which generates optimized MIPS assembly. The resultant assembly file is then fed into the

[8] The outcomes of this branch are always $d-1$ not-takens followed by one taken, which are well predicted by local branch predictors for reasonable values of d. Our tournament branch predictor does comprise a local branch predictor.

Table 3.2: Order in which the compiler passes (including prefetching) are invoked.

Compiler Pass	Description
cpp	C preprocessor
snoot	convert C into non-standard SUIF
porky -defaults	produce standard SUIF
porky -ucf-opt	simple optimizations on unstructured control flow
porky -dead-code	dead-code elimination
structure	build structured control flow out of unstructured control flow
porky -fold	constant folding
porky -find-fors	find “for” loops
porky -const-prop	constant propagation
porky -fold	constant folding
porky -copy-prop	copy propagation
porky -dead-code	dead-code elimination
porky -unused-syms -unused-types	remove symbols and types that are never referenced
porky -no-empty-table	dismantle all blocks that have empty symbol tables
porky -no-empty-fors	dismantle “for” loops with empty bodies
porky -control-simp	control simplification
porky -forward-prop	forward propagation
porky -cse	common sub-expression elimination
porky -unused-syms -unused-types	remove symbols and types that are never referenced
porky -const-prop	constant propagation
porky -ivar	induction variable detection
porky -know-bounds	replace comparisons of upper and lower bounds of a loop inside the loop body with the known result of that comparison
porky -const-prop	constant propagation
porky -fold	constant folding
porky -no-empty-fors	dismantle “for” loops with empty bodies
rds_pf	**perform prefetching analysis and insert prefetches for RDSs**
porky -Darrays -Dblocks -Dfors -Difs -Dloops -Dfcmmas -no-call-expr -reassociate	dismantle SUIF array instructions dismantle SUIF “block” constructs dismantle SUIF “for” constructs dismantle SUIF “if” constructs dismantle SUIF “loop” constructs dismantle SUIF divfloor, divceil, min, max, abs, and mod instructions replace calls within expressions by local variables array reassociation
swighnflew	prepare a SUIF file for mexp and mgen
porky -no-index-spill	dismantle SUIF “for” constructs with spilled index variable
oldsuif	translate from new SUIF to old SUIF
oynk -Pconst	constant propagation
oynk -Psr	strength reduction
oynk -Pconst -Fsu	constant propagation
oynk -Pdstore	dead store elimination
mexp	massage SUIF constructs to MIPS-palatable form
oynk -Preg	register allocation
mgen	generate MIPS assembly code
as	generate MIPS object code

native assembler on IRIX 5.3 to produce the object code. In addition to many standard optimization passes, SUIF provides a set of library routines to manipulate entities such as types, symbols, instructions, expressions, etc. that are used throughout the compilation process. By leveraging these routines, coding our algorithms becomes relatively easy. For example, it is straightforward in SUIF to obtain type-related information such as the type of the objects that a pointer type points to, or the types of fields within a structure type. This greatly simplified the task of implementing the algorithm for identifying RDS types (shown earlier in Figure 3.7).

Our prefetching schemes were implemented as a new SUIF pass, which accepts a SUIF file from the previous pass and outputs another SUIF file with prefetches inserted. The position of our prefetching pass relative to other passes in the entire compilation process is shown in Table 3.2. We adopted the same strategy used by Mowry [95] as to where the prefetching pass should occur in the compilation process. His strategy suggests that the prefetching pass should occur *before* the pass that decomposes high-level SUIF constructs such as "loops" into low-level SUIF constructs such as branch instructions (which is the pass right after **rds_pf** in Table 3.2). The main advantage of this is that all high-level SUIF constructs are available to the prefetching pass. For example, our algorithm for recognizing RDS accesses in Figure 3.9 needs to know the innermost surrounding loop of a recurrent pointer update, which is much easier to compute with high-level SUIF than with low-level SUIF. Mowry also suggested that the prefetching pass would better be placed before scalar optimization passes (those with "oynk" in Table 3.2) in order to leverage these optimizations to minimize instruction overhead of prefetching code inserted. By doing this, neither do we need to generate highly-optimized prefetching code by the prefetching pass itself nor re-run time-consuming scalar optimizations after the prefetching pass.

3.4 Experimental Framework

To evaluate the performance of our three prefetching schemes, we performed detailed cycle-by-cycle simulations of the entire Olden benchmark suite [109] on a dynamically-scheduled, superscalar processor. The Olden benchmark suite contains ten pointer-based applications written in C, which are briefly summarized in Table 3.3. The rightmost column in Table 3.3 shows the number and size of each node type that was dynamically allocated. In addition, some relevant run-time statistics are provided in Table 3.4.

Table 3.5 shows the parameters of the baseline model used by the experiments in this chapter. These parameters are mainly derived from the MIPS R10000 [139], with vari-

Table 3.3: Benchmark characteristics.

Benchmark	Description	Recursive Data Structures Used	Input Data Set	Node Memory Allocated
BH	Barnes-Hut's N-body force-calculation algorithm	Heterogeneous octree	4K bodies	4128 x 136 B = 548 KB + 2021 x 88 B = 173 KB
Bisort	Sorts two disjoint bitonic sequences and then merges them	Binary tree	250,000 integers	131,017 x 12 B = 1535 KB
EM3D	Simulates the propagation of electromagnetic waves in a 3D object	Singly-linked lists	2000 H-nodes, 100 E-nodes, 75% local	4000 x 28 B = 109 KB + 400,000 x 4 B = 1562 KB
Health	Simulation of the Columbian health care system	Four-way tree and doubly-linked lists	max. level = 5, max. time = 500	341 x 100 B = 33 KB + 57,111 x 16 B = 892 KB
MST	Finds the minimum spanning tree of a graph	Array of singly -linked lists	1024 nodes	1024 x 20 B = 20 KB
Perimeter	Computes perimeters of regions in images	A quadtree	4K x 4K image	235,717 x 28 B = 6445 KB
Power	Solves the power system optimization problem	Multi-way tree and singly-linked lists	10,000 customers	200 x 56 B = 11 KB + 1000 x 96 B = 94 KB + 10,000 x 32 B = 313 KB
TreeAdd	Sums the values distributed on a tree	Binary tree	1024K nodes	1,048,576 x 12 B = 12,288 KB
TSP	Traveling salesman problem	Binary tree and doubly-linked lists	100,000 cities	131,071 x 40 B = 5120 KB
Voronoi	Computes the voronoi diagram of a set of points	Binary tree	20,000 points	633,032 x 16 B = 9891 KB + 32,768 x 32 B = 1024 KB

Table 3.4: Run-time statistics. "Insts Grad." is the total instructions graduated. "Loads Grad." is the total loads graduated. The percentages of loads that were found in each of the four possible places are shown under "Where Loads Were Found", where "Combined" are loads that were combined with other in-flight references. "Average Load Miss Penalty" includes penalties of both full misses and partial misses.

Benchmark	Insts Grad.	Loads Grad.	Where Loads Were Found				Average Load Miss Penalty (cycles)
			D-Cache	Combined	S-Cache	Memory	
BH	1902 M	437 M	96.54%	1.53%	1.79%	0.13%	16.8
Bisort	668 M	132 M	95.21%	2.75%	1.55%	0.49%	27.4
EM3D	404 M	55 M	96.64%	0.91%	2.00%	0.45%	29.1
Health	209 M	58 M	69.38%	0.88%	13.54%	16.20%	46.1
MST	329 M	53 M	90.80%	2.64%	2.25%	4.31%	56.4
Perimeter	356 M	61 M	92.19%	4.80%	0.11%	2.90%	67.1
Power	2008 M	357 M	99.83%	0.10%	0.07%	0.00%	14.3
Treeadd	110 M	30 M	98.60%	0.09%	0.06%	1.25%	68.8
TSP	1085 M	145 M	97.62%	1.31%	0.65%	0.45%	39.3
Voronoi	249 M	81 M	98.37%	0.75%	0.73%	0.14%	22.3

ations corresponding to advancements in more recent processors (e.g., more functional units and a more accurate branch predictor). Our simulator is driven by *mable* [38], which performs instruction-by-instruction emulation of MIPS executables. An important advantage of emulation-driven simulations over trace-driven simulations that is particularly relevant to this study is the ease of supporting *non-excepting* memory operations [25]. Prefetch instructions are typically defined as non-excepting in the sense that they do not raise exceptions. This property enables the compiler to insert prefetches as early as it wants without causing exceptions that should not happen in the original execution. We implement non-excepting memory instructions in mable by specially handling all memory instructions (i.e. loads, stores, and prefetches) that access virtual page 0.

To minimize the impact of store stalls during the initialization of dynamically-allocated objects, we use our own memory allocator for these experiments which is similar to `mallopt` provided in the Irix C library [125], but also contains built-in prefetching to avoid such store misses. This optimization alone led to dramatic improvements (greater than two-fold speedups) over using `malloc` for some of the applications—particularly the ones that frequently allocate small objects.

Table 3.5: Baseline simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 2 Memory, 2 Branch
Reorder Buffer Size	64
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction Scheme	Tournament between a local and a global history predictors: local predictor: 4096 12-bit local history entries and 4096 2-bit saturation counters global predictor: 1 12-bit global history register and 4096 2-bit saturation counters choice predictor: 4096 2-bit saturation counters total size = 9.5 KB

Memory Parameters	
Line Size	32B
Primary Instruction Cache	16KB, 2-way set-associative random-replacement
Primary Data Cache	16KB, 2-way set-associative random-replacement; write-back write-allocate; 2 banks
Unified Secondary Cache	512KB, 2-way set-associative random-replacement; write-back write-allocate; 4 banks
Data Cache Miss Handlers	8
Data Cache Fill Time (Requires Exclusive Access)	4 cycles
Primary-to-Secondary Miss Latency	12 cycles (plus any delays due to contention)
Primary-to-Memory Miss Latency	75 cycles (plus any delays due to contention)
Primary-to-Secondary Bandwidth	16 bytes/cycle
Secondary-to-Memory Bandwidth	8 bytes/cycle

3.5 Experimental Results

We now present results from our simulation studies. We start by evaluating the performance of compiler-inserted greedy prefetching, and then compare this with compiler-inserted history-pointer prefetching and data-linearization prefetching. Next, we evaluate the potential performance gains from better analysis to reduce unnecessary prefetches. We then explore the performance impact of architectural support. Finally, we quantitatively compare the performance of our prefetching schemes with the only previously published compiler prefetching technique for pointer-based codes.

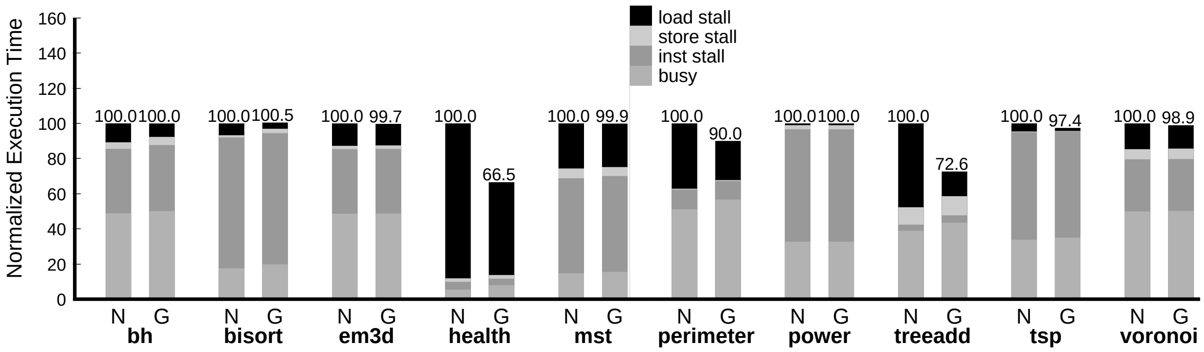


Figure 3.15: Performance of compiler-inserted greedy prefetching (**N** = no prefetching, **G** = greedy prefetching).

3.5.1 Performance of Compiler-Inserted Greedy Prefetching

The results of our first set of experiments are shown in Figure 3.15 and Table 3.6. Figure 3.15 shows the overall performance improvement offered by greedy prefetching, where the two bars correspond to the cases without prefetching (**N**) and with greedy prefetching (**G**). These bars represent execution time normalized to the case without prefetching, and they are broken down into four categories explaining what happened during all potential graduation slots.[9] The bottom section (*busy*) is the number of slots when instructions actually graduate, the top two sections are any non-graduating slots that are immediately caused by the oldest instruction suffering either a load or store miss,[10] and the *inst stall* section is all other slots where instructions do not graduate. Note that the *load stall* and *store stall* sections are only a first-order approximation of the performance loss due to cache stalls, since these delays also exacerbate subsequent data dependence stalls.

As we see in Figure 3.15, four applications enjoy a speedup ranging from 3% to 50% (the other six are within 2% of their original performance). For the applications with the largest memory stall penalties (i.e. `health`, `perimeter`, and `treeadd`), much of this stall time has been eliminated. Table 3.6 indicates that the load stall time is significantly reduced in most cases. This is accomplished by reducing the primary load miss rate, or the average load miss penalty, or both (which is the more common case). The miss penalty could be reduced even if the prefetch does not complete at a perfect timing. If a prefetch finishes so early that it is replaced from the primary cache before it can be referenced, it is still likely to be found in the secondary cache. On the other hand, if

[9]The number of graduation slots is the issue width (4 in this case) multiplied by the number of cycles. We focus on graduation rather than issue slots to avoid counting speculative operations that are squashed.

[10]Store misses only stall the processor when the 32-entry memory issue buffer is full.

Table 3.6: Memory performance improvement for compiler-inserted greedy prefetching.

	No Prefetch		Greedy Prefetch		
Benchmark	Load D-Cache Miss Rate	Average Load Miss Penalty (cycles)	Load D-Cache Miss Rate	Average Load Miss Penalty (cycles)	Load Stall Reduction
BH	3.46%	16.8	1.87%	21.9	28.0%
Bisort	4.79%	27.4	2.45%	27.1	39.1%
EM3D	3.36%	29.1	3.35%	28.7	2.0%
Health	30.62%	46.1	20.77%	39.0	30.0%
MST	9.20%	56.4	9.99%	62.3	-22.4%
Perimeter	7.81%	67.1	5.12%	44.8	51.0%
Power	0.17%	14.3	0.15%	14.8	14.0%
Treeadd	1.40%	68.8	0.97%	29.1	67.0%
TSP	2.38%	39.3	0.67%	27.2	80.0%
Voronoi	1.63%	22.3	1.57%	20.8	18.0%

a prefetch is late and the miss occurs while the prefetch access is still in progress, the miss latency can still be partially hidden. The only exception in Table 3.6 is `mst` where both the miss rate and average miss penalty are increased by greedy prefetching. The problem is memory contention, which was intensified when both prefetch and load misses for different data lines occurred in a very short period of time. As a result, substantial number of load misses were delayed by prefetches for a few cycles. Fortunately, memory contention is not a problem in all other applications, where 2% to 80% of the original load stall cycles are eliminated.

Turning our attention to the costs of greedy prefetching, Figure 3.15 shows that the instruction overhead of prefetching increases the sum of the *busy* and *inst stall* sections by less than 10% in eight of the benchmarks (there are 12% and 18% increases in `treeadd` and `health`, respectively). Only in the case of `bisort` does prefetching overhead more than offset the reduction in memory stalls, thereby resulting in a slight performance degradation. Later, in Section 3.5.5, we will explore how to reduce this overhead with the aid of profiling information.

To understand these performance results in greater depth, we study the *miss coverage*, *memory traffic*, and *instruction overhead* of greedy prefetching in the following subsections.

Miss Coverage

Figure 3.16 shows the number of load D-cache misses in the without prefetching and greedy prefetching cases, which are divided into three categories. A *partial miss* is a

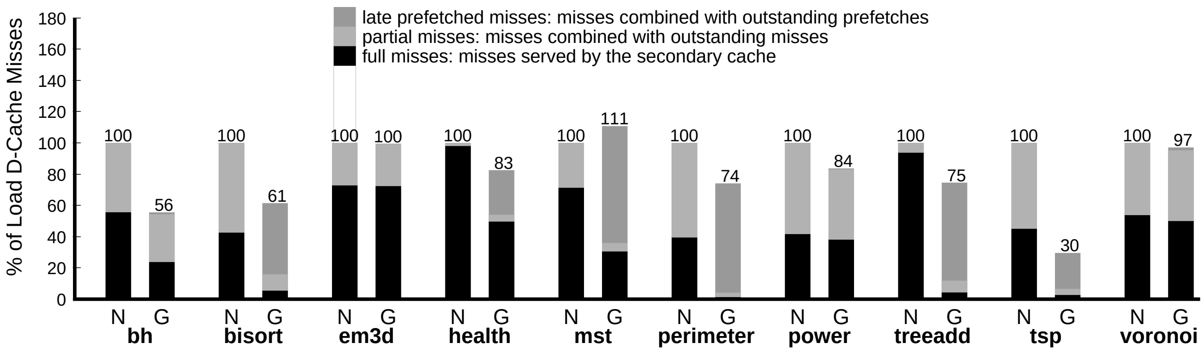


Figure 3.16: Breakdown of all load D-cache misses (**N** = no prefetching, **G** = greedy prefetching).

D-cache miss that combines with an outstanding miss to the same line, and therefore does not necessarily suffer the full miss latency. A *full miss*, on the other hand, does not combine with any access and therefore suffers the full latency. A *late prefetched miss* is a D-cache miss that combines with an outstanding prefetch (i.e. the prefetch was launched too late). If prefetching has perfect *miss coverage*, all of the full and partial misses would have been converted into hits (which do not appear in the figure) or at least into late prefetched misses. However, for `em3d`, `power`, and `voronoi`, more than 80% of the original misses are not prefetched because most of these misses are caused by array or scalar references—hence prefetching RDSs yields little improvement. For `bh` and `health`, 50%-60% of the original misses are not covered; we will take a closer look at the causes of these misses later in the case studies of these benchmarks. In the remaining five benchmarks, there are only less than 35% uncovered misses. Also, the *late prefetched misses* category indicates that 20%-80% of misses are prefetched late in six cases. This category is most prominent in `mst`, where the compiler is unable to prefetch early enough during the traversal of very short linked lists within a hash table. Since the natural jump-pointers in greedy prefetching offer little control over prefetching distance, it is not surprising that scheduling is imperfect. Also in `mst`, greedy prefetching generates 11% more load misses than the original case due to *cache pollution*. Fortunately, this is not a problem for the other nine benchmarks.

Memory Traffic

An important concern for any prefetching technique is how much extra memory traffic it will generate. Figure 3.17 shows the amount of traffic between the primary and secondary caches. Each bar in Figure 3.17 is divided into three sections, explaining if

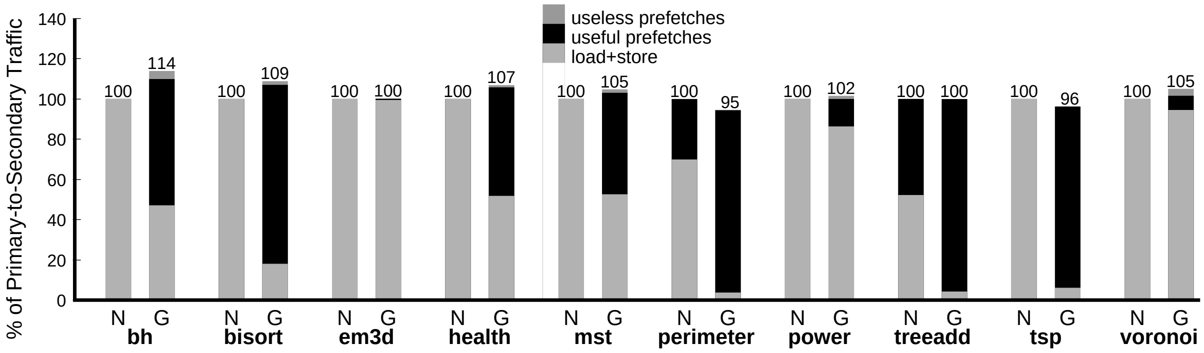


Figure 3.17: Breakdown of the traffic between the D-cache and secondary cache (**N** = no prefetching, **G** = greedy prefetching).

a transfer is triggered by a normal reference (*load+store*), or instead by a prefetch[11]. Prefetch transfers are further classified as *useful* or *useless*, depending on whether the prefetched data gets used by a load or store before it is displaced from the primary cache. Ideally, prefetching will not increase memory traffic, since the original memory references will simply be converted into prefetches. (In fact, previous studies have demonstrated that prefetching can actually *reduce* the memory traffic in a shared-memory multiprocessor through exclusive-mode hints [95].) However, since the natural jump-pointers used by greedy prefetching may point to nodes that will not be accessed in the near future (or perhaps not at all), greedy prefetching can potentially increase memory traffic through useless prefetches. Fortunately, as we see in Figure 3.17, greedy prefetching has increased memory traffic by at most 14% for all applications (in two cases—`perimeter` and `tsp`—the traffic actually decreased due to fortuitous cache replacement behavior in the set-associative caches). Hence greedy prefetching does not appear to be suffering from useless prefetches. Therefore, prefetch filtering mechanisms such as the one we used for instruction prefetching are not necessary here.

Instruction Overhead

Software prefetching has two kinds of instruction overhead: the instructions that generate prefetch addresses and the prefetch themselves. The amount of this overhead depends on the prefetching algorithm itself as well as on how optimized the prefetch codes that the compiler can generate. Table 3.7 shows the instruction overhead of greedy prefetching, which is divided into three categories: *Prefetches*, *Loads/Stores*, and *Others*

[11] Prefetches also exist in the **N** cases of `perimeter` and `treeadd` to cope with store stall, as we explained in Section 3.4.

Table 3.7: Instruction overhead of greedy prefetching.

Benchmark	Overall Overhead (percentage of original instruction count)				Per-Prefetch Overhead		
	Prefetches	Loads/ Stores	Others	Total	Loads/ Stores	Others	Total (including the prefetch itself)
BH	1.1%	1.1%	0.3%	2.5 %	1.0	0.3	2.3
Bisort	5.0%	6.2%	5.1%	16.4%	1.2	1.0	3.2
EM3D	0.01%	0.01%	0.01%	0.03%	0.9	1.1	3.0
Health	16.8%	9.2%	9.7%	35.7%	0.6	0.6	2.2
MST	3.3%	1.5%	2.6%	7.4%	0.5	0.8	2.3
Perimeter	5.6%	2.8%	3.4%	11.8%	0.5	0.6	2.1
Power	0.0%	0.0%	0.0%	0.01%	0.4	0.6	2.0
Treeadd	3.7%	5.6%	2.8%	12.1%	1.5	0.8	3.3
TSP	2.0%	1.0%	1.0%	4.0%	0.5	0.5	2.0
Voronoi	0.2%	0.1%	0.2%	0.5%	0.7	0.7	2.4

which includes all additional instructions that are neither prefetches nor loads/stores (`NOP` instructions are also not counted). *Total* is simply their sum, which is less than 10% in six cases, between 10% to 20% in three cases, and about 36% in `health`. For the last case, the 36% increase in the raw instruction count was translated into a combined increase of only 18% in the *busy* and *inst stall* sections in Figure 3.15. This demonstrates that the superscalar machine we simulated is quite effective at overlapping this instruction overhead with other computation, and this is generally true for other benchmarks. We also observe from Table 3.7 that a prefetch is accompanied by another 1.5 additional instructions on average, and over half of them are loads/stores.[12] Later, in Section 3.5.6, we will study the performance impact of the number of memory functional units.

After considering the overall instruction overhead, we now focus on the cost of prefetches themselves. Figure 3.18 shows the fraction of dynamic prefetches that are *unnecessary* because the data is found in the primary cache. For each application, we show four different bars indicating the total (dynamic) unnecessary prefetches caused by static prefetch instructions with hit rates up to a given threshold. Hence the bar labeled "**100**" corresponds to all unnecessary prefetches, whereas the bar labeled "**99**" shows the total unnecessary prefetches if we exclude prefetch instructions with hit rates over 99%, etc. This breakdown indicates the potential for reducing overhead by eliminating static

[12]Intuitively, we would expect that a greedy prefetch must require at least one extra load to generate the prefetch address, which is not the case for applications like `mst` and `power` in Table 3.7. There are two reasons for this discrepancy. The main reason is that each greedy prefetch actually results in two prefetch instructions for fetching two consecutive lines and these two prefetches need only one load to obtain the first line address. Another reason is that the compiler occasionally optimizes away some of these extra loads by using loads in the original program that generate the same addresses.

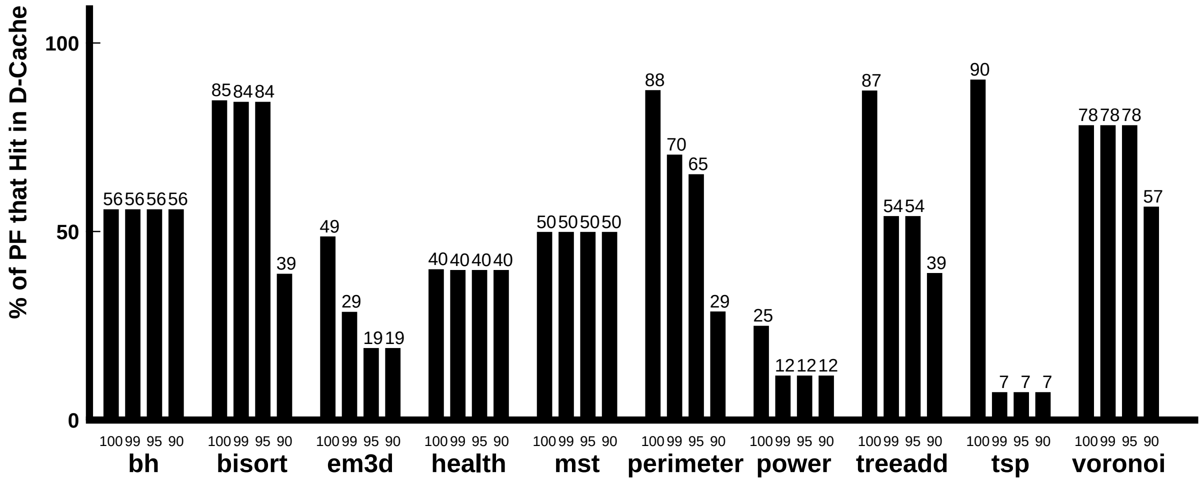


Figure 3.18: Unnecessary greedy prefetches.

prefetch instructions that are clearly of little value. For example, eliminating prefetches with hit rates over 99% would eliminate over 90% of the unnecessary prefetches in `tsp`, thus decreasing overhead significantly. In contrast, reducing overhead with a flat distribution (e.g., `bh`) is more difficult since prefetches that sometimes hit also miss at least 10% of the time (therefore, eliminating them may sacrifice some latency-hiding benefit). We will quantify the benefit of eliminating unnecessary prefetches later in Section 3.5.5.

3.5.2 Case Studies

Having performed a quantitative evaluation of greedy prefetching, we now look at the memory access behaviors of individual applications in more detail. Doing this not only gives us a source-level understanding of the performance results reported in the last section, but may also provide opportunities to further improve performance.

bh: Nodes of an octree are traversed in `bh_walk()`, and over 70% of load stalls occur in `bh_test()` and `bh_work()` (see Figure 3.19). However, as we observe from Figure 3.16, a substantial fraction of these misses are not covered by prefetches. This is because nodes in `bh` are large relative to the cache line size (32 bytes), and consequently prefetching only two lines (our default prefetch size) cannot cover the entire node. Another interesting point in Figure 3.19 is that the eight children of the current node `t` are prefetched *after*, but not before `bh_test(t)`, since the actual number of prefetch instructions to be inserted is 16 (2 lines for each child), which exceeds the threshold value (10) used by the compiler to decide whether prefetches

```
void bh_walk(..., node*t, ...)  {
  if (bh_test(t)) {
    prefetch(t→children[0]);
    prefetch(t→children[1]);
    prefetch(t→children[2]);
    prefetch(t→children[3]);
    prefetch(t→children[4]);
    prefetch(t→children[5]);
    prefetch(t→children[6]);
    prefetch(t→children[7]);
    for (k=0; k<8; k++){
      r = t→children[k];
      if (r)
        bh_walk(..., r, ...);
    }
  } else
      bh_work(..., t, ...);
}
```

Figure 3.19: Abstract code fragment with greedy prefetches from `bh`.

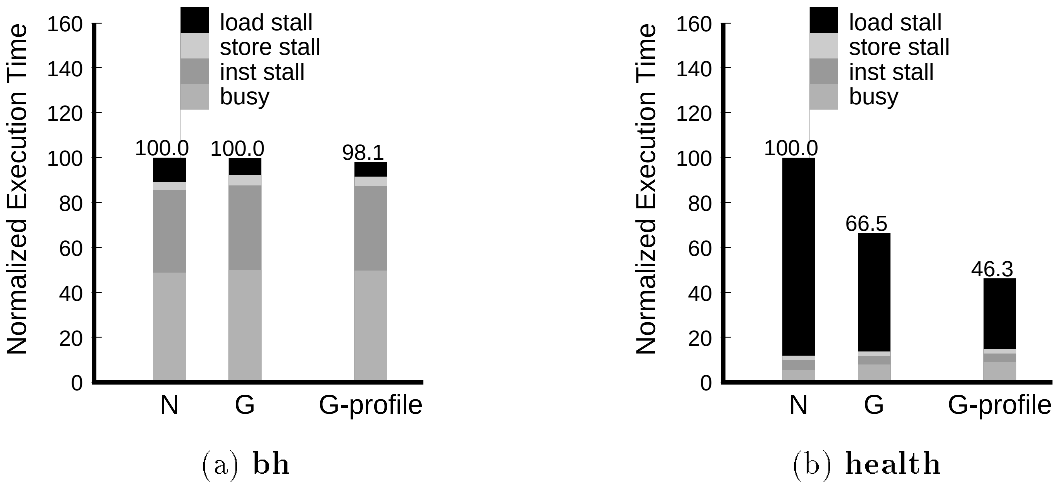


Figure 3.20: Performance of profiling-assisted greedy prefetching for `bh` and `health` (**N** = no prefetching, **G** = greedy prefetching, **G-profile** = profiling-assisted greedy prefetching).

should be inserted at a later place.

Performance of this benchmark can be improved by profiling information in the following way. In addition to `bh_walk()`, the compiler also inserts greedy prefetches into a few other procedures in `bh`. Unfortunately, many of these prefetches turn out to be useless and lead to the 14% increase in the memory traffic that we have seen in Figure 3.17. Figure 3.20(a) shows that if the compiler inserts prefetches into `bh_walk()` only but not the other procedures, we can get a 2% performance improvement (the **G-profile** bar).

```
int Bimerge(root,spr_val,dir) {
   rv = root→value;
   pl = root→left; pr = root→right;
   ...
   while (pl != NIL) {
      prefetch(pl→left);
      prefetch(pl→right));
      prefetch(pr→left);
      prefetch(pr→right);
      lv = pl→value;
      pll = pl→left; plr = pl→right;
      rv = pr→value;
      prl = pr→left; prr = pr→right;
      if (...)  {
         if (...)  {
            SwapValRight(pl,pr,plr,prr,lv,rv);
            pl = pll; pr = prl;
         } else {
            pl = plr; pr = prr;
         }
      } else {
         if (...)  {
            SwapValLeft(pl,pr,pll,prl,lv,rv);
            pl = plr; pr = prr;
         } else {
            pl = pll; pr = prl;
         }
      }
   }
   if (root→left != NIL) {
      prefetch(root→left);
      prefetch(root→right);
      ...
      rl = root→left; rr = root→right;
      ...
      root→value=Bimerge(rl,value,dir);
      spr_val=Bimerge(rr,spr_val,dir);
   }
   ...
}
```

Figure 3.21: Abstract code fragment with greedy prefetches from `bisort`.

bisort: The main RDS is a binary tree, and the important cache misses occur in `Bimerge()`, which contains both loops and recursion (see Figure 3.21). The four "grandchildren" of `root` are prefetched early in the while loop. Nearly all load misses are prefetched but about 40% of these prefetches arrive late. The relatively large fraction of unnecessary prefetches shown in Figure 3.18 is caused by the pair of prefetches for `root→left` and `root→right` in Figure 3.21. Locality analysis might help this case by recognizing that a portion of data accessed in the recursive calls has already been brought into the cache by the while loop.

health: Over 90% of load stalls are due to linked-list accesses inside `waiting()` (see Figure 3.22). Despite a 36% increase in total instruction count, the 30% reduction in load stalls results in a large speedup. But Figure 3.16 also indicates that

```
void waiting(Village *village, List *list)
  while (list != NULL) {
    prefetch(list→forward);
    i = village→hosp.free_personnel;
    p = list→patient;
    if (i > 0) {
      t = village→hosp.free_personnel;
      village→hosp.free_personnel = t-1;
      p→time_left = 3;
      p→time = t + 3;
      l = &(village→hosp.waiting);
      removeList(l, p);
      l = &(village→hosp.assess);
      addList(l, p);
    } else {
      t = p→time;
      p→time = t + 1;
    }
    list = list→forward;
  }
}
```

Figure 3.22: Abstract code fragment with greedy prefetches from `health`.

```
void *HashLookup(int key, Hash hash) {
  j = (hash→mapfunc)(key);
  for(ent = hash→array[j];
    ent && ent→key!=key;
    ent=ent→next)
      prefetch(ent→next);
  if (ent) return ent→entry;
  return NULL;
}
```

Figure 3.23: Abstract code fragment with greedy prefetches from `mst`.

there are still half of load misses left unprefetched. Profiling information points out that the major source of these misses is the dereference of the pointer field `patient` within `List` nodes. Our greedy prefetching algorithm does not recognize `list→patient` as an RDS access since there is no recurrent pointer update for the `patient` object type. As we have discussed earlier in Section 3.3, the compiler does not prefetch pointers unless they point to RDSs. If the compiler also prefetches `list→patient` in this case, we will enjoy an additional speedup of 42%, as illustrated in Figure 3.20(b).

mst: About 90% of load stalls occur in `HashLookup()`, where it searches for an item in an array of linked lists (see Figure 3.23). Load misses happen at two pointer dereferences: `ent→key` and `hash→mapfunc`. Although the compiler prefetches `ent→next`, only a small portion of the latency can be hidden since the loop body is so small. The compiler does not prefetch `hash` as it is not an RDS pointer, and even if it did, there would not be sufficient time to hide the latency. Lacking time to

Table 3.8: Minor schemes under history-pointer prefetching.

Label	Description
H	Updating history-pointers during RDS traversals
H-A	Updating history-pointers during both RDS traversals and allocation
H2	Scheme **H** with two history-pointers per RDS node
H2-A	Combining schemes **H2** and **H-A**

prefetch ahead appears to be a general problem with hash tables, and prefetching prior to the hash function invocation is beyond the scope of our algorithm.

perimeter: A quadtree is traversed through recursive procedure calls. Most load misses are covered by greedy prefetches, but 88% of them are unnecessary. There are two reasons for these unnecessary prefetches: (i) the same parts of the quadtree can be visited through different recursive procedures, thus resulting in unanticipated data locality; and (ii) each node contains a pointer to its parent, which the compiler prefetches along with the four child pointers, but the parent is already in the cache.

treeadd: Numbers distributed on a binary tree are added together through a postorder recursive traversal. Prefetching both children of each tree node is able to hide 67% of load stall, and this translates into a 38% speedup. The 87% unnecessary prefetches are a consequence of the abundant data locality enjoyed by adjacent nodes in the traversal.

tsp: Each RDS node contains four pointers: two for binary tree-like accesses, and two for doubly-linked list-like accesses. Prefetching these four pointers results in an 80% reduction of load stall. The speedup is less impressive since load stall accounts for only a few percents of the total execution time. The large number of unnecessary prefetches occur for the same reasons as `perimeter`.

3.5.3 Performance of History-Pointer Prefetching

In this section, we investigate the performance of history-pointer prefetches inserted by our compiler. We first present the results with the prefetching distance (i.e. d in Figure 3.5) fixed to three nodes ahead. Later on, we will vary this parameter to see its performance impact.

We experimented with a number of minor schemes under history-pointer prefetching, as described in Table 3.8. The ordinary scheme is labelled as **H**, where history pointers

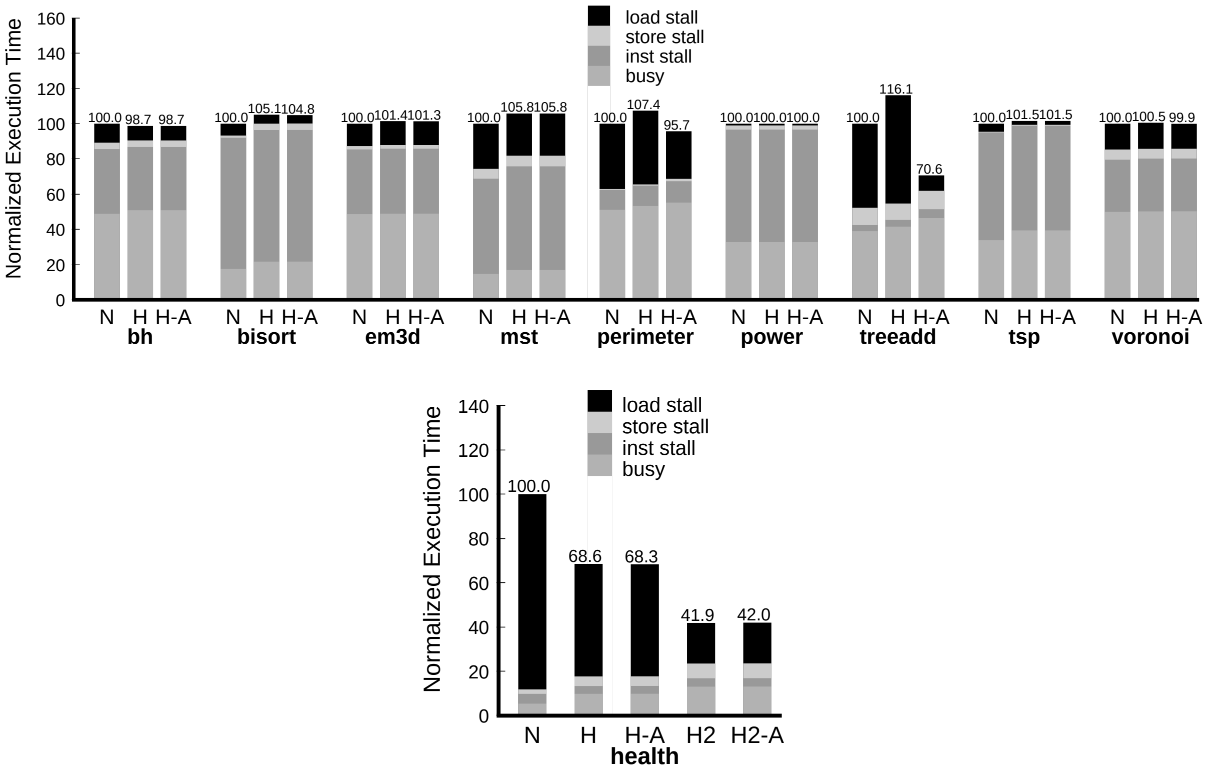


Figure 3.24: Performance of history-pointer prefetching (**N** = no prefetching, **H**, **H-A**, **H2**, and **H2-A** are minor history-pointer prefetching schemes as described in Table 3.8).

are updated each time an RDS is traversed. Scheme **H-A** differs from **H** by adding code to update history-pointers also during the time that the RDS is allocated in memory (Note that depending on whether there are recurrent pointer updates in the memory allocation routine, RDS allocation may or may not be recognized as RDS traversals.) The motivation for scheme **H-A** is that in some applications, an RDS is traversed only once in the entire execution and hence any history-pointers set up during that traversal have no chance of being used later. Therefore, the best that we can do is to set up a history-pointer to point to an RDS node immediately after it is allocated, whereby it can be prefetched via the history-pointer during the only traversal of the RDS. Since in general it is quite difficult for the compiler to know exactly where an RDS is allocated in the program (especially if the program has its own memory allocation routines instead of using those provided by the language), the compiler relies on hints from the programmer on where the memory allocation takes place. Scheme **H2** is only applicable to `health`, where two history-pointers are added to each `List` record—one for prefetching `list` and the other for prefetching `list`$\rightarrow$`parent` (illustrated earlier in Figure 3.22). Scheme **H2-A** is simply a combination of **H2** and **H-A**.

Table 3.9: Memory performance improvement for history-pointer prefetching.

	No Prefetch		History-Pointer Prefetch			
Benchmark	Load D-Cache Miss Rate	Average Load Miss Penalty (cycles)	Minor Scheme	Load D-Cache Miss Rate	Average Load Miss Penalty (cycles)	Load Stall Reduction
BH	3.46%	16.8	H	2.45%	17.3	31.0%
Bisort	4.79%	27.4	H-A	2.93%	27.0	41.2%
EM3D	3.36%	29.1	H-A	3.65%	27.3	-71.0%
Health	30.62%	46.1	H2	5.15%	25.6	91.6%
MST	9.20%	56.4	H	6.93%	66.0	-1.9%
Perimeter	7.81%	67.1	H-A	7.38%	72.6	41.5%
Power	0.17%	14.3	H	0.14%	15.3	-103.9%
Treeadd	1.40%	68.8	H-A	0.30%	15.2	94.0%
TSP	2.38%	39.3	H	0.69%	37.9	21.5%
Voronoi	1.63%	22.3	H-A	1.67%	21.2	44.1%

Figure 3.24 shows the performance of history-pointer prefetching (schemes **H2** and **H2-A** are only applicable to `health`). First of all, we observe that **H-A** performs substantially better than **H** in `perimeter` and `treeadd`. In `perimeter`, the two main traversals of the same quadtree are in different visiting orders: one of them mimics the tree creation order while the other does not. As a result, updating history-pointers while allocating the tree nodes helps to launch useful prefetches during the first traversal. In `treeadd`, the tree creation order is identical to the visiting order of the only traversal. History-pointer prefetching performs particularly well in `health` because the structure of the lists accessed in `waiting()` (see Figure 3.22) are modified only slowly throughout the over ten thousand times it is called. We have also learned from the case studies in Section 3.5.2 that many misses in `health` are caused by dereferences of `list`$\rightarrow$`parent`. Adding a history-pointer for prefetching `list`$\rightarrow$`parent` produces an 138% speedup over the case without prefetching, in spite of the even larger space overhead. Also note that this speedup is greater than the one represented by the **G-profile** bar in Figure 3.20(b) since prefetches have more time to finish with history-pointer prefetching. Comparing the performance of history-pointer prefetching with that of greedy prefetching shown earlier in Figure 3.15, history-pointer prefetching has a noticeable performance advantage over greedy prefetching in `bh`, `health`, and `treeadd` but is significantly worse in `bisort`, `mst`, and `perimeter`.

As we did for greedy prefetching, we use the additional performance metrics shown in Figures 3.25-3.27 and Tables 3.9-3.10 to obtain a deeper understanding of how well history-pointer prefetching performs. Only the best performing minor history-pointer

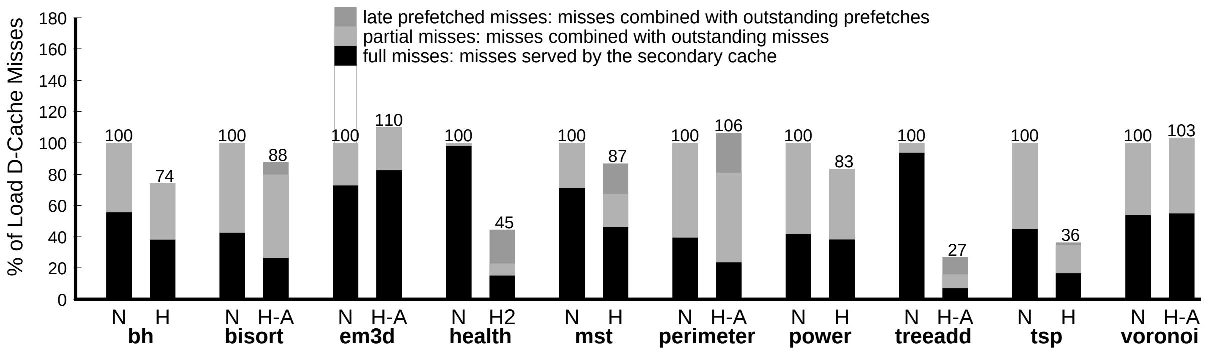


Figure 3.25: Breakdown of all load D-cache misses (**N** = no prefetching, **H**, **H-A**, and **H2** are history-pointer prefetching schemes).

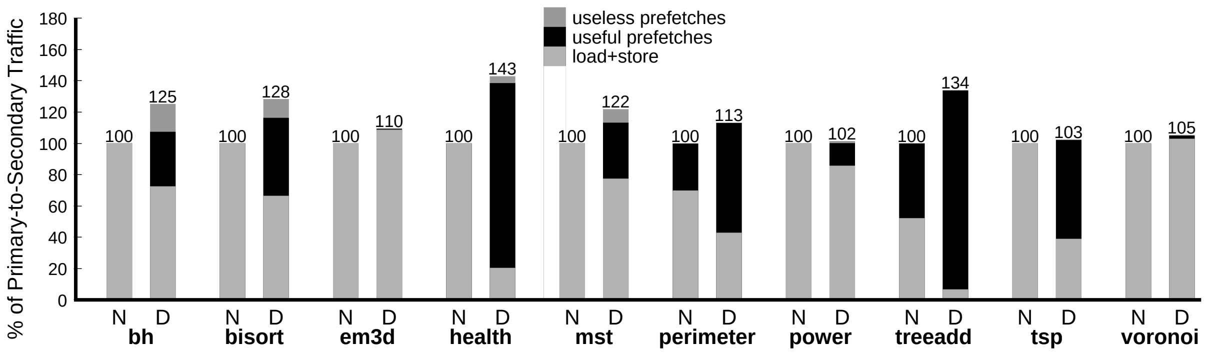


Figure 3.26: Breakdown of the traffic between the D-cache and secondary cache (**N** = no prefetching, **H**, **H-A**, and **H2** are history-pointer prefetching schemes).

prefetching schemes for individual benchmarks are shown in these figures and tables. Highlights of these results include:

- The over 90% load stall reduction in `health` and `treeadd` is a consequence of the improvement in both the miss coverage and timeliness over greedy prefetching. In contrast, the coverage of history-pointer prefetching is inferior in the cases of `bh`, `bisort`, `mst`, `perimeter`, and `tsp`.

- History-pointer prefetching generates more memory traffic than greedy prefetching in nearly all the benchmarks. The presence of history-pointers increases the data set size. This is most prominent in `health` where two history pointers are added to each `List` node, posing a 66% space overhead.

- Although history-pointer prefetching has a smaller fraction of unnecessary prefetches,

Table 3.10: Instruction overhead of history-pointer prefetching.

		Overall Overhead (percentage of original instruction count)				Per-Prefetch Overhead		
Benchmark	Minor Scheme	Prefetches	Loads/ Stores	Others	Total	Loads/ Stores	Others	Total (including the prefetch itself)
BH	H	0.5%	2.1%	1.7%	4.3%	4.1	3.3	8.4
Bisort	H-A	3.9%	19.2%	2.3%	25.4%	4.9	0.6	6.5
EM3D	H-A	0.0%	0.5%	0.2%	0.7%	45.0	14.0	60.0
Health	H2	34.4%	105.1%	41.2%	180.6%	3.1	1.2	5.3
MST	H	3.3%	9.2%	5.2%	17.7%	2.8	1.6	5.4
Perimeter	H-A	0.7%	5.5%	2.1%	8.3%	8.0	3.0	12.0
Power	H	0.0%	0.0%	0.0%	0.02%	2.9	2.5	6.4
Treeadd	H-A	1.9%	14.8%	2.8%	19.4%	8.0	1.5	10.5
TSP	H	1.9%	7.7%	5.8%	15.4%	4.0	3.0	8.0
Voronoi	H-A	0.1%	0.4%	0.2%	0.7%	7.5	3.9	12.4

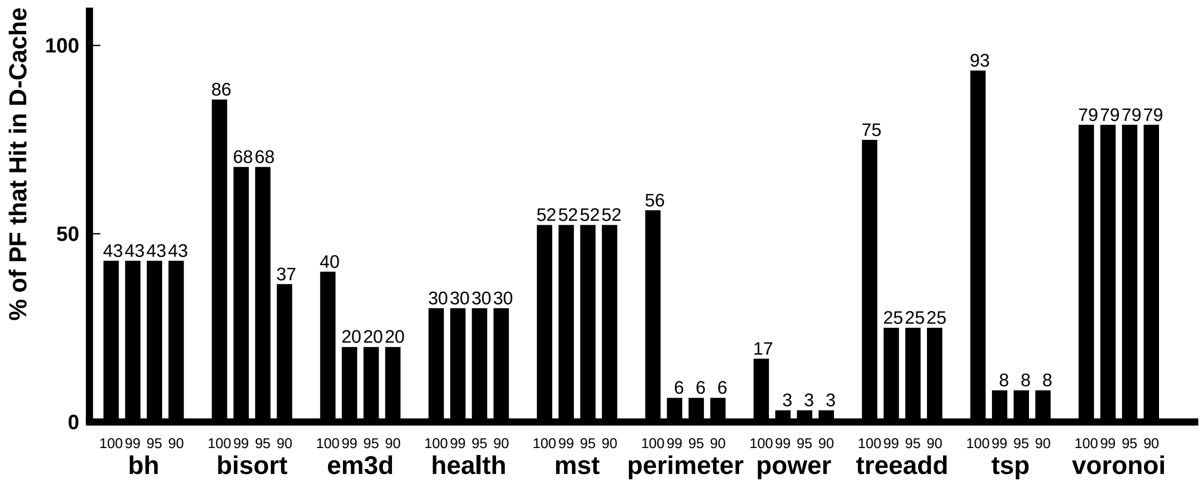


Figure 3.27: Unnecessary history-pointer prefetches.

it has much more instruction overhead than greedy prefetching due to the extra work required to maintain the history-pointers. This overhead can potentially be reduced by updating the history-pointers less frequently.

Varying the Prefetching Distance

A key parameter of the history-pointer prefetching algorithm is the prefetching distance. Ideally, increasing the prefetching distance would convert some late prefetches into on-

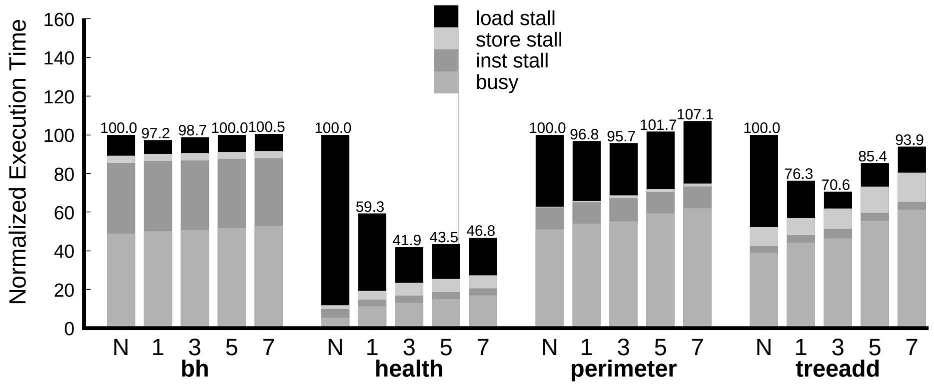


Figure 3.28: Performance impact of the history-pointer prefetching distance (**N** = no prefetching, **x** = prefetching x nodes ahead).

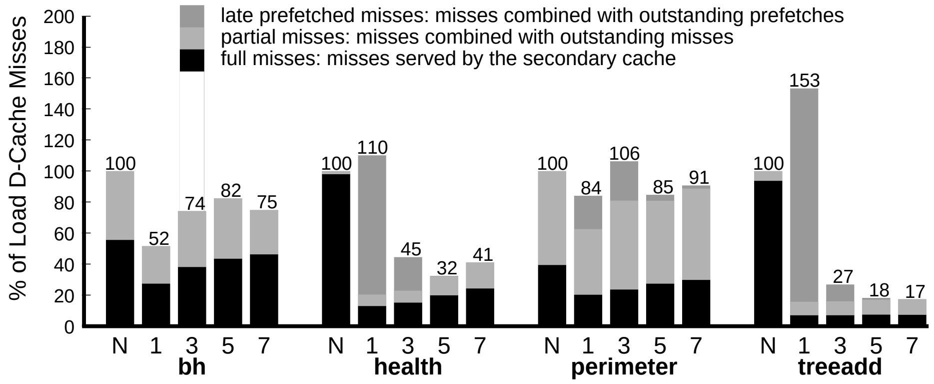


Figure 3.29: Breakdown of all load D-cache misses for history-pointer prefetching distances ranging from one to seven (**x** = prefetching x nodes ahead).

time ones and hence provide further performance gain. In practice, however, there are other concerns in choosing this parameter. One is whether the prefetch coverage could be hurt by increasing the prefetching distance. This could happen if the traversal order recorded by the history-pointers is not exactly the same as the actual one. Another concern is about the instruction overhead. If the history-pointer prefetching implementation that has instruction overhead proportional to the prefetching distance is used (i.e. the default implementation for a prefetching distance smaller than eight, which is the scalar-FIFO implementation in Figure 3.14), the benefits provided by a larger prefetching distance could potentially be offset by the increased overhead.

Figure 3.28 shows the performance for the four benchmarks whose performance is improved by history-pointer prefetching, with prefetching distances ranging from one to seven. The **3** bars correspond to the best performing cases in Figure 3.24. As we see in

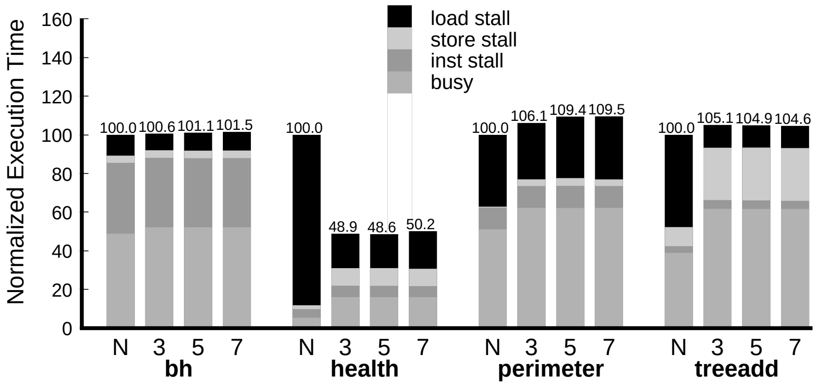


Figure 3.30: Performance impact of the prefetching distance when the array-FIFO implementation of history-pointer prefetching exemplified in Figure 3.14 is used (**N** = no prefetching, **x** = prefetching x nodes ahead).

Figure 3.28, increasing the prefetching distance from one to three improves performance in three cases. However, performance starts to drop with larger prefetching distances. There are two reasons for this performance degradation. First, Figure 3.29 indicates that the miss coverage does suffer from larger prefetching distances in `bh`, `health`, and `perimeter`. Second and more importantly, the additional overheads of larger prefetching distances more than offset their latency reduction benefits.

To investigate if these overheads will be less of a problem by using the history-pointer prefetching implementation that has constant instruction overhead (i.e. the array-FIFO implementation in Figure 3.14), we repeated our experiments using this implementation. The results in Figure 3.30 demonstrate that although the overheads did remain constant, they are still significantly larger than those of the default implementation (in Figure 3.28) in most cases for this range of prefetching distances.

In summary, for these applications running on the machine we modeled, the scalar-FIFO implementation of history-pointer prefetching with a prefetching distance of three nodes ahead appears to be a good compromise when the miss coverage, timeliness, and instruction overheads are all taken into account.

3.5.4 Performance of Data-Linearization Prefetching

We now evaluate the performance of data-linearization prefetching. The results presented in this section assume that the data layout of programs is not modified. In Chapter 4, we will apply data-linearization prefetching, together with the dynamic layout optimizations enabled by memory forwarding, on a number of applications including `bh`, `mst`, and `health`.

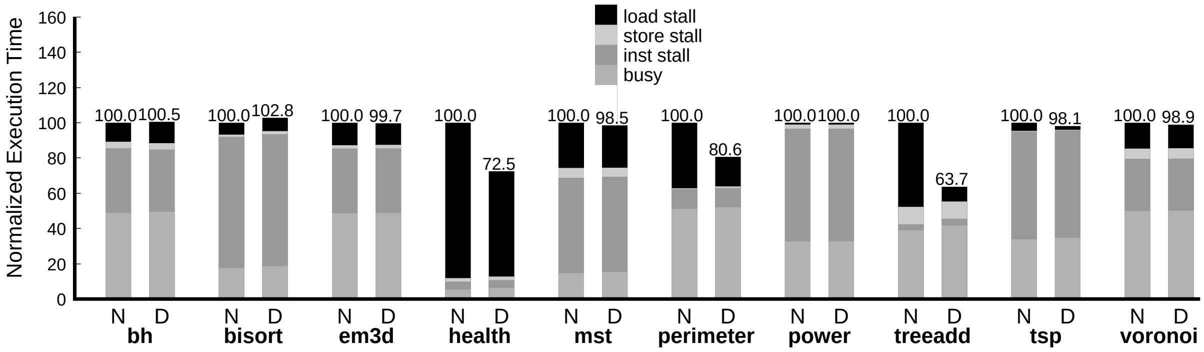


Figure 3.31: Performance of data-linearization prefetching (**N** = no prefetching, **D** = data-linearization prefetching).

Figure 3.31 shows the performance of data-linearization prefetching with a prefetching distance of three nodes ahead. Data-linearization prefetching is particularly applicable to both `perimeter` and `treeadd`, because the creation order is identical to the major subsequent traversal order in both cases. As a result, data linearization does not require changing the data layout and yet achieves significant speedups over greedy and history-pointer prefetching. For both benchmarks, detailed results from Figure 3.32 and Table 3.12 reveal that data-linearization prefetching has converted a large fraction of late prefetched misses into hits, while at the same time incurred the least instruction overhead among our three prefetching schemes. For the other eight benchmarks, Figure 3.32 shows that data-linearization prefetching generally covers fewer misses than greedy prefetching and hence performs worse in a few cases. We also observe from Figure 3.33 that a relatively large number of prefetches are useless in four benchmarks. This is not surprising since the data layout has not been linearized in these cases and hence sequentially prefetched items may be of little use. Finally, Table 3.12 clearly indicates that data-linearization prefetching has substantially smaller instruction overhead compared with the other two schemes. This is because only one prefetch is issued per node and no pointer dereference is needed to generate the prefetch address (this explains why more than half of the benchmarks have zero "*Loads/Stores*" overhead in Table 3.12).

Varying the Prefetching Distance

In the same way we evaluated history-pointer prefetching, we measured the performance of data-linearization prefetching for a range of prefetching distances. The results are shown in Figure 3.35. Interestingly, Figure 3.35 shows a trend similar to that of history-pointer prefetching: a prefetching distance of three nodes appears to be a good choice

Table 3.11: Memory performance improvement for data-linearization prefetching.

Benchmark	No Prefetch		Greedy Prefetch		
	Load D-Cache Miss Rate	Average Load Miss Penalty (cycles)	Load D-Cache Miss Rate	Average Load Miss Penalty (cycles)	Load Stall Reduction
BH	3.46%	16.8	4.17%	16.4	-19.7%
Bisort	4.79%	27.4	4.94%	28.5	-16.0%
EM3D	3.36%	29.1	3.35%	28.6	2.1%
Health	30.62%	46.1	31.17%	31.0	31.5%
MST	9.20%	56.4	8.67%	59.9	3.4%
Perimeter	7.81%	67.1	3.69%	45.8	67.5%
Power	0.17%	14.3	0.14%	14.8	18.4%
Treeadd	1.40%	68.8	0.52%	13.9	92.5%
TSP	2.38%	39.3	1.30%	36.1	32.6%
Voronoi	1.63%	22.3	1.55%	20.4	12.4%

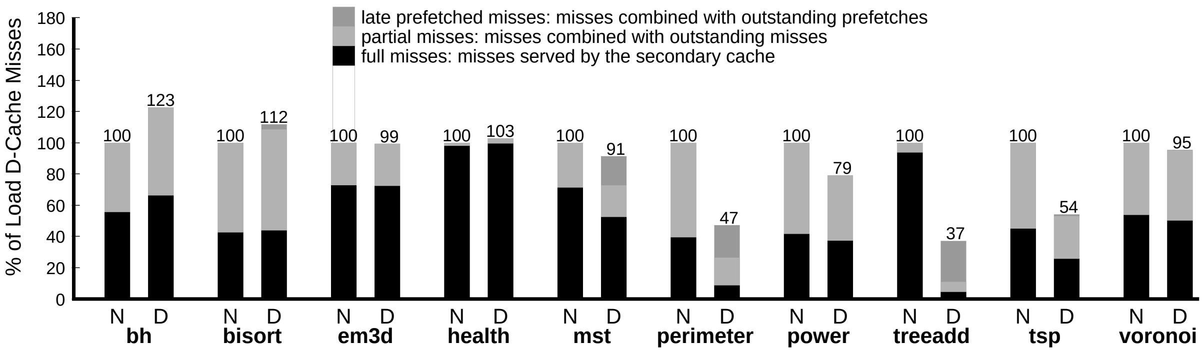


Figure 3.32: Breakdown of all load D-cache misses (**N** = no prefetching, **D** = data-linearization prefetching.)

for these applications and machine configurations. Larger prefetching distances degrade performance in `health`, `mst`, and `perimeter` because the miss coverage begins to drop when we prefetch further than three nodes ahead. The performance degradation, however, is not as severe as with history-pointer prefetching since the instruction overhead remains small across different prefetching distances. For `treeadd`, performance is actually improved by increasing the prefetching distance since most late prefetched misses have been successfully converted into hits when prefetching seven nodes ahead, while the miss coverage remains high.

Figure 3.33: Breakdown of the traffic between the D-cache and secondary cache (**N** = no prefetching, **D** = data-linearization prefetching).

Table 3.12: Instruction overhead of data-linearization prefetching.

Benchmark	Overall Overhead (percentage of original instruction count)				Per-Prefetch Overhead		
	Prefetches	Loads/ Stores	Others	Total	Loads/ Stores	Others	Total (including the prefetch itself)
BH	0.5%	0.5%	0.5%	1.5%	1.0	1.0	3.0
Bisort	3.9%	3.3%	1.6%	8.9%	0.8	0.4	2.2
EM3D	0.0%	0.0%	0.0%	0.02%	0.2	0.5	1.7
Health	17.9%	0.2%	9.7%	27.9%	0.0	0.5	1.5
MST	3.3%	0.4%	2.6%	6.3%	0.1	0.8	1.9
Perimeter	1.4%	0.0%	0.7%	2.1%	0.0	0.5	1.5
Power	0.0%	0.0%	0.0%	0.01%	0.0	0.5	1.5
Treeadd	3.7%	0.0%	1.9%	5.6%	0.0	0.5	1.5
TSP	2.0%	0.0%	1.0%	3.0%	0.0	0.5	1.5
Voronoi	0.2%	0.0%	0.2%	0.4%	0.2	1.1	2.3

3.5.5 Reducing Overhead Through Locality Analysis

Our compiler currently does not attempt to analyze data locality across RDS node accesses. As a result, we may prefetch nodes unnecessarily that already reside in the cache (as discussed earlier in Section 3.5.1). For numeric applications, sophisticated locality analysis techniques have been combined with loop splitting techniques to isolate the dynamic iterations that should be prefetched [96]. Unfortunately, the control structures in RDS codes are less amenable to isolating dynamic node visitations, so our only option may be to eliminate static prefetch instructions altogether. This makes sense for prefetches that are almost always unnecessary (i.e. have very high hit rates).

To estimate the performance potential of exploiting locality information, we used

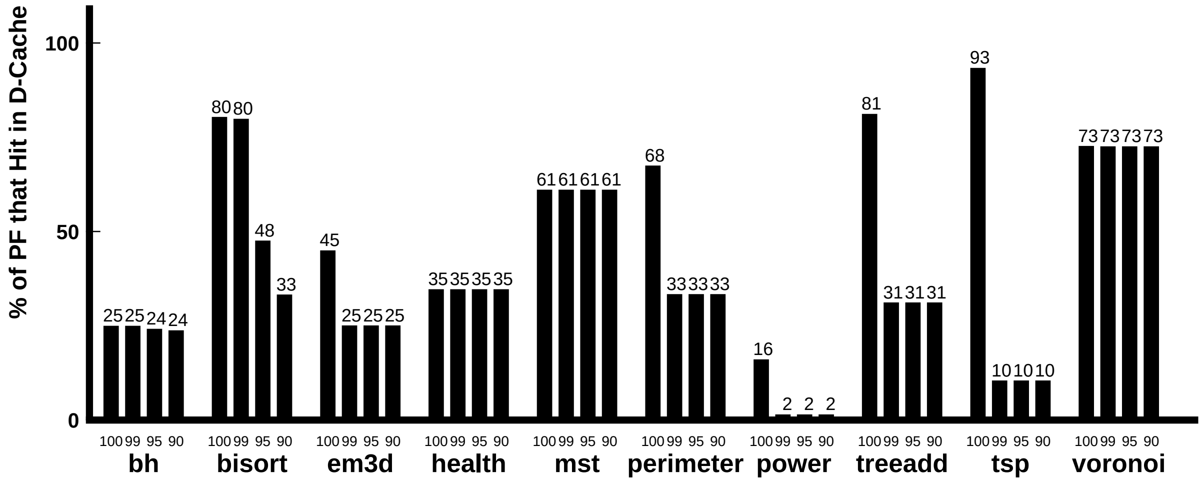


Figure 3.34: Unnecessary data-linearization prefetches.

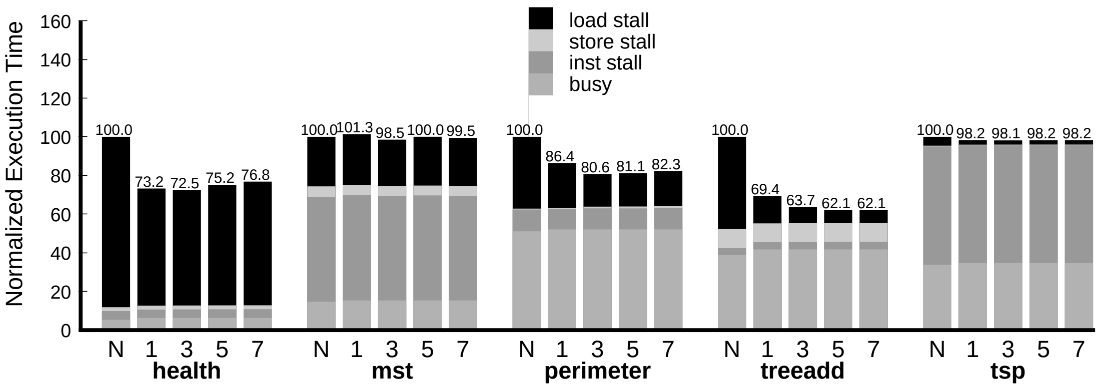
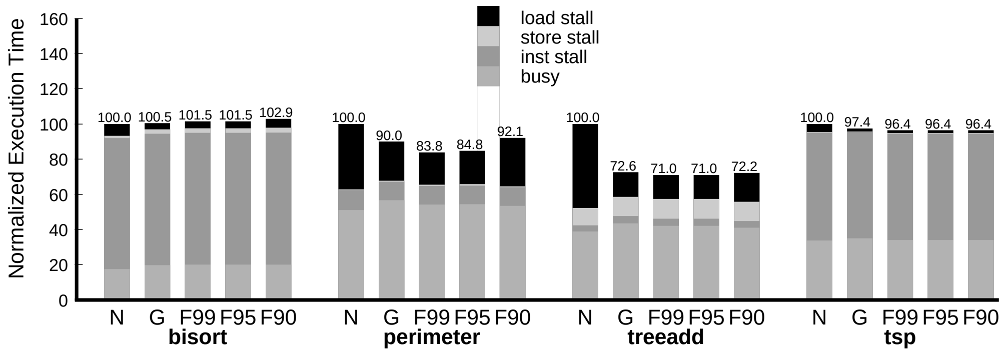
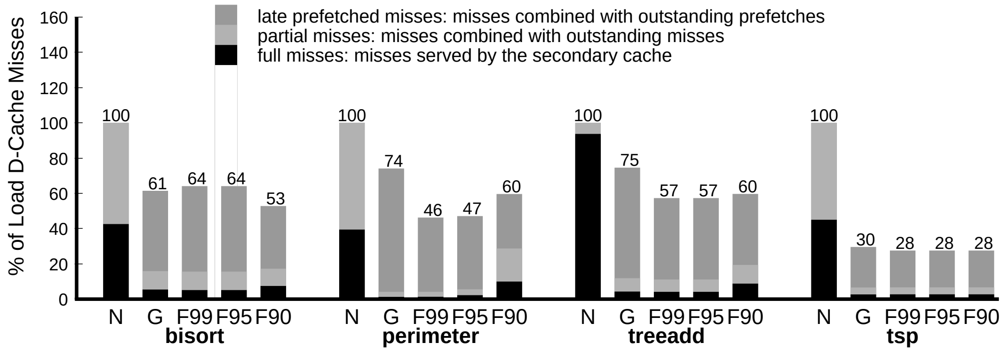


Figure 3.35: Performance impact of the data-linearization prefetching distance (**N** = no prefetching, **x** = prefetching x nodes ahead).

memory feedback information from our simulator to eliminate prefetch instructions with hit rates above a certain threshold from the greedy prefetching code. Figure 3.36 shows our results for the four applications that were affected by setting this threshold to 99%, 95%, and 90% hit rates. As we see in Figure 3.36, eliminating prefetches with hit rates above 99% improves performance by 1-6% for three applications by eliminating unnecessary prefetches without sacrificing much coverage. However, eliminating prefetches with hit rates over 90% does hurt performance in `bisort`, `perimeter`, and `treeadd` since the miss coverage drops substantially. Therefore, improved locality analysis may help performance by eliminating prefetches that are almost always unnecessary (e.g., the "`parent`" pointer in `perimeter`), but without more powerful techniques for isolating dynamic node



(a) Execution time



(b) Breakdown of all load D-cache misses

Figure 3.36: Performance of feedback-guided greedy prefetching on four benchmarks (**G** = greedy prefetching, **Fxx** = greedy prefetching where static prefetch instructions with hit rates over xx% have been eliminated).

visitations, the gains do not appear to be as large as with numeric codes.

3.5.6 Architectural Issues

In this section. we study how our prefetching schemes perform under variations of several machine parameters—in particular those that are related to the memory subsystem. We focus on the six benchmarks whose performance is improved by one or more of our schemes. For each scheme, we chose the version that achieved the best performance in the previous sections for the experiments in this section. We begin by looking at the impact of varying the miss latencies.

Miss Latencies

The miss latency is the most important single parameter since it directly determines the amount of memory stalls. While having larger miss latencies will increase the per-

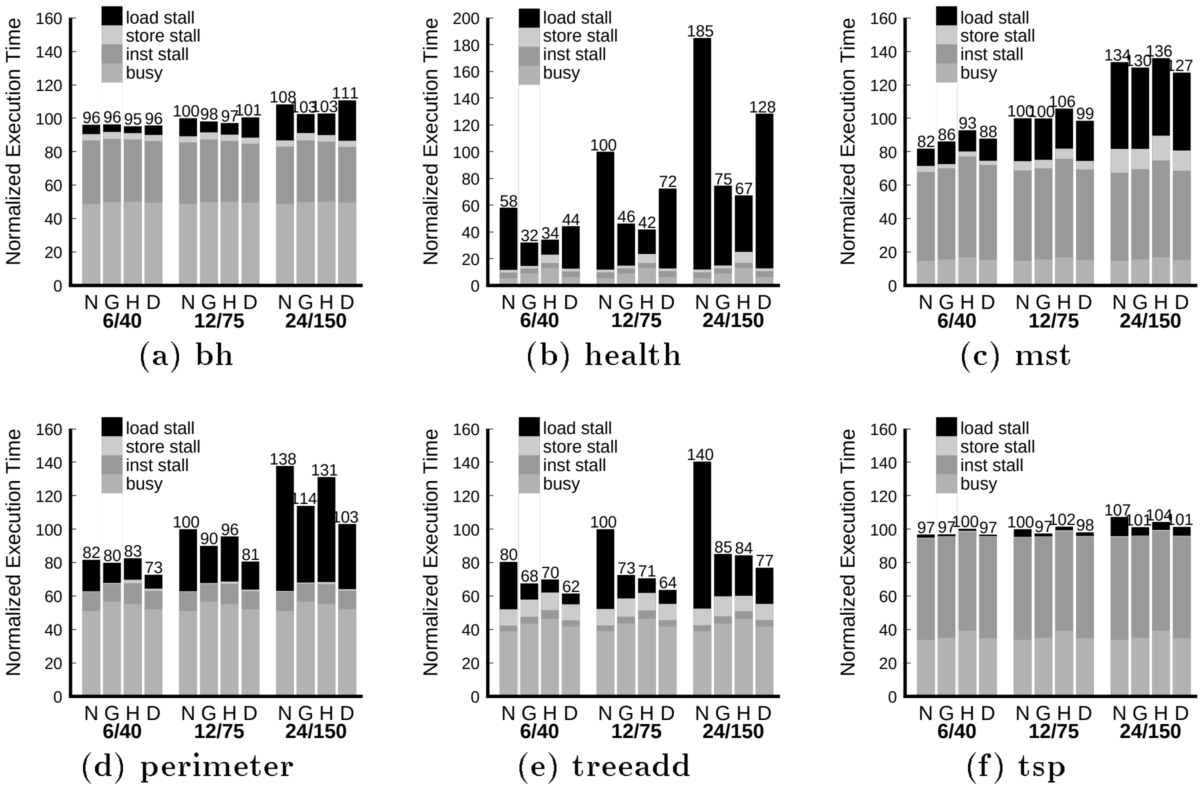


Figure 3.37: Performance of our prefetching schemes with varying *miss latencies* (**N** = no prefetching, **G** = greedy prefetching, **H** = history-pointer prefetching, and **D** = data-linearization prefetching).

formance potential of prefetching, this potential is unlikely to be effectively exploited if prefetches are not launched far enough in advance. This is particularly a concern for greedy prefetching which has little control over the prefetching distance. To understand how scalable the performance of our schemes is with respect to the miss latency, we did the following two experiments.

In the first experiment, we applied our schemes to the six benchmarks with two new configurations of miss latencies. The results are shown in Figure 3.37, where each latency configuration is identified as **xx/yy** meaning that the primary-to-secondary latency and the primary-to-memory latency are **xx** cycles and **yy** cycles, respectively, plus any delay due to contention. Hence, our baseline configuration is **12/75**, and the two new configurations represent the cases that memory latency is less or more problematic. Concentrating first on the cases *without* prefetching in Figure 3.37, we see that performance is very sensitive to miss latencies. In fact, doubling the miss latencies from one configuration to the other actually doubles the amount of load stalls. This implies that the increased miss latencies are *not* automatically tolerated by the out-of-order execution

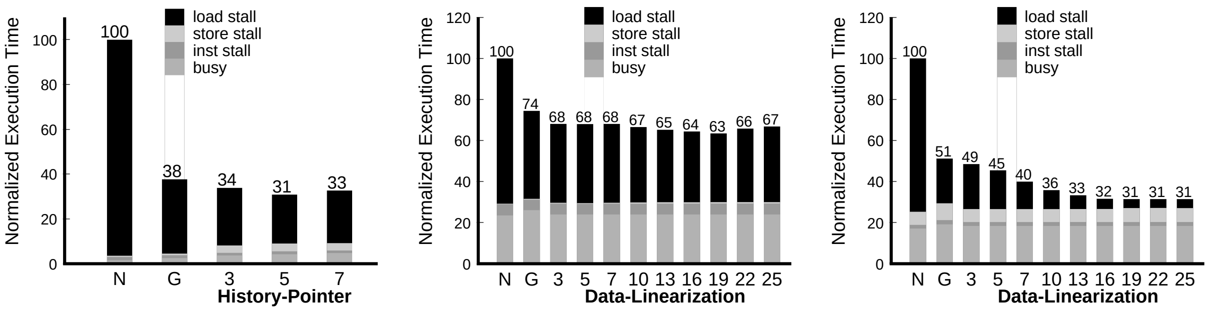


Figure 3.38: Performance of our prefetching schemes with 50-cycle primary-to-secondary latency and 300-cycle primary-to-memory latency (**N** = no prefetching, **G** = greedy prefetching, **x** = prefetching distance of x nodes for history-pointer or data-linearization prefetching).

of our processor. The good news is that performance of our prefetching schemes does increase along with the latency. Even for greedy prefetching, it achieves larger speedups with longer latencies. And for history-pointer and data-linearization prefetching, we do not need to adjust the prefetching distance (which is three for all three configurations) to maintain good speedups across these latencies.

Having shown that our schemes still perform well across the range of miss latencies experienced by current or next-generation machines, we test our schemes further in the second experiment by using an even more extreme latency configuration: 50 cycles and 300 cycles for primary-to-secondary and primary-to-memory latencies, respectively. We are specifically interested in knowing if increasing the prefetching distance of history-pointer prefetching and data-linearization prefetching can cope with such large latencies. Figure 3.38 shows the results for the three benchmarks that suffer the most from cache misses. We are encouraged by the finding that greedy prefetching still delivers high performance despite its lack of control over the prefetching distance. By tuning the prefetching distance, history-pointer and data-linearization prefetching can offer even larger performance improvements. While history-pointer prefetching achieves the best performance in `health` at a relatively small prefetching distance(five), data-linearization prefetching performs the best in both `perimeter` and `treeadd` when prefetching as far as 19 nodes ahead.

In summary, all the three prefetching schemes are fairly robust with respect to the miss latency. Very large miss latencies can be effectively handled by increasing the prefetching distance in history-pointer and data-linearization prefetching.

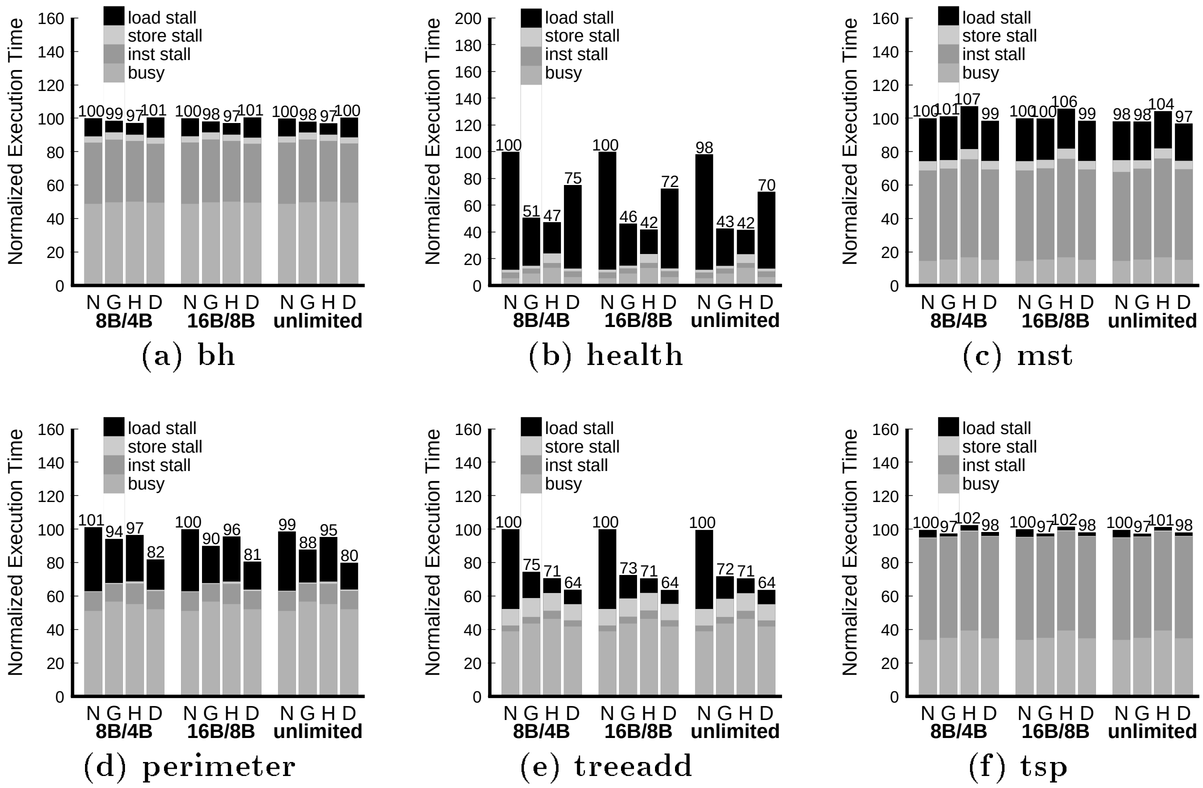


Figure 3.39: Performance of our prefetching schemes with varying *memory bandwidth* (**N** = no prefetching, **G** = greedy prefetching, **H** = history-pointer prefetching, and **D** = data-linearization prefetching).

Memory Bandwidth

Prefetching can improve performance only if sufficient bandwidth is available in the memory subsystem. We have already demonstrated that our prefetching schemes work well with the realistic bandwidth assumed in the baseline model. Nevertheless, for the following reasons, we are also interested in knowing how these schemes would perform if more or less bandwidth is available. If having *more* bandwidth significantly increases the performance improvement of prefetching, it means our prefetching schemes are somewhat bandwidth-limited. Although the results in previous sections indicate that our prefetching schemes only moderately increase the total memory traffic, bursty traffic may create contention that increases the overall execution time substantially. On the other hand, if decreasing the bandwidth does not affect the performance of our schemes significantly, it means they will also be applicable in machines where memory bandwidth is a more limited resource (e.g., some lower-end machines or machines with a single memory bus shared by multiple processors).

Recall that the baseline model has 16 bytes/cycle primary-to-secondary bandwidth and 8 bytes/cycle secondary-to-memory bandwidth. Figure 3.39 shows the impact of decreasing this bandwidth by a half (**8B/4B**) and of increasing it to unlimited bandwidth (**unlimited**). We make the following observations from Figure 3.39. First, while reducing the bandwidth does degrade the performance of our schemes somewhat, the overall performance gain still remains high. Hence our prefetching schemes can achieve good performance in a spectrum of bandwidth that is common for recent machines. Second, increasing the bandwidth beyond the baseline one does not lead to significant performance improvement. Therefore, we conclude that our schemes are not bandwidth-limited.

Cache Size Variations

In this section, we study the impact of cache size with a threefold motivation. First, we would like to estimate the performance gains achievable by our schemes in the presence of much larger input data sets. However, very large data sets are usually infeasible for our experiments due to the simulation time constraints. An alternative to increasing the data set size is to scale down the cache size correspondingly. Second, we are also interested in knowing how much performance benefit prefetching can offer on machines with much larger caches. One approach to coping with memory latency, employed by some processors such as the HP PA-8500 [75], is to include unusually large on-chip caches. Though we do not consider this brute-force approach a complete solution because of its potentially negative impact on the cache hit access time as well as its significant increase in the die size, it is useful to know if prefetching is needed for such large caches. Finally, it is important to know how big a cache is required to hold prefetched data long enough for them to be used and to prevent useful data from being displaced by prefetches. This is particularly relevant for greedy prefetching, where both the prefetching distance and accuracy are not precisely controlled. To address these three issues, we performed the following two experiments.

In the first experiment, we applied our schemes with five new sets of cache sizes. The results are shown in Figure 3.40, where **xx/yy** denotes a combination of a **xx**-byte primary data cache and a **yy**-byte secondary unified cache. The baseline case is the set of bars above **16K/512K**. We begin by concentrating on how the code without prefetching performs across the different cache sizes, and then later compare this to the behavior of the code with prefetching.

First, let us consider the case *without* prefetching, we note a variety of different behaviors in Figure 3.40. At one extreme are `perimeter` and `treeadd`, where performance is nearly insensitive to the cache size until 1M/16M caches are used. This is because uses

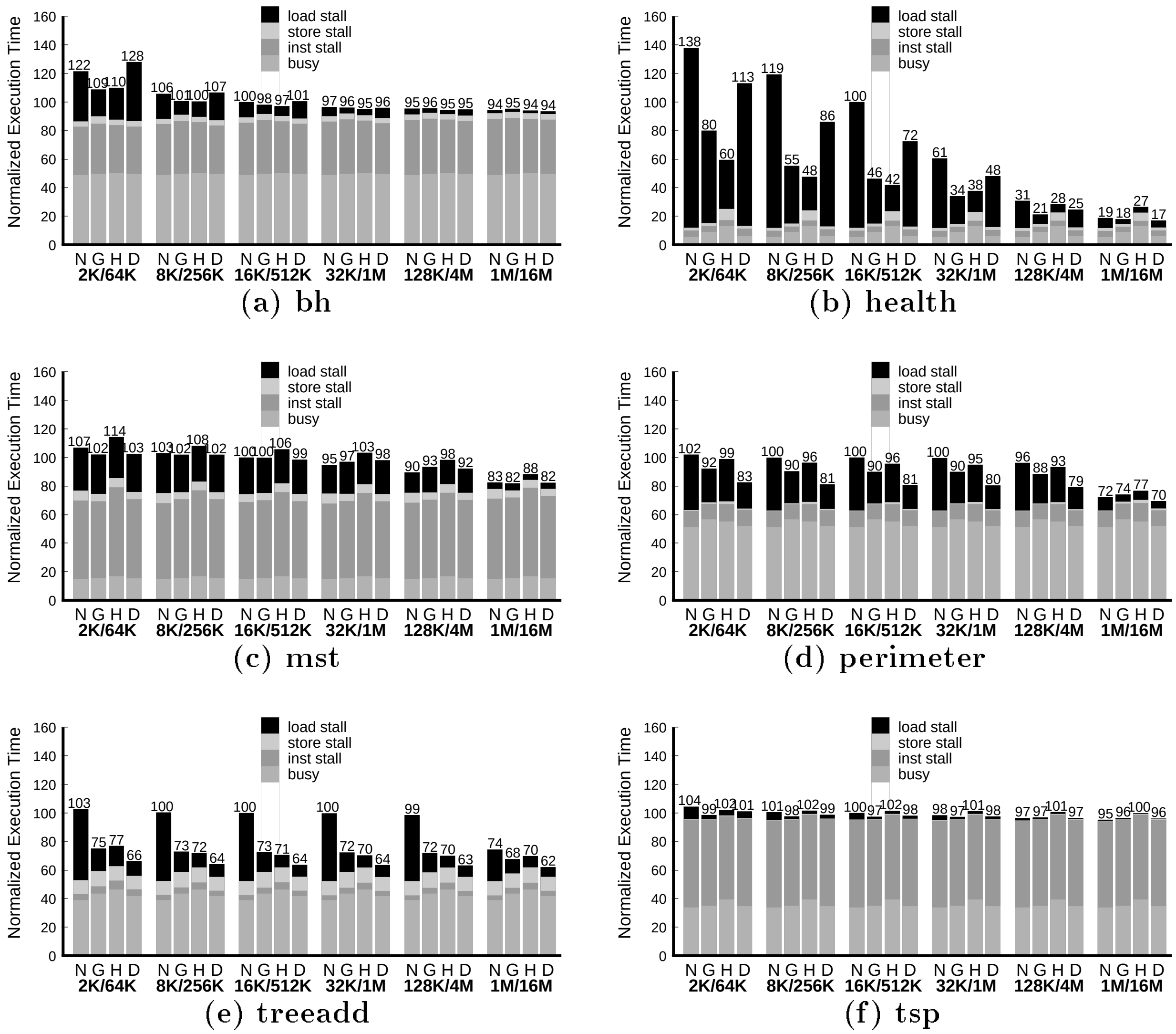
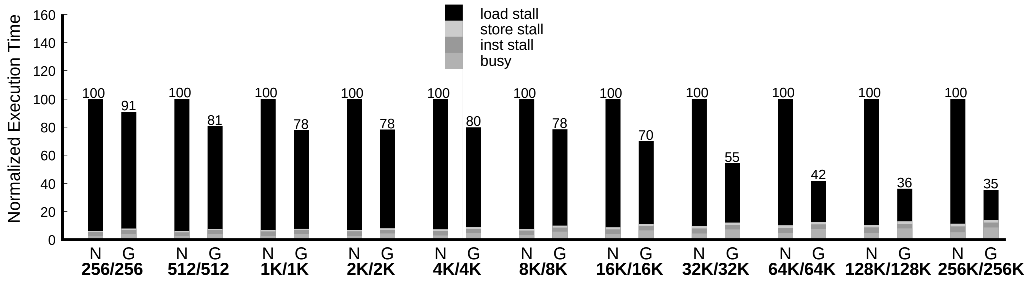
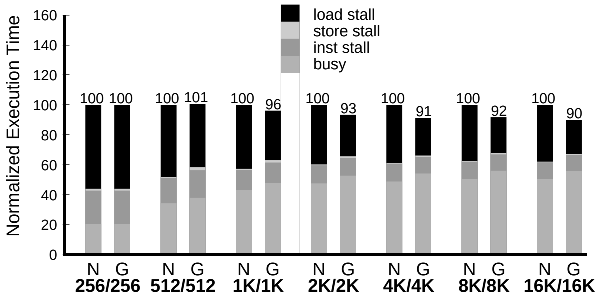


Figure 3.40: Performance of our prefetching schemes with varying *cache sizes* (**N** = no prefetching, **G** = greedy prefetching, **H** = history-pointer prefetching, and **D** = data-linearization prefetching).

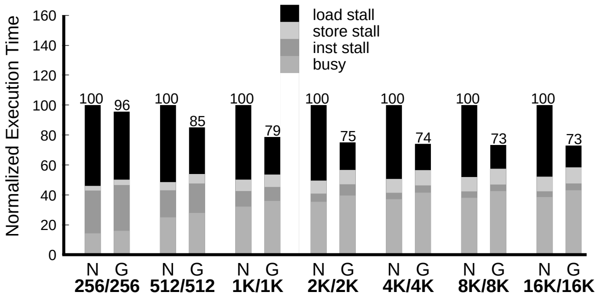
of the same RDS node are very far apart in time and the large number of intervening references will flush the node from the cache unless the cache is as big as 1M/16M (Table 3.3 shows that `perimeter` and `treeadd` have 6 MB and 12 MB of memory allocated, respectively). At the other extreme is `tsp`, where data reuses occur so closely in time that even a 2K/64K cache is large enough to retain the data. In between these two extremes are `bh`, `mst`, and `health`, where we can find multiple knees in performance and each knee occurs when a key data structure fits in the cache. This phenomenon is most noticeable in `health` which accesses a large number of linked lists over time. Consequently, bigger caches are able to retain more lists for future reuse.



(a) **health**



(b) **perimeter**



(c) **treeadd**

Figure 3.41: Performance of greedy prefetching over a wide range of *cache sizes* (**N** = no prefetching, **G** = greedy prefetching). Execution time is renormalized for each cache size.

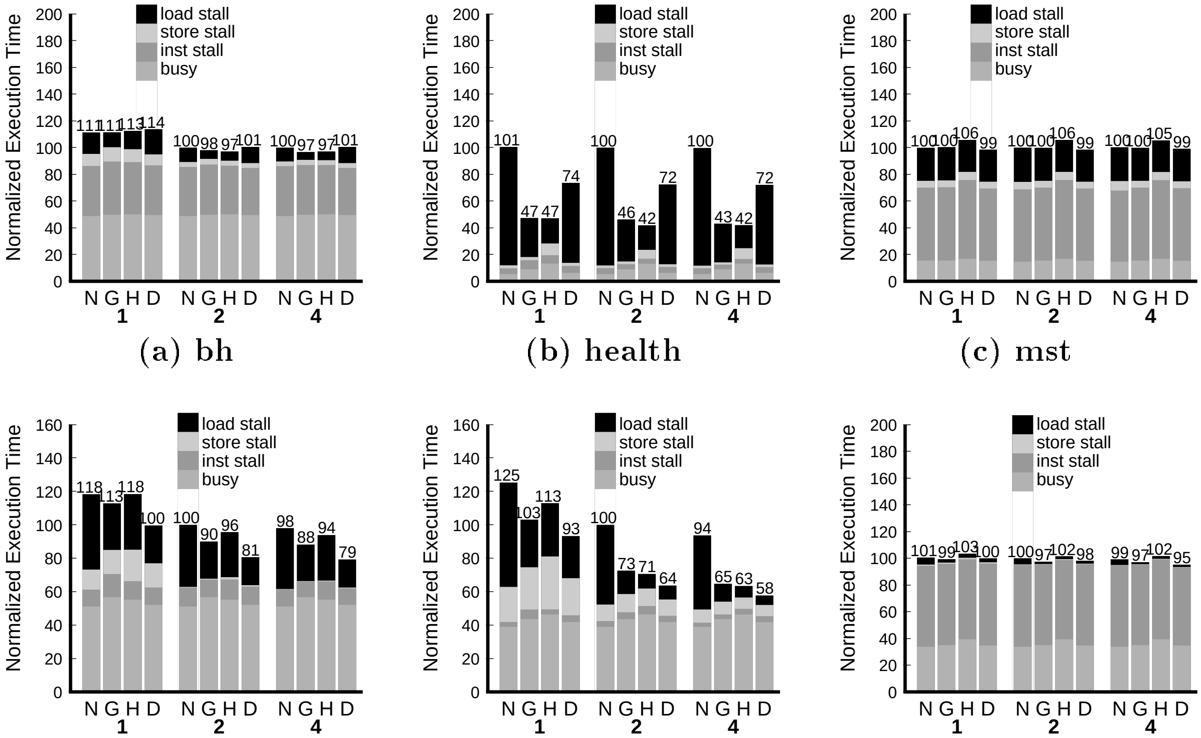
Tuning our attention to the code *with* prefetching, Figure 3.40 shows that in general the relative performance improvement offered by prefetching is larger with smaller cache sizes. The obvious reason for this is that smaller caches have more latency to hide due to capacity misses, and thus there is more room for improvement. For instance, history-pointer prefetching offers only a 3% speedup for `bh` on a 16K/512K cache but provides an 11% speedup on a 2K/64K cache. Comparative performance of the three prefetching schemes is also affected by the cache size. For example, history-pointer prefetching is 48% faster than data-linearization prefetching for `health` on a 2K/64K cache but is 59% slower on an 1M/16M cache. As the cache gets bigger, prefetches become less beneficial while instruction overhead is increasingly important. Therefore prefetching schemes with smaller overhead are favored in the very large caches. Overall, we are encouraged to see that prefetching performs reasonably well over this range of cache sizes. In fact, in three of the six cases (`health`, `perimeter`, and `treeadd`), the best performance of code with prefetching on 2K/64K caches was better than code without prefetching on 32K/1M caches. And even with 1M/16M caches, prefetching can still manage to offer performance gains in these three cases.

The purpose of the second experiment is to determine the cache capacity needed to hold greedily prefetched data. In this experiment, we applied greedy prefetching to `health`, `perimeter`, and `treeadd` over a wide range of cache sizes. To simplify the analysis, the primary and secondary caches were set to the same capacity in each case. In addition, both caches were 8-way set-associative to avoid any clouding effects on the results due to conflict misses. The results are shown in Figure 3.41. First, we note that the relative performance improvement of greedy prefetching with very small cache sizes is much smaller than what we have seen in the baseline case. In fact, greedy prefetching degrades performance of `perimeter` for cache sizes smaller than 1K bytes. Fortunately, relative performance of prefetching improves along with the cache size. For both `perimeter` and `treeadd`, a 4K/4K cache is big enough to hold most greedily prefetched data. Note that this result is quite consistent with our argument in Section 3.2.2 (summarized by Equation 3.1), which states that as long as the cache is sufficiently larger than an RDS node, capacity misses would not prevent greedily prefetched items from staying in the cache until they are used. For `health`, however, a cache as big as 128K bytes is needed to fully exploit the potential of greedy prefetching. Why is such a large cache size required? The answer can be found by considering again the procedure `waiting()` shown in Figure 3.22, where most load misses in `health` occur. In each iteration of the while-loop, procedures `removeList()` and `addList()` are called to search `list`→`patient` through the lists at `village`→`hosp.waiting` and `village`→`hosp.assess`, respectively. Therefore the resultant working set in each iteration of `waiting()` is so large that we need more than 64K bytes to keep the prefetched data from being displaced.

To summarize, we have seen that the performance advantages of our prefetching schemes often remain significant even as the cache size is varied. We also find that cache sizes that are common for recent processors are sufficiently capacious to hold both the working set and prefetched data.

Number of Memory Functional Units

The results from Tables 3.7, 3.10, and 3.12 suggest that most part of prefetching overhead is due to additional memory instructions. On average, each prefetch costs 1.8, 10.4, and 1.2 new memory instructions (including the prefetch itself) for greedy, history-pointer, and data-linearization prefetching, respectively. These extra instructions place more demand on the memory functional units and could potentially slow down other parts of the execution. Therefore, we study in this section the performance impact of the number of memory units.



(a) **bh** (b) **health** (c) **mst**

(d) **perimeter** (e) **treeadd** (f) **tsp**

Figure 3.42: Performance of our prefetching schemes with varying *number of memory units* (**N** = no prefetching, **G** = greedy prefetching, **H** = history-pointer prefetching, and **D** = data-linearization prefetching).

Similar to a few more recent superscalar processors such as the Alpha 21264 [66] and the HP PA-8500 [75], our baseline model contains two memory units. However, earlier processors like the MIPS R10000 [139] and the PowerPC 604 [121] have only one memory unit. Therefore it is useful to know how our schemes perform on machines with a single memory unit. On the other hand, we would like to know if it is justified to add more memory units to support our schemes better. To answer these questions, we experimented our schemes with one, two, and four memory units (having more than four memory units would not help since at most four instructions can be issued per cycle and all functional units are already fully-pipelined). The results are shown in Figure 3.42, where the number of memory units is displayed underneath the prefetching schemes. As we observe from the figure, our prefetching schemes perform well in general across these three configurations. Nevertheless, having two memory units is important for the absolute performance of `perimeter` and `treeadd` for both with and without prefetching cases, since this improves performance by up to 37% over a single unit. Due to the memory instruction overhead, the relative performance gains of prefetching are larger

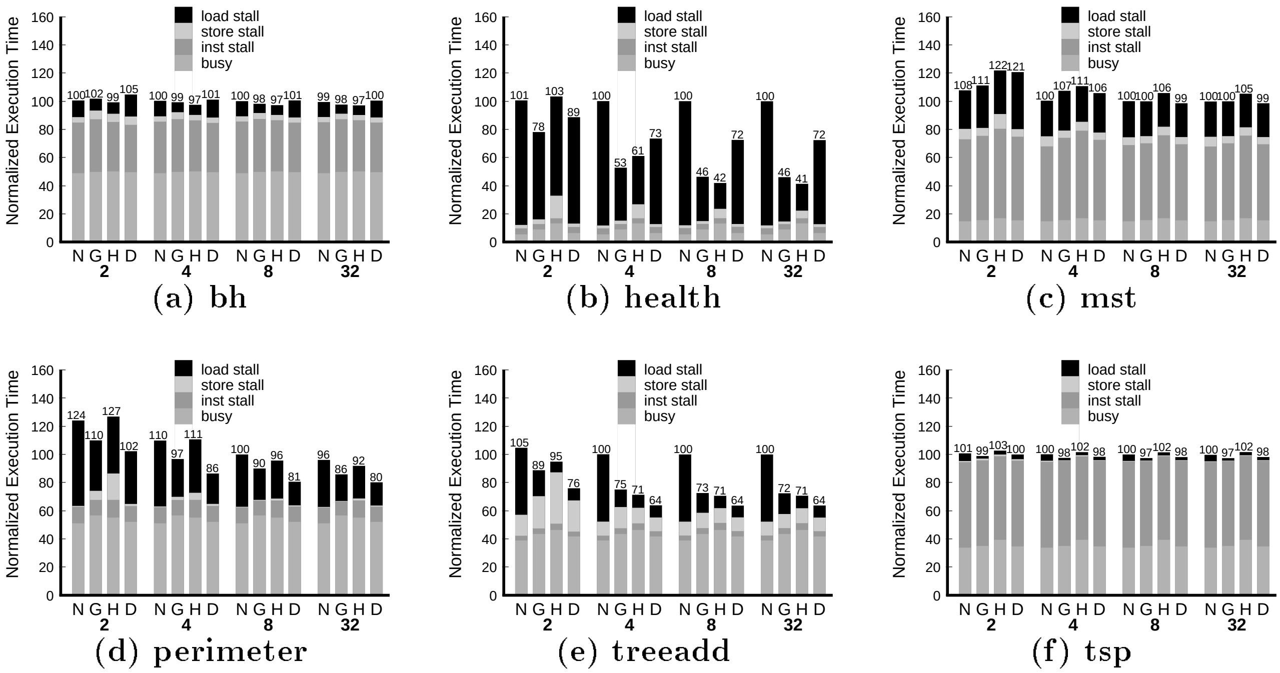


Figure 3.43: Performance of our prefetching schemes with varying *number of miss handlers* (**N** = no prefetching, **G** = greedy prefetching, **H** = history-pointer prefetching, and **D** = data-linearization prefetching).

with two memory units than with only one. Among the three prefetching schemes, history-pointer prefetching tends to be most sensitive to the memory unit count as a result of its relatively large memory instruction overhead. Figure 3.42 also suggests that we may not need more than two units to support these prefetching schemes because the marginal gain of having four units is considerably smaller.

Number of Miss Handlers

In this section, we investigate the effects of the number of miss handlers. Miss handlers were originally proposed by Kroft [69] to implement a *lockup-free cache*—a cache that can handle other requests while a miss is still outstanding. A miss handler, also known as a "miss information/status handling register" (MSHR) in Kroft's paper, keeps track of an in-flight cache-line fetch (i.e. a primary cache miss). When the line returns, appropriate actions will be done to complete the reference based on the information stored in the miss handler.

There is a functionality of miss handlers that is particularly relevant to prefetching: the capability of *combining* a new reference with an outstanding *prefetch* accessing the same line. Such combining offers two advantages. First, the later reference will not be sent all the way up to the next memory hierarchy level though the data is not present in the primary cache. Thus, memory bandwidth is saved. Second and more importantly, when the later reference is a *load*, it stalls for only the amount of time that the prefetch still needs to finish but not for the entire miss latency. This is especially important in cases where we observed substantial number of these *late prefetched misses* in the miss coverage graphs.

Obviously, the number of miss handlers is a concern since it decides how many references can be processed at a time. This number may not be very large in practice because the amount of state involved in each handler is non-trivial due to the generality of the MSHR mechanism (e.g., the data address, pointers to the cache entry and destination register, written data, and some other control information are recorded in the handler). While more recent processors such as the Alpha 21264 [66] provides as many as 32 load miss handlers, some others provide only a few (e.g., four in the MIPS R10000 [139]). If all handlers are busy when there is a new memory reference that neither hits in the primary cache nor can be combined with one of the outstanding misses, there are two options to handle this reference, depending on its type: for a load or store, it must wait in a buffer until a handler is free; for a prefetch, it can wait or be simply dropped. Either option may have negative impact on performance. Since prefetches create additional competition for these handlers, it is important to find out how many miss handlers are required to accommodate our prefetching schemes.

Figure 3.43 shows the results of an experiment where we varied the number of miss handlers between 2 and 32. Our baseline model has eight handlers. This graph shows that prefetching performance suffers from having only two handlers. This is especially a problem for history-pointer prefetching which incurs much *store* stall time for updating history pointers. Fortunately, this problem nearly disappears once we have eight handlers. Further adding more handlers does not increase performance much except for `perimeter`. Another statistic from our simulations indicates that there are fewer than eight outstanding data references over 99% of the time, and therefore eight handlers are sufficient in most cases. For `perimeter`, however, there is still about 10% of the time where up to 16 misses are outstanding. Overall, the number of miss handlers provided by recent processors (usually eight or more) appears to be adequate for exploiting our schemes.

Support for Non-Excepting Memory Operations

Because of the speculative nature of prefetching, prefetches may occasionally access invalid addresses and hence generate exceptions that should not happen in normal ex-

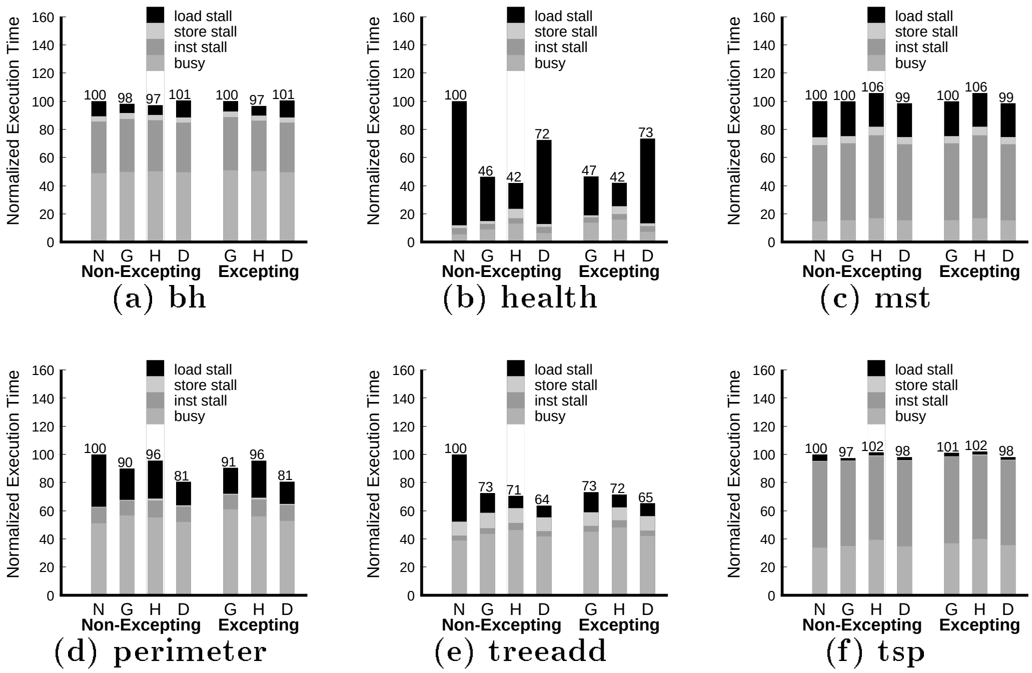


Figure 3.44: Performance of our prefetching schemes with *excepting* vs. *non-excepting* prefetches (**N** = no prefetching, **G** = greedy prefetching, **H** = history-pointer prefetching, and **D** = data-linearization prefetching).

ecution. In array-based codes, invalid prefetch addresses typically only occur if one prefetches off the end of an array. The compiler can avoid this hazard by peeling off the last few loop iterations and disables prefetching in these iterations. In contrast, invalid prefetch addresses may occur unexpectedly in RDS codes due to NULL pointers. There are two approaches to tackling this problem. The first one is to guard each prefetch by a NULL test. The second one is to make prefetches *non-excepting*—instructions that will never raise exceptions [25]. While the first approach allows machines not having native prefetch instructions to emulate prefetches by regular loads, the NULL tests create additional instruction overhead.

To quantify the performance benefit of having non-excepting prefetches, we forced the compiler to enclose any prefetches that may have invalid addresses with a NULL test. In order to minimize overhead, the compiler first looks for any existing NULL tests that can be used for this purpose. If no such tests are found, a new one will be inserted. As we see in Figure 3.44, the performance gaps between the non-exception and excepting versions are usually 1% or less. Only in the case of greedy prefetching in `bh` and `tsp` are there a 2% and a 4% difference, respectively. This result demonstrates that our dynamically-scheduled processor was quite able to hide the overhead of these

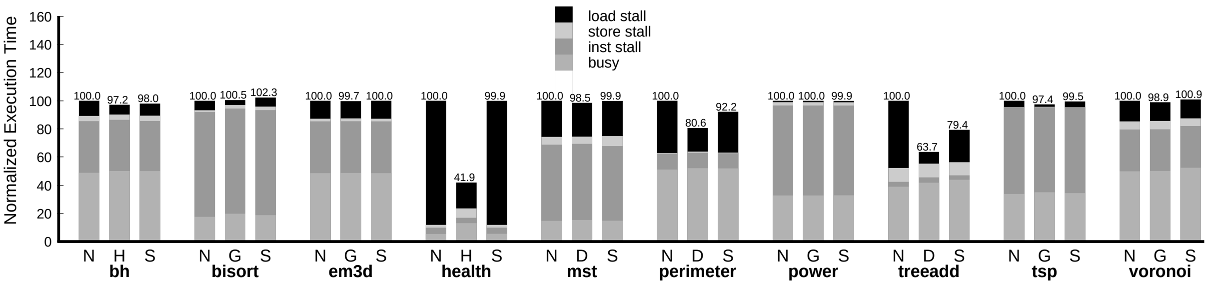


Figure 3.45: Performance comparison between SPAID and our schemes (**N** = no prefetching, **G** = greedy prefetching, **H** = history-pointer prefetching, **D** = data-linearization prefetching, and **S** = SPAID).

NULL tests. In a previous study [79], however, we found that the overhead of these tests was more significant (up to 7% increase in execution time). The reason is that our processor model has been evolving over time following the technology trend, and the one used in this chapter has a more advanced branch predictor and a larger reorder buffer than the one that we used previously had. As a result, instruction overhead has become less of a problem. Nevertheless, non-excepting prefetches still have the advantage of handling addresses that are invalid but not null at all. In addition to non-excepting prefetches, non-excepting *load* instructions also appear to be quite useful for prefetching pointer-based codes, although we currently are not exploiting them aggressively in our compiler.

3.5.7 Performance Comparison with SPAID

The final set of experiments compares the performance of our schemes against SPAID, the only other compiler-based pointer prefetching technique found in the literature. To have a quantitative comparison, we implemented several versions of SPAID in our experimental framework with different numbers of prefetches inserted per call site. Our results are consistent with the conclusion in the SPAID paper [77] that the best performance is achieved by inserting only one prefetch per call site—the **S** bars in Figure 3.45 correspond to this optimal case. When a procedure has multiple pointer arguments, we select the first one pointing to any RDS to prefetch. We also improved the performance of the proposed SPAID scheme for `treeadd` from a slowdown of 13% to a speedup of 26% by prefetching two cache lines at a time rather than one. As we see in Figure 3.45, the best performer of our schemes outperforms the optimal SPAID in all cases except `power`. Performance advantages of our schemes are most prominent in the three cases that suffer the most from load stalls. The problem with SPAID is that it pays significant prefetching overhead without covering many cache misses. In contrast, our schemes do a better job of choosing what to prefetch, and can schedule prefetches earlier to hide more latency.

3.6 Chapter Summary

While automatic compiler-inserted prefetching has shown considerable success in hiding the memory latency of array-based codes, the compiler technology for successfully prefetching pointer-based data structures has thus far been lacking. In this chapter, we propose three prefetching schemes which overcome the pointer-chasing problem, and we implement them in a research compiler. Our experiments with these three schemes on a modern dynamically-scheduled processor produced the following results:

1. Automatic compiler-inserted prefetching can accelerate pointer-based applications by as much as more than twofold. While greedy prefetching performs quite well in spite of its simplicity, the more sophisticated schemes can offer further performance gains by predicting the order in which RDSs are traversed.

2. This performance benefit of prefetching is a combined result of significant reduction in memory stall, while incurring only a relatively small overhead in memory traffic and instructions. While our schemes attempt to launch prefetches on time, a substantial fraction of the miss latency is hidden by late prefetches. On the other hand, our results demonstrate that instruction overhead of prefetching is not overwhelming as the dynamically-scheduled machine is quite successful in tolerating these extra instructions.

3. From an architectural perspective, our results suggest that:

    - Our schemes are robust with respect to the miss latency, memory bandwidth, and cache capacity.
    - For the range of miss latencies found in recent machines, history-pointer and data-linearization prefetching work well with relatively small prefetching distances (around three). Much larger latencies can be handled by increasing the prefetching distance of these two schemes.
    - At least two memory units and eight handlers are needed to support these prefetching schemes.

4. Our schemes perform significantly better than the only other compiler-inserted prefetching scheme for pointer-based codes.

Overall, these results are quite encouraging, and they suggest that the latency problem for pointer-based codes may be addressed largely through the prefetch instructions that already exist in many recent microprocessors.

Chapter 4

Facilitating Data Locality Optimizations by Memory Forwarding

4.1 Introduction

Having demonstrated in the last chapter that *tolerating* data latency in pointer-based applications is viable through software-controlled prefetching, we address in this chapter an even more challenging but important problem: how to *reduce* latency in non-numeric applications via *locality optimizations*.

Cache performance depends on two factors: *when* data items are accessed, and *where* they exist in the address space. Therefore, locality optimizations typically do one of two things: they either restructure the *computation*, or else they restructure the *data layout*. The idea behind restructuring the computation is that given a fixed data layout, we would like to manipulate the ordering of accesses such that multiple accesses to the same data item (or cache line) occur close together in time, thereby enhancing locality [22, 134]. In contrast, the idea behind optimizing the data layout is that given that a set of data items are accessed close together in time in the original computation, we would like to actively arrange them in the address space such that: (i) we *create spatial locality* by allocating them at contiguous addresses (thereby enhancing the effectiveness of long cache lines and simplifying prefetch address generation); (ii) we *avoid cache conflicts* by ensuring that they do not reside in separate lines which map into the same cache sets; and (iii) we *avoid false sharing* [58, 128] by ensuring that items accessed by different processors fall within separate cache lines. While both approaches have received considerable attention in the past, our focus in this study is on facilitating data layout optimizations.

There are two possibilities for when we manipulate data layout. The first approach—which we call *static placement*—is to assign an object to its optimized address when it is created [19]. The second approach is to move an object (perhaps more than once) *after* it has been allocated; we refer to this latter approach as *data relocation* (or simply *relocation*). The advantage of static placement is its simplicity. The advantage of relocation, however, is that it can adapt to dynamic program behavior. Previous studies have shown that relocation-based optimizations such as *copying* [71, 127] and *clustering* [31] can offer impressive performance gains.

In general, relocation-based data layout optimizations involve the following three steps:

1. **Guaranteeing Correctness:** Either the programmer or the compiler must prove that relocating the data will *never* break the program; otherwise, the optimization is unsafe.

2. **Estimating the Cost/Benefit Tradeoff:** The potential optimization should only be performed if the performance benefit is expected to outweigh the overheads involved in relocating the data. This estimation could be based on some combination of programmer knowledge, static compiler analysis, profiling feedback, or run-time information.

3. **Generating Relocation Code:** Additional code must be inserted to perform the actual data relocation at run-time.

Despite the high performance potential of many relocation-based optimizations, the key stumbling block which often prevents them from being used in practice is the first step—i.e. *guaranteeing correctness*. To safely move data, we must guarantee that any future references to the object will find it at its new location. The fundamental problem is that updating the precise set of pointers[1] to a given object requires perfect aliasing information related to that object. In general, computing such precise information is beyond the capabilities of the compiler,[2] and is even quite difficult for the programmer for large programs. In the face of uncertainty, we must conservatively assume that relocating an object will break the program—no matter how unlikely this may seem in reality—and therefore the optimization cannot be performed.

There is one mechanism in modern systems which provides a very limited form of safe data relocation: the virtual memory system. The operating system can relocate an entire page of memory in the physical address space without breaking the program by simply copying the page and updating its virtual-to-physical mapping. One cache optimization

[1]We use the term "pointers" loosely to refer to any mechanism for generating an address pointing to the object in question.

[2]This is especially true for heap-allocated objects in languages like C. In fact, the pointer aliasing problem is theoretically undecidable.

which exploits this flexibility is *page coloring* [17, 65], whereby the operating system attempts to avoid mapping conflicts in large off-chip caches. Therefore, by adding a layer of indirection *within* the memory system, we can move data safely and transparently without any special language or compiler support. Unfortunately, the virtual memory system only provides this flexibility at the granularity of an entire page. To actively create spatial locality *within* a cache line, we must have this flexibility at a *word granularity*. However, applying standard virtual memory techniques at such a fine granularity—i.e. setting the page size to be one word—is not a viable solution, due to the enormous overheads that this would involve. (Not only would the number of page table entries and the TLB size grow enormously, but also the cache tags would have to be maintained at a word granularity.) Instead, we propose a completely different solution.

4.1.1 Our Solution: Memory Forwarding

To give software the flexibility to apply relocation-based data layout optimizations at any time without concern over violating program correctness, we propose a mechanism called *memory forwarding* which guarantees the safety of relocation at a word granularity (In our discussion, we define the "word" size to be equal to the size of a pointer.). The basic idea behind memory forwarding is that when we relocate an object, we store its new address in its old location and mark the old location so that hardware recognizes it as a *forwarding address*. Therefore if the program accidentally accesses the old address, the hardware will automatically forward the reference to the object's new location, thereby guaranteeing the correct result. Moreover, our scheme only pays the run-time overhead of an extra indirection when it is actually necessary[3]—i.e. when the alternative is to violate program semantics. In the far more common cases of references to non-relocated objects, or references that have been properly updated to point to the new addresses of relocated objects, our scheme imposes no performance overhead. The space overhead of our scheme is also low (a 1.5% fixed memory cost on a 64-bit architecture).

With memory forwarding support, the decision of whether to apply a relocation-based optimization reduces solely to evaluating its cost/benefit performance tradeoffs. In effect, memory forwarding enables software to optimistically *speculate* that when it relocates an object, it has successfully updated all pointers to that object to point to its new location. If the speculation fails, then there is a recovery cost (i.e. dereferencing the forwarding address), but the execution still proceeds correctly. Therefore, as in all forms

[3]This contrasts to an approach taken by some machines in the old days, where indirection is required for *every* single data access. For example, in the Burroughs B5000 [114], a data item is always accessed through indexing a descriptor that specifies the base address and size of the corresponding data region.

of speculation, one is gambling that the speculation is correct often enough that the benefit outweighs the cost. Another feature of our mechanism which helps improve these odds is that software can optionally specify that dereferencing a forwarding address will invoke a user-level trap that enables software to update the offending pointer to point to the object's new address. Hence software can learn from its mistakes to avoid repeating them.

4.1.2 Related Work

It is interesting to note that the work which is most closely related to our study occurred well over a decade ago in the context of architectures that directly supported the *Lisp* programming environment [92, 107, 120, 126]. Performance concerns were quite different back then: main memory was relatively small and expensive, and cache miss latencies were less problematic because the gap between microprocessor and memory speeds was dramatically smaller. (In fact, a number of microprocessor-based systems did not even have caches.) Therefore the primary concern in optimizing memory performance back then was minimizing the overall *space* requirements of a program, so that it would fit into main memory and avoid paging to disk. Two aspects of the Lisp environment made this challenging: the need to perform automatic garbage collection, and the relative space inefficiency of the ubiquitous list structures in the language. In addition, another aspect of the Lisp language which resulted in specialized hardware support was the need to determine object data types at run-time.

Although the performance goals which inspired specialized hardware and software support in these Lisp machines are quite different from our goal of improving cache performance, there are nonetheless a number of interesting overlaps between our support and some features of these earlier machines. We now discuss the connections between this previous work and our study from three different perspectives: tagged memory, garbage collection, and data layout optimizations.

Tagged Memory

To make objects self-descriptive with respect to their types, a number of Lisp architectures [92, 107, 120, 126] associated a *tag* with each memory location. As we will see later in Section 4.2.1, our memory forwarding scheme also requires a form of tagged memory to distinguish forwarding addresses from normal data. A key difference, however, is that the tags in Lisp machines provided much more functionality than in our case, and therefore they required more overhead. For example, the SPUR architecture [126] added eight bits of tag to each 32 bits of memory (a 25% overhead), whereas our scheme only requires one tag bit per 64 bits of memory (a 1.5% overhead) in a modern 64-bit architecture.

A fact that is even more relevant to our study is that a form of memory forwarding (using tagged memory) has appeared in previous Lisp machines, albeit for a very different purpose. The concept of an *invisible pointer* (which is similar to our forwarding address) was proposed twenty-four years ago by Greenblatt [46], and the Symbolics 3600 [92] used one of its tags to implement a *forwarding pointer*. The motivation behind these mechanisms was threefold: to enable the insertion of an item into a *cdr-coded* list [14], to facilitate incremental garbage collection, and to implement overlapping arrays. In contrast, our focus is on improving the cache performance of programs written in C, and therefore none of these issues apply. In essence, what we are doing is taking a very old mechanism and adapting it to a completely new purpose within the context of modern out-of-order superscalar processors.

Garbage Collection

A common feature among these Lisp machines is that they support some form of automatic garbage collection. Garbage collection algorithms involve phases where they identify two classes of data items: those that can be *reclaimed*, and those that can be *relocated*. A data item can be reclaimed when it can no longer be accessed through any pointers that are still active, and a data item can be relocated if *all* pointers to the old location can be updated to point to the new location. In both cases, the key challenge is identifying all pointers which point to the given location. In languages such as Lisp, ML, and Java, where the use of pointers is either restricted or disallowed altogether, one can solve this problem in practice. In contrast, in languages such as C and C++ which do not restrict pointer usage, one generally cannot exactly determine which pointers point to a given object, and therefore automatic garbage collection (and data relocation) is extremely difficult. Finally, it is interesting to note that a form of memory forwarding is used in *copying garbage collectors* [30, 91], whereby the forwarding addresses are used to preserve data consistency during the distinct phases when collection takes place. In contrast, our approach must preserve the forwarding addresses across the entire execution of the program, and not just during garbage collection phases.

Data Layout Optimizations

An important topic in Lisp research is how to represent list structures compactly. List compaction can be performed either separately or during garbage collection. Most of the list compaction techniques designed for Lisp [11, 14, 50, 76] involve either moving or copying the original list to a new, denser set of locations. As we discussed above, data relocation in Lisp does not pose the safety problems that we encounter in C. However, our memory forwarding support gives us the flexibility to exploit some of these same list compaction techniques—e.g., a technique called *list linearization* [33]—for the sake of improving spatial locality in C programs.

4.1.3 Objectives and Overview

We make the following contributions in this chapter. First, we propose a solution to the problem of safely relocating data at a fine granularity to improve the cache performance of programs written in languages such as C which do not support garbage collection. Although the concept of memory forwarding was proposed over two decades ago in the context of Lisp machines, to the best of our knowledge, we are the first to propose that it be adapted to facilitate a broad class of data layout optimizations to improve cache performance. Second, we discuss how memory forwarding can be implemented within modern out-of-order superscalar processors (which are quite different from the processors in which other forms of forwarding have been implemented in the past). Third, we suggest a number of optimizations which can benefit from memory forwarding. Finally, we quantitatively evaluate the benefits and overheads of our scheme by using it to apply a number of different run-time locality optimizations to a collection of non-numeric applications running on a modern superscalar processor.

The remainder of this chapter is organized as follows. We begin in Section 4.2 with an overview of memory forwarding and how it can be used. Section 4.3 discusses issues related to implementing memory forwarding in a modern processor. Sections 4.4 and 4.5 present our experimental methodology and experimental results, respectively, to demonstrate the usefulness of the mechanism. Finally, we conclude in Section 4.6.

4.2 Memory Forwarding

We now discuss the basic concepts behind memory forwarding, how to handle a complication arising from operations depending on pointer values, a number of applications of forwarding, and some issues related to its performance.

4.2.1 Basic Concepts

Memory forwarding enables *aggressive* yet *safe* data relocation. As we mentioned earlier in Section 4.1.1, the basic idea is to store the new address of an object into its old memory location, and to mark this old location as a *forwarding address*. Whenever a

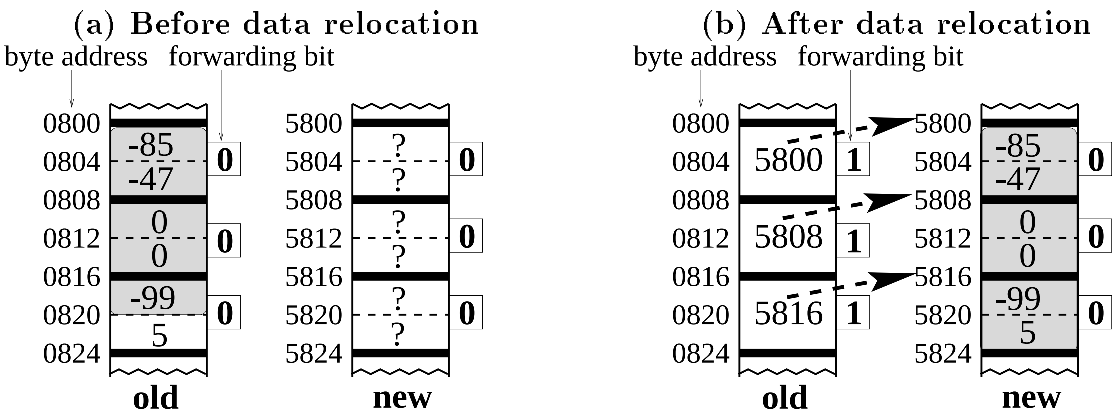


Figure 4.1: Example of data relocation with memory forwarding (a memory word is 8 bytes, and addresses are in decimal).

forwarding address is accessed, the hardware (probably in cooperation with software) will automatically dereference that location to find the object at its new location.

There are three implications of this mechanism in terms of memory storage. First, the minimum unit of data that can be relocated is the width of a pointer—which we refer to as a "word" throughout this chapter—since otherwise there would not be enough space to store the forwarding address.[4] Note that it is possible to relocate byte-sized objects—this simply means that enough neighboring bytes must be moved at the same time to comprise an entire word. Second, a chunk of data that is relocated must be word-aligned, so that the alignment of the forwarding address is predetermined. Note that this still allows us to perform byte-sized loads and stores to forwarded objects—the byte offset into the new location is simply assumed to be the same as it was at its original address. We consider these first two restrictions to be quite minor, especially given that our only option for safe relocation today is page-sized, page-aligned chunks of data. Finally, to enable the hardware to distinguish forwarding addresses from regular data, we attach a one-bit tag (called a *forwarding bit*) to each word in memory. For a 64-bit architecture, this results in a space overhead of only 1.5%, and therefore is reasonably efficient.

Figure 4.1 shows a simple example of how the memory contents and forwarding bits are modified upon data relocation. Assume that we have a 64-bit architecture, and that we would like to relocate five 32-bit elements from addresses `0800-0816` to addresses `5800-5816` (these addresses are in decimal notation). Figure 4.1(a) shows the memory contents

[4]One could imagine creating a more elaborate scheme for compressing the size of forwarding address pointers (e.g., by restricting the distance between the old and new address to something that fit within its former size), but this would involve additional complexity and fancier tag storage, so we do not consider such an approach further.

and forwarding bits before relocation (note that none of the forwarding bits have been set). To relocate a word, we first copy it to its new location, and then we *simultaneously* write its new address into its old location and set the corresponding forwarding bit at the same time. Figure 4.1(b) shows the state of memory after the relocation. Notice that to relocate the 32-bit subword at address `0816`, we must also relocate the 32-bit subword at address `0824` (which contains the value 5) along with it. After the relocation, a 32-bit load of the subword at address `0804` will be forwarded to address `5804`—which is computed by adding the forwarding address (`5800`) to the byte offset within the word (`4`)—thereby returning the correct value of `-47`.

To simplify our discussion throughout the remainder of the chapter, we now define two terms which we will use frequently:

Initial address: The address of the *first* location accessed by a memory reference. For example, in Figure 4.1(b), the initial address of a write to word `0816` is `0816` itself.

Final address: The address of the *last* location accessed by a memory reference. For example, in Figure 4.1(b), the final address of a write to word `0816` is `5816`. When data is not forwarded, the final address equals the initial address.

4.2.2 Handling Operations That Depend on Pointer Values

In addition to preserving the correctness of pointer *dereferences*, another concern in data relocation is preserving the correctness of operations that depend on the pointer *values* themselves. Such operations include pointer comparisons and saving pointer values to a file. Applying data relocation to programs with these operations is unsafe, even with memory forwarding, since the very act of relocating data changes the data address. In fact, the only general solution is to disallow relocation of objects whose addresses could affect the outcomes of such operations. Fortunately, after considering how pointer values are typically used in normal C programs, we can identify an important special case: equality tests of pointers that potentially point to the same type of relocatable objects[5]. And the good news is that we can preserve the correctness of these tests even in the presence of data relocation by comparing the pointers with respect to their *final* addresses. Although our memory forwarding hardware does not perform this check automatically, the compiler can easily insert additional instructions (described in Section 4.3) to look up the final addresses for these comparisons. We have implemented such a compiler pass,

[5]Other kinds of pointer comparisons such as $p < q$ or $p \geq q$ may be used to test array-element addresses but should be rarely used for data structures, like linked lists or trees, that are most relevant to relocation.

and the resulting software overhead is included in our performance results. As we will see later in Section 4.5, this overhead does not present a problem.

So far, we have assumed that the compiler knows whether a pointer involved in an operation points to a relocatable object so that it can safely ignore all cases involving no such pointers. However, this assumption cannot be realized by simply inspecting the *declared* types of the pointers in question, because the type coercion feature of C can confuse the compiler. Instead, the compiler can make use of some type-inferencing tools [9, 101, 122, 123] that perform relatively simple pointer analyses to infer operand types. Recent advances in program analysis have demonstrated that these tools can handle large programs efficiently without sacrificing much accuracy. Nevertheless, it is essential to realize that though these tools can tell the types of objects, they do not solve the pointer aliasing problem itself (e.g., most of them are not intended to distinguish different elements within the *same* linked data structures). Therefore, even with the help of these tools, we still need a mechanism like memory forwarding to provide the correctness guarantee for aggressive data relocation.

4.2.3 Applications of Memory Forwarding

While the act of dereferencing a forwarding address clearly does not improve performance on its own, the advantage of memory forwarding support is that it enables a wide range of data layout optimizations which can enhance cache performance. Not only are these optimizations useful for mitigating the impact of memory latency, they can also be used to conserve memory *bandwidth*. We now briefly describe some of these potential optimizations.

Improving Spatial Locality

A straightforward method of actively improving spatial locality is to take data items which are accessed close together in time, but which are scattered sparsely throughout the address space, and pack them into adjacent memory locations. This form of data packing makes cache lines much more effective, and it can potentially reduce the number of capacity, compulsory, and conflict misses. Not only does this improve performance, it can also reduce memory bandwidth consumption, which in turn can help reduce power consumption (which is becoming an increasingly important concern).

Aside from traditional processor cache hierarchies, there are other situations where enhancing spatial locality is particularly important. For example, there have been recent proposals to integrate a processor and memory on the same chip [104, 113]. To utilize the full memory bandwidth of such an architecture, it is important to exploit large amounts of

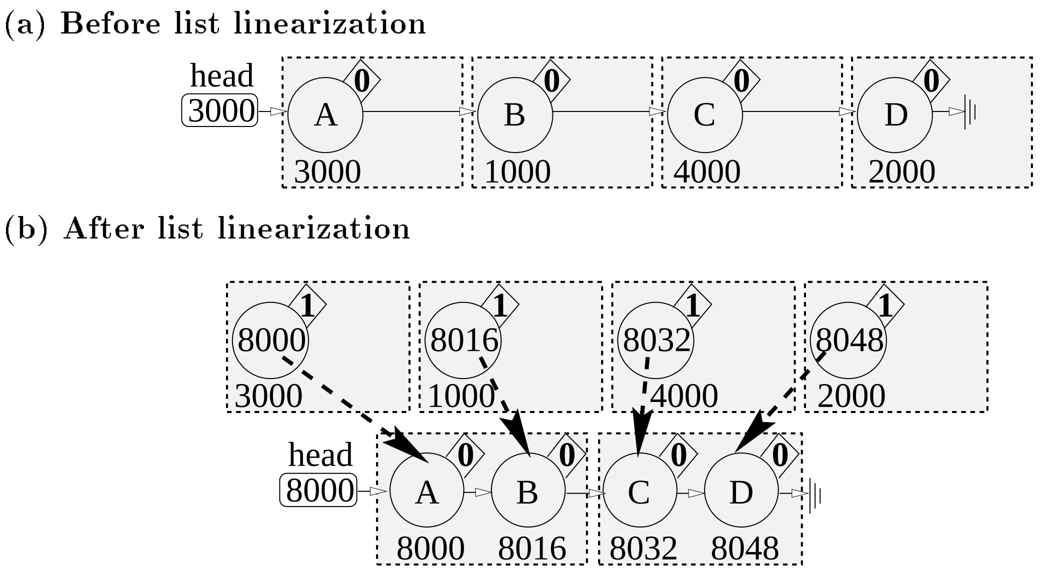


Figure 4.2: Example of list linearization with memory forwarding, assuming that cache lines and list elements are 32 and 16 bytes long, respectively (addresses are in decimal).

spatial locality—e.g., the Saulsbury *et al.* study [113] used 512-byte cache lines. Another portion of the memory hierarchy where spatial locality is extremely important is on disks, where seek times are enormous relative to the time it takes to transfer an additional byte at a contiguous address. Although both of these scenarios are beyond the scope of our performance study, we believe that they also provide rich opportunities for exploiting memory forwarding to enhance performance.

A good example of a technique which uses packing to enhance spatial locality is *list linearization*. As we will see later in Section 4.5, this technique can offer dramatic performance improvements. The idea behind list linearization is to relocate the nodes of a linked list so that they reside in contiguous memory locations. Depending on whether the list structure continues to change over time, the linearization process can be invoked either just once, or else periodically to adapt to the changing structure. Although list linearization can potentially offer large performance gains, it is very difficult to safely use this optimization in practice for general C programs due to the possibility of pointers outside of the linked list itself pointing to list elements. Fortunately, with memory forwarding support, we can apply list linearization at any time without worrying about whether all potential pointers to list elements have been properly updated.

Figure 4.2 shows an example of list linearization with memory forwarding. Before linearization, the four nodes of the list (i.e. nodes `A`, `B`, `C`, and `D`) are scattered throughout memory such that they reside in four separate cache lines, as shown in Figure 4.2(a). List linearization packs the four nodes into a contiguous memory region starting at location `8000`, as shown in Figure 4.2(b). As a result, the four relocated nodes occupy only two cache lines, rather than four, thereby potentially eliminating half of the cache misses due to this list as we continue to revisit it. Note that the forwarding addresses and forwarding bits have been set properly such that we will still maintain correct execution even if a stray pointer accesses a list element at its old address. However, we expect that most accesses to the list will find it directly at its new address, thereby enjoying the enhanced spatial locality.

Increasing Prefetching Effectiveness

As we discussed in Chapter 3, the effectiveness of prefetching pointer-based data structures is largely limited by the pointer-chasing problem. One of the techniques we proposed in Chapter 3 to generate prefetch addresses early enough is data-linearization prefetching (see Section 3.2.2), which maps heap-allocated nodes that are accessed close together in time into adjacent memory locations. Memory forwarding nicely supports this technique by allowing such data mapping to be done dynamically during the execution without worrying about the safety problem. In fact, this was our original motivation for studying memory forwarding.

Reducing Cache Conflicts

Cache conflict misses can be reduced through either hardware or software techniques. Hardware-based techniques such as increasing the set associativity or adding victim caches [61] can handle small degrees of conflict, but their effectiveness declines as more conflicting addresses map to the same cache entries. In contrast, software-based techniques such as *data copying* [71, 127] and *coloring* [31] tackle the problem from its root by rearranging the data layout; hence they can potentially resolve even severe degrees of conflict. *Data copying* [71] was originally proposed to reduce conflict misses within tiled (or "blocked") numeric applications. Since a given tile is reused many times after it is brought into the cache, it is particularly problematic if different elements *within* the tile conflict with each other. To avoid this problem, the data copying optimization first copies a tile to a contiguous set of addresses in a temporary array before using it; since these locations do not conflict with one another, the problem is eliminated. Another technique called *data coloring* [31] was proposed as a method of reducing conflict misses in pointer-based data structures. The idea is to partition the cache into logically separate regions (or *colors*). By relocating data structure elements which are accessed close together in time to separate regions of the cache, conflict misses can be avoided. Memory forwarding can help facilitate both copying and coloring techniques by guaranteeing that they are safe.

Reducing False Sharing

In cache-coherent shared-memory multiprocessing systems, *false sharing* [58, 128] occurs when two or more processors access distinct data items which happen to fall within the same cache line (which is the unit of coherence), and at least one access is a write. False sharing can hurt performance dramatically as the line ping-pongs between processors despite the fact that no real communication is taking place. By relocating those unrelated data items to distinct cache lines, false sharing can be avoided. Memory forwarding would be especially helpful in avoiding false sharing in *irregular* shared-memory applications, where proving that data items can be safely relocated is difficult.

In summary, memory forwarding enables a broad range of relocation-based optimizations; we have presented just a partial list of such optimizations. We would also like to emphasize that these optimizations are applicable not only to caches but also to the other levels of the memory hierarchy. For example, we can apply data relocation to improve the spatial locality within pages (and hence on disk) for out-of-core applications.

4.2.4 Performance Issues

A relocation-based optimization will improve overall performance if two conditions hold: (i) the new data layout actually provides better memory performance than the original layout; and (ii) the gain in the memory performance outweighs the optimization overhead. This overhead includes the extra execution time due to actually relocating the data, and may also include forwarding overhead if any references actually need to be forwarded after the relocation. While the overhead of relocating the data may seem to be a concern at first glance, our experimental results indicate that it is usually not a problem because relocation is invoked infrequently and modern processors are fast enough to afford this extra computation. In addition, we find that the performance overhead of forwarding is negligible in many cases because most data references are updated properly and do not need to be forwarded. We observe that the real performance concern is ensuring that the reorganized data layout actually delivers higher memory performance than the original layout.

(B, R) = `Unforwarded_Read(word*` A`)`:[a] Read the forwarding bit associated with the word at address A into register B *and* read the word's value into register R *atomically*, with forwarding disabled. If the forwarding bit is set, this is a forwarding address; otherwise this is a regular data value.

*A = `Unforwarded_Write(bit` B, `register word` R`)`:[b] Set the forwarding bit associated with the word at address A to B's value *and* write R's value into the word *atomically*, with forwarding disabled.

[a] If an instruction of the underlying ISA cannot have three operands, we can hardwire B to a special register that is set by every `Unforwarded_Read` automatically.

[b] Again, if three-operand instructions are not allowed, we can hardwire B to a special register that can be set prior to the `Unforwarded_Write`. Alternatively, we can have two separate instructions for `Unforwarded_Write(`R,0`)` and `Unforwarded_Write(`R,1`)`.

Figure 4.3: Proposed instruction set extensions to support memory forwarding (C syntax is used to improve readability).

4.3 Implementation Issues

We now discuss the support that we need from the instruction set, the hardware, and the software to implement memory forwarding in modern superscalar processors.

4.3.1 Extensions to the Instruction Set Architecture

To exploit memory forwarding, the machine must have some way to manipulate the forwarding information—i.e. the forwarding addresses and the forwarding bits. Rather than taking a purely hardware-based approach, we propose to extend the underlying instruction set architecture (ISA) by adding a few instructions which will allow software to manipulate the forwarding information directly. The advantages of this approach are its programmability and flexibility. In addition, we expect the software overhead to be low since forwarding information changes relatively infrequently.

Figure 4.3 shows our proposed ISA extensions, which consist of two new instructions. `Unforwarded_Read` and `Unforwarded_Write` allow software to manipulate memory with the forwarding mechanism disabled. For example, in Figure 4.1(b), a normal `Read` (i.e. with the forwarding mechanism enabled) of the word at address `0808` will get the forwarded value of `0`, but an `Unforwarded_Read` of the same word will get `5808`, which is the forwarding address. Both new instructions access the forwarding bit and the word *atomically* in order to preserve data consistency, especially in multiprocessor applications.

To demonstrate how software can make use of these new instructions, Figure 4.4(a) shows two procedures for relocating a data object of size `n_words` from `src` to `tgt`, and

(a) Data Relocation

```
// src = address of the object before relocation
// tgt = address of the object after relocation
// n_words = number of words to relocate
void Relocate(word* src, word* tgt, int n_words) {
  boolean relocated;
  word content;
  (relocated, content) = Unforwarded_Read(src);
  while (relocated) { // loop until the final address is reached
    src = (word*)content; // content is in fact a forwarding address
    (relocated, content) = Unforwarded_Read(src);
  }
  actualRelocate(src, tgt, n_words);
}
 void actualRelocate(word* src, word* tgt, int n_words) {
  for(; n_words > 0; --n_words) { // relocate each word in the object
    word temp;
    boolean dontCare;
    // save the content of src
    (dontCare, temp) = Unforwarded_Read(src);
    // copy the original content of src to tgt
    *tgt = Unforwarded_Write(temp, 0);
    // setup the forwarding address and forwarding bit
    *src = Unforwarded_Write(tgt, 1);
    // prepare for the next word
    src += 1; tgt += 1;
  }
}
```

(b) List Linearization

```
extern char* memory_pool; // a pool of space for data relocation
// head_handle = address of the list head pointer
void ListLinearize(node ** head_handle) {
  node ** handle, *tgt;
  handle = head_handle; // start from the list head
  while (*handle) {
    // grab space from the pool
    tgt = (node*)memory_pool;
    // increment the pool pointer
    memory_pool += sizeof(node);
    // relocate the node pointed-to by handle to the address stored in tgt
    Relocate(*handle, tgt, sizeof(node)/sizeof(word));
    // append the relocated node to the linearized list
    *handle = tgt;
    // prepare for next node
    handle = &(tgt→next);
  }
}
```

Figure 4.4: Procedures using the proposed ISA extensions to implement (a) data relocation and (b) list linearization.

then storing `tgt` as the forwarding address into `src`.[6] Procedure `Relocate()` loops until a clear forwarding bit is read so that `tgt` will be appended at the end of the forwarding chain (if any). Figure 4.4(b) shows a procedure called `ListLinearize()` (which we will use frequently later in our experiments) which calls `Relocate()` to perform list linearization. The parameter `head_handle` is the address of the list head. Note that the *address* of the list head (rather than its value) is passed into `ListLinearize()` because we want to modify the list head to point to the new locations after the relocation is performed. (This effect was illustrated earlier in Figure 4.2(b), where the value of `head` is changed to `8000` after the linearization.) By doing so, the next time that the list is accessed via the list head, the new locations will be accessed directly without touching the old locations. Finally, note that in Figure 4.4(b), the new locations for the relocated nodes are allocated from a pool of contiguous memory, thereby creating spatial locality.

4.3.2 Hardware Support

We now discuss the hardware modifications necessary to support memory forwarding. The key insight which helps us keep the hardware simple is that references which actually require forwarding are expected to occur *rarely* (if ever). The forwarding mechanism is simply a safety net which allows us to continue to preserve program correctness in case the unexpected happens. In other words, we can treat forwarding as an *exception*. We will design the hardware to be fast in the common case—i.e. a normal, non-forwarded reference—and we are less concerned about the performance penalty when forwarding is actually invoked, since that is rare. Hence a legitimate option is to use a processor's normal exception handling mechanism to implement forwarding.

One hardware requirement that was mentioned earlier in Section 4.2.1 is that we need tagged memory. A number of systems which supported tagged memory have been built in the past [92, 126]. One difference with our scheme (as discussed earlier) is that we require less tag storage overhead than previous schemes; otherwise, it is quite similar. We now discuss the more novel features of our hardware support in greater detail.

Dereferencing Forwarding Addresses

In the presence of memory forwarding, the data referencing mechanism must be able to follow *forwarding chains* of arbitrary lengths. More specifically, when a memory word is accessed by a data reference, its forwarding bit is tested. If this bit is set, then the

[6] We assume a uniprocessor environment in this example. In a multiprocessor environment, we must make sure that no one modifies the word between the two `Unforwarded_Write`'s in `actualRelocate`. This can be accomplished by a number of mutual exclusion mechanisms.

original data address will be replaced by the contents of the word just accessed (which contains a forwarding address), and a new memory access using the forwarding address will be launched. This process repeats until a clear forwarding bit is read (we will discuss how cycles might be handled later in this section), at which point the data reference can proceed as usual. One option is to implement this dereferencing mechanism purely within hardware; another is to implement it using a software-based exception handler (where the exception is triggered by accessing a word with its forwarding bit set). With the ISA extensions that we propose, it would be straightforward for a software handler to chase the forwarding pointer chain. Although the forwarding bit cannot be tested until the memory location is brought into the primary cache, this is no different from the delays associated with checking ECC or parity bits.

Data Dependence Speculation

One consequence of memory forwarding is that we do not know the final data address of a given reference until the reference is nearly completed. This delayed generation of the final address poses a potential problem in out-of-order superscalar machines. These machines normally allow a load access to proceed before an earlier store, provided that the load and store are to different addresses. If either address is unknown, the conservative approach is to delay the load until both addresses are resolved. With memory forwarding, since the final address of a store is not known until the store actually completes, this delay would cause the conservative approach to never execute a load ahead of an earlier store.

Fortunately, there is a solution to this problem. A technique called *data dependence speculation* [32, 93, 94, 131] allows a load to speculatively execute before an earlier store, even if the store address is unknown. If it turns out that the load was not dependent on the store, then the speculation succeeds; otherwise, a true dependence has been violated, and the effects of the incorrect speculation must be undone. Recent out-of-order superscalar processors [55, 57, 74] have already implemented the *unselective* form of data dependence speculation. (The *unselective* approach means that a load is speculated whenever possible; in contrast, the *selective* approach only performs the load speculative if it is *predicted* to not be dependent on preceding stores [93].) With support for data dependence speculation (either unselective or selective), we can speculate that the final address of a reference will be the same as its initial address (i.e. we do not expect the reference to be forwarded), and therefore the delayed final-address generation will not degrade performance in the common case where the reference is not forwarded after all. If forwarding does occur, then our speculation would only be incorrect in the case where the load and store had different initial addresses but the same final address. In our experiments, we observed that incorrect data dependence speculation almost never occurred; hence it appears to be a very effective solution to supporting memory forwarding.

Handling Forwarding Cycles

A *forwarding cycle* is created when software erroneously inserts an address more than once into a forwarding chain. The hardware must have some mechanism for detecting and breaking forwarding cycles; otherwise, the machine could be stalled forever chasing the forwarding chain. Detecting forwarding cycles *accurately* is an expensive operation; for each hop, the hardware would have to match the current forwarding address against all previous forwarding addresses dereferenced by the same data reference. Because of this high cost—and also because we expect forwarding cycles to be extremely rare—we would prefer that the hardware instead perform a fast but possibly inaccurate check for a cycle during normal execution, and only perform accurate cycle detection when it is necessary. One possibility is to predetermine a limit on the number of forwarding hops that are allowed for a given data reference. We simply maintain a counter (which could be implemented either in hardware or in software) to keep track of the number of forwarding hops performed so far, and when this count exceeds the limit, we raise an *exception*. The corresponding software exception handler will then perform an accurate cycle check. If it is a false alarm, then we will reset the counter and resume execution; otherwise, the execution will be aborted.

Providing User-Level Traps Upon Forwarding

In addition to the *system-level* exception handlers which might be provided to support the dereferencing of forwarding addresses and the detection of forwarding cycles, it may also be useful to provide a lightweight *user-level* trapping mechanism that would be invoked upon accessing a forwarded location. Such a mechanism would be useful for allowing the application to tune its own performance in the following two ways. First, one could write a *profiling tool* to gather forwarding-related statistics for the purpose of improving the performance of a future execution of the program. For example, one might record which instructions experienced forwarding for the sake of eliminating that forwarding in future runs of the program. Second, a user-level trap handler could be used to optimize away forwarding (and thereby improve performance) *on-the-fly*. For example, one could write a tool that updates stray pointers on-the-fly to point directly to their correct final addresses, thereby avoiding the need to invoke the forwarding mechanism again. (Note that one must have application-specific knowledge in order to do this.).

Our user-level trapping mechanism might be implemented in a similar fashion to the trapping version of *informing memory operations* proposed by Horowitz *et al.* [53]. The primary difference would be that informing memory operation traps are invoked upon cache misses, and forwarding traps would be invoked upon detecting a forwarding address. Horowitz *et al.* [53] proposed a number of ways to specify user-level trap handlers to the hardware, ranging from a single handler for the entire program to a unique handler for each memory instruction. In our case, for example, there may be a single *profiling* handler for the entire program, but a fairly large number of *pointer updating* handlers for different memory instructions.

4.3.3 Software Support

Having discussed the hardware support for memory forwarding, we now focus on its impact on *software*.

Initialization of Forwarding Bits

The forwarding bit of a memory word must already be clear when it is used by a program for the first time. To guarantee this, the operating system must perform an `Unforwarded_Write(0,0)` operation on all words in a region of memory to initialize it before making that memory available to an application.

Deallocating Forwarded Data

When an object is deallocated, all memory reachable via the chain of forwarding addresses for that object should be deallocated as well. A simple way to accomplish this is to create a *wrapper* memory-deallocation routine which first deallocates all of the memory allocated on the forwarding chain, and then calls the original memory-deallocation routine, which can be either a system-provided procedure (e.g., `free()` in C and `delete() in C++`) or a user-defined procedure if the program performs its own memory management.

Memory Alignment

Since the minimal granularity of memory forwarding is a word, software must ensure that two different objects which are being relocated to two different destinations do not share the same word, since we cannot store two different forwarding addresses in that same word. In other words, relocatable objects must be word-aligned. Enforcing this alignment can be accomplished either by specifying the alignment to the memory allocator for dynamically-allocated objects, or else by tuning the alignment option in the

(a) Original Codes

```
// start = address to begin with
// end = address to finish with
inc_array(int* start, int* end)
{
  int* a;
  for (a = start; a != end; a++)
    ++*a;
}
// first = address to begin with
// last = address to finish with
inc_list(node* first, node* last)
{
  node* p;
  for (p = first;
    p != last;
    p=p->next)
    ++p->data;
}
```

(b) With Codes Added to Preserve Outcomes of Pointer Comparisons

```
// start = address to begin with
// end = address to finish with
inc_array(int* start, int* end)
{
  int* a;
  for (a = start; a != end; a++)
    ++*a;
}
// first = address to begin with
// last = address to finish with
inc_list(node* first, node* last)
{
  node* p;
  for (p = first;
    // preserve the outcome of this test
    final_address(*p) != final_address(*last);
    p=p->next)
    ++p->data;
}
```

Figure 4.5: Example of adding codes to preserve the outcomes of pointer comparisons.

compiler if some relocatable objects are statically allocated. In most compilers—e.g., the MIPS C compiler that we used in our experiments—aggregate objects are already aligned to word boundaries by default.

Preserving Outcomes of Pointer Equality Tests

The compiler is responsible for replacing all pointer equality tests that could be affected by relocation with explicit code to look up and compare final addresses. As we have discussed in Section 4.2.2, type-inferencing tools [9, 101, 122, 123] can help the compiler avoid inserting these more costly comparisons by ignoring cases where pointers cannot point to relocated objects. Figure 4.5 shows an example of adding codes to preserve the outcomes of pointer equality tests. In the original code shown in Figure 4.5(a), there is a pair of underlined pointer comparisons. Assuming that list nodes can be relocated but array elements cannot, the compiler will replace the pointer comparison `p != last` by a new one `final_address(*p) != final_address(*last)`, where `final_address(m)` computes the final address of the memory reference `m`. The comparison `a != end` needs not be replaced since the compiler knows from type inference that `a` and `end` cannot point

Table 4.1: Application characteristics.

Name	Description	Source	Input Data Set	Optimizations Applied
BH	Barnes-Hut's N-body force calculation algorithm	Olden [21]	4K bodies	Subtree clustering
Compress	Compresses and decompresses file in memory	SPEC95	A file of 150K characters	Array merging
Eqntott	Translation of boolean equations into truth tables	SPEC92	int_pri_3.eqn	Packing of hash table elements
Health	Simulation of the Columbian health care system	Olden	max. level = 5 max. time = 500	List linearization
MST	Finds the minimum spanning tree of a graph	Olden	1K nodes	List linearization
Radiosity	Virtual image rendering using hierarchical radiosity	IRISA [90]	A scene consisting of 10 lightly furnished rooms	List linearization
SMV	A symbolic model checker	CMU [89]	The "dme2.smv" file provided in the package	List linearization
VIS	A verification and synthesis system for finite-state hardware systems	The VIS group [15]	A reduction of the 8 queens problem to combinational equivalence checking	List linearization

Table 4.2: General run-time statistics. "Insts Grad." is the total instructions graduated. "Loads Grad." is the total loads graduated. The percentages of loads that were found in each of the four possible places are shown under "Where Loads Were Found", where "Combined" are loads that were combined with other in-flight references. "Average Load Miss Penalty" includes penalties of both full misses and partial misses.

Benchmark	Insts Grad.	Loads Grad.	Where Loads Were Found				Average Load Miss Penalty (cycles)
			D-Cache	Combined	S-Cache	Memory	
BH	1902 M	437 M	96.54%	1.53%	1.79%	0.13%	16.8
Compress	521 M	96 M	89.04%	0.63%	9.88%	0.46%	16.0
Eqntott	1825 M	204 M	89.52%	5.41%	4.50%	0.57%	21.6
Health	209 M	58 M	69.38%	0.88%	13.54%	16.20%	46.1
MST	329 M	53 M	90.80%	2.64%	2.25%	4.31%	56.4
Radiosity	4341 M	1065 M	94.21%	2.71%	2.81%	0.26%	16.7
SMV	294 M	61 M	87.71%	3.68%	4.94%	3.67%	52.6
VIS	411 M	97 M	86.16%	1.04%	10.28%	2.52%	26.5

to relocated objects. The code for implementing `final_address(m)` is already included in the procedure `Relocate()` shown in Figure 4.4.

4.4 Experimental Framework

To evaluate the potential performance benefits of memory forwarding, we modeled it in a modern processor and used it to enable a number of relocation-based optimizations which we applied to a collection of *non-numeric* applications. These applications were chosen because they could potentially be speeded up by these optimizations according to some profiling information but compilers are unable to guarantee the safety. The goals of these optimizations were improving spatial locality and prefetching effectiveness. Since current compiler technology does not support these optimizations (mainly because their safety cannot be proven), we added these optimizations to the applications manually. Table 4.1 describes the eight applications used in our experiments along with the optimizations that we applied. Some general run-time statistics are also shown in Table 4.2. All applications were run to completion in our simulations.

We added our proposed ISA extensions to the underlying MIPS ISA by making use of a few machine instruction sequences that never appear in ordinary programs (e.g., loading a value into a register which is hardwired to the value zero). We modeled the full performance effects of maintaining and dereferencing the forwarding addresses. Since our simulator is essentially trace-driven, the data reference addresses that it receives from the trace have not taken memory forwarding into account. To emulate the forwarding effect accurately, our simulator records *two* pieces of information. First, it records every memory word in the application that has been specified as "forwarded" along with its forwarding address. Second, it records whether a pointer that had been pointing to a given object gets updated to the object's new address if that object is relocated. For example, in Figure 4.2(b), the simulator must record the fact that `head` points to location `0800` rather than location `3000` after the linearization. The simulator needs this second piece of information to decide whether forwarding address dereferencing is really needed when a word is accessed (according to the unmodified address in the trace) which has its forwarding bit set. If the word is accessed through a pointer which is already pointing to the new address, then it is not necessary to dereference the forwarding address.

We implemented unselective data dependence speculation in our simulator. An ambiguous data dependence is stored in a table until the two final data addresses involved in the dependence are determined. If the dependence is incorrectly speculated, then the simulator will then re-execute all instructions after (and including) the instruction which had violated the dependence. We replaced the memory deallocation calls in the applications by calls to our own memory deallocator which first checks whether there is any memory residing in forwarding chains which must be freed. We wrote a compiler

Table 4.3: The five line sizes and the corresponding miss latencies used in the experiments.

Line Size (bytes)	Primary-to-Secondary Miss Latency (cycles)	Primary-to-Memory Miss Latency (cycles)
32	12	75
64	18	93
128	30	129
256	54	201
512	102	345

Table 4.4: Forwarding-related statistics.

Benchmark	Average Number of Forwarding Hops	Instruction Overhead of Optimizations	Space Overhead
BH	0.00	4.1%	1.7MB (109%)
Compress	0.00	5.4%	0.5MB (1%)
Eqntott	0.00	0.1%	0.5MB (23%)
Health	0.00	7.7%	4.7MB (184%)
MST	0.00	10.0%	12.0MB (48%)
Radiosity	0.00	0.05%	0.6MB (6%)
SMV	0.06	26.6%	2.2MB (24%)
VIS	0.00	8.5%	14.9MB (202%)

pass in SUIF [133] that automatically determined which pointer comparisons needed to be replaced by final address comparisons. The overhead of executing these replaced comparisons is included in our simulations.

The processor model used for the experiments in this chapter is identical to the one used in Chapter 3 (the simulation parameters are already shown in Table 3.5 on page 83). To study how successfully spatial locality is exploited, five line sizes—ranging from 32B to 512B—were used in our experiments, along with five corresponding sets of miss latencies (longer lines have longer transfer times). They are shown in Table 4.3.

We compiled our applications with `-O2` optimization using the standard MIPS C compilers and the SUIF compiler [133] under IRIX 5.3. For the experiments which required the insertion of software prefetches into the source code, we used the SUIF compiler; otherwise, the MIPS compiler was used.

4.5 Experimental Results

We now present results from our simulation studies. We start by evaluating the performance of a number of aggressive locality optimizations enabled by memory forwarding

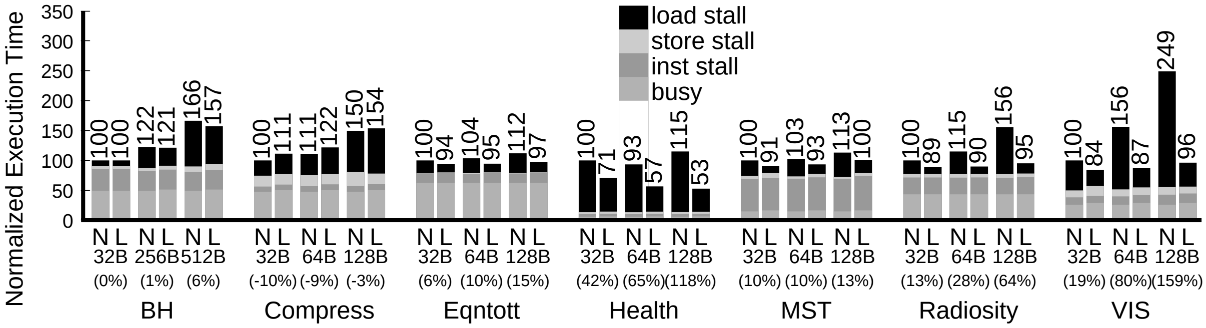


Figure 4.6: Performance of locality optimizations for various cache line sizes (**N** = not optimized, **L** = locality optimized).

(which we simply refer to as *locality optimizations*). Next, we study the impact of these optimizations on prefetching effectiveness. We then examine the details of individual applications, explaining the optimizations that we applied to each application. Finally, we study the performance impact of forwarding overhead for one of the applications.

4.5.1 Performance of Locality Optimizations

Before studying the performance of locality optimizations, we first look at the forwarding-related statistics reported in Table 4.4. First, we find that memory references in seven of the eight applications do not need to be forwarded and therefore the execution overhead of forwarding does not matter for them. The only exception is `SMV`, where on average a 0.06 forwarding hop is required for a reference. As a result, `SMV` is the only application that is affected by forwarding overhead. Second, the instruction overhead of our locality optimizations is usually only a small fraction of the total instruction count in the unoptimized cases (the 27% increase in the instruction count of `SMV` is due largely to the computation of final addresses at pointer equality tests). So, overall, the actual impact of this overhead on the execution time is quite small, as we will see shortly. Finally, the "*Space Overhead*" column in Table 4.4 shows the amount of virtual memory space needed to accommodate relocated data (the number in the parentheses alongside is the percentage increase with respect to the amount of memory[7] required by the original program). Despite the relatively large percentage increases in virtual memory size in some of these programs, the absolute amount (ranging from 0.5MB to 14.9MB) presents little problem in modern machines, and the simulation results include the impact of this overhead on performance.

Figure 4.6 shows the performance of our locality optimizations for various cache line

[7]This was measured as the *maximum* amount of virtual memory ever reserved by the program.

sizes. Seven of our eight applications are included in Figure 4.6. We will show the performance of `SMV` separately, later in Section 4.5.4. For each application in Figure 4.6, we show three line sizes, each of which has two cases: the bar on the left (**N**) is the original case where no locality optimization is applied, and the bar on the right (**L**) is the case with locality optimizations. For all applications except `BH`, the three line sizes used are **32B**, **64B**, and **128B**. For `BH`, we instead use line sizes of **32B**, **256B**, and **512B**, because the optimization applied to `BH` (subtree clustering) requires a cache line containing at least two tree nodes, and this requires cache lines longer than 128B (this optimization will not be turned on for lines shorter than 256B, and that is why the **N** and the **L** bars are identical for the **32B** line size in `BH`).

Each bar in Figure 4.6 represents execution time normalized to the **N** case of the **32B** line size, and is broken down into four categories explaining what happened during all potential graduation slots (explaination for these four categories can be found in Section 3.5.1 on page 84). In addition, there is a percentage in parentheses below each pair of bars representing the speedup of the optimized over the unoptimized case for the given line size.

Our first observation from Figure 4.6 is that performance generally degrades when line size increases, especially for the unoptimized cases. This trend is due to a lack of spatial locality in these applications, which means that longer lines offer little performance advantage. Fortunately, our locality optimizations (which are enabled by memory forwarding) improve the spatial locality of these application significantly. As we see in Figure 4.6, the optimized cases outperform the unoptimized cases for the same line sizes in all applications except `Compress`, and the speedups increase along with line size. The performance improvement can be dramatic—with 128B lines, `Health` and `VIS` enjoy more than twofold speedups. Among our optimizations, list linearization is particularly powerful since it improves the performance of `Health`, `MST`, `Radiosity`, and `VIS` substantially. It is interesting to note that in `Health`, the absolute performance of the optimized cases increases along with line size. This is due to the prefetching benefits of long cache lines after spatial locality is greatly improved. `Compress` is an exceptional case where the locality gets worse in the optimized cases. We also observe from Figure 4.6 that the instruction overhead of these locality optimizations is usually low (as they only slightly increase the total of the "*busy*" and "*inst. stall*" categories), which suggests that these optimizations could be invoked even more frequently during the execution to further improve the data layout.

While execution time is the most important performance metric, further insight can also be gained by examining the impact on total cache misses. Figure 4.7(a) shows the

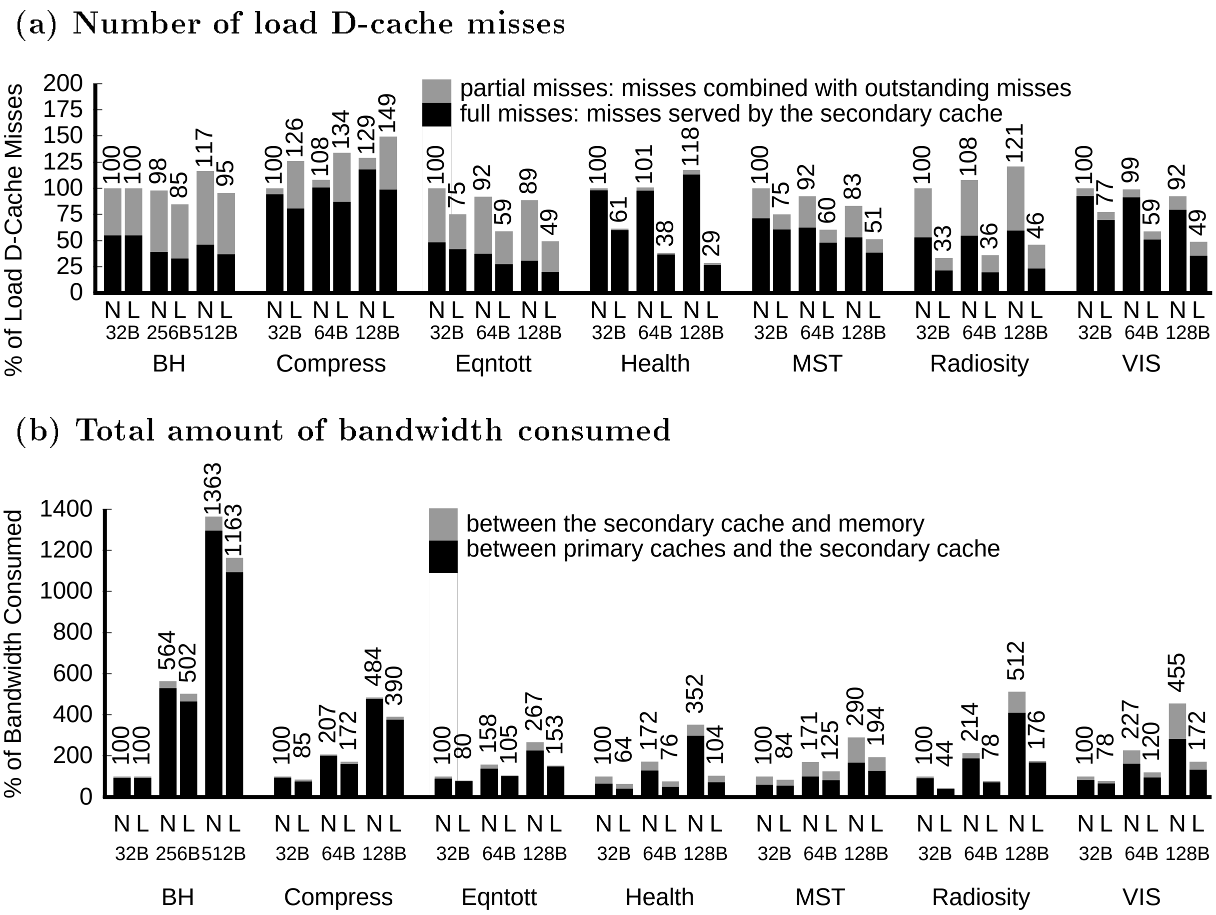


Figure 4.7: Additional performance metrics for the impact of locality optimizations (**N** = not optimized, **L** = locality optimized). The y-axes are normalized to the **N** cases of the **32B** line size.

number of load D-cache misses in the unoptimized and optimized cases for different line sizes. Each bar is normalized to the **N** case of the **32B** line size, and is divided into two categories indicating how a D-cache miss is serviced. A *partial miss* is a D-cache miss that combines with an outstanding miss to the same line, and therefore does not necessarily suffer the full miss latency. A *full miss*, on the other hand, does not combine with any access and therefore suffers the full latency. Figure 4.7(a) clearly demonstrates that the improved spatial locality offered by locality optimizations reduces the miss count substantially, with more than a 35% reduction in misses in 11 out of the 21 cases (seven applications with three line sizes each). In many cases, both partial misses and full misses are reduced, and hence the total miss penalty decreases accordingly.

Figure 4.7(b) shows another useful performance metric: the total amount of bandwidth consumed by our applications. Each bar in Figure 4.7(b) denotes the total number

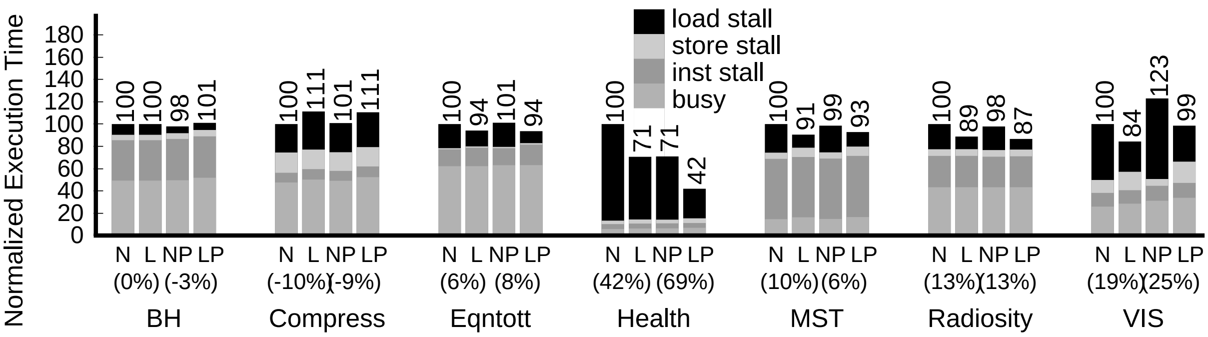


Figure 4.8: Performance impact of locality optimizations on prefetching. (**N** = not optimized, **L** = locality optimized, **NP** = prefetching without locality optimizations, **LP** = prefetching with locality optimizations).

of bytes transferred between the primary and secondary caches (the bottom section), and the amount transferred between the secondary cache and main memory (the top section). Again, each bar is normalized to the **N** case of the **32B** line size. It is clear from Figure 4.7(b) that locality optimizations reduce the bandwidth consumption in nearly all cases, and achieve a bandwidth reduction of twofold or more in a few cases. Thus we see that these optimizations deliver not only higher performance, but also reduced bandwidth consumption.

4.5.2 Impact on the Effectiveness of Prefetching

We now turn our attention to the interaction between our locality optimizations and the effectiveness of prefetching. For the three applications (`BH`, `Health`, `MST`) that appeared in our data prefetching study in Chapter 3, we used the compiler to insert *data-linearization prefetches* into the programs automatically. For the remaining four applications, we manually added software prefetches for a few static loads that suffer significantly from cache misses. Prefetches are inserted at the earliest points in the program where the prefetch addresses are known (this is done in an identical fashion for both the original and locality-optimized cases). To fully exploit the potentially improved spatial locality we employed *block prefetch* instructions, which can prefetch one or more consecutive cache lines. For both the unoptimized and optimized cases, we experimented with a range of prefetch block sizes, and we report the results with the block size that performed the best in each case.

Figure 4.8 shows how prefetching performs both with (**LP**) and without (**NP**) locality optimizations[8]. For the sake of comparison, the **N** and **L** cases from Figure 4.6 are also included in Figure 4.8. The cache line size is fixed at 32B. We observe from Figure 4.8 that the performance of prefetching is improved by locality optimizations in five applications, and two of them (`VIS` and `Health`) enjoy speedups of over 25%. We note that four of these five applications operate heavily on linked lists, and we have already learned from Chapter 3 that prefetching linked lists—especially those that are short and traversed within small loop bodies—is particularly difficult because of the pointer-chasing problem. As we can see in Figure 4.8, the *list linearization* optimization is quite successful in alleviating this problem. Moreover, in two of the five applications in which locality is substantially improved, combining locality optimizations and prefetching (**LP**) produces better results than either technique alone (most noticeably in `Health`). However, this is not the case for `MST` and `VIS` because of their considerable prefetching overhead.

Figure 4.9 shows the two additional performance metrics for better understanding our results. Figure 4.9(a) shows a new category of misses (*late prefetched misses*) not seen in Figure 4.7(a). This category accounts for those load references that were prefetched but still missed because their prefetches were launched too late (so these references were combined with their prefetches which were still outstanding). The large reduction in the miss count of the **LP** cases over **NP** cases in five applications clearly demonstrates that our locality optimizations remarkably improve the effectiveness of prefetching. For `BH`, despite the **LP** case having similar load D-cache misses to the **NP** case, **LP** actually performs worse than **NP** due to the overhead of data relocation in the **LP** case[9]. We also observe from Figure 4.9(a) that `VIS` suffers from the cache pollution caused by prefetching in the **NP** case, as its miss count is increased by 20% over the original. Fortunately, our locality optimization is able to eliminate this cache pollution effect and actually helps prefetching reduce the miss count by half that of the original case. Figure 4.9(b) shows the amount of bandwidth consumed by prefetching. Unlike the cases without prefetching, **LP** does not necessarily consume less bandwidth than **NP**, since the prefetch block size may be larger in the **LP** cases.

4.5.3 Case Studies

Having studied the overall performance, we now look at the individual applications in more detail.

---

[8]For `BH`, `Health`, and `MST`, the **NP** cases in Figure 4.8 and the **D** cases in Figure 3.31 on page 100 essentially have the same prefetch codes. The slight performance differences are mainly due to the use of block prefetches in the **NP** cases.

[9]Though our optimization for `BH` is not active in the **L** case for 32B lines, it is turned on in the **LP** case since prefetching can exploit the potential spatial locality created by our optimization.

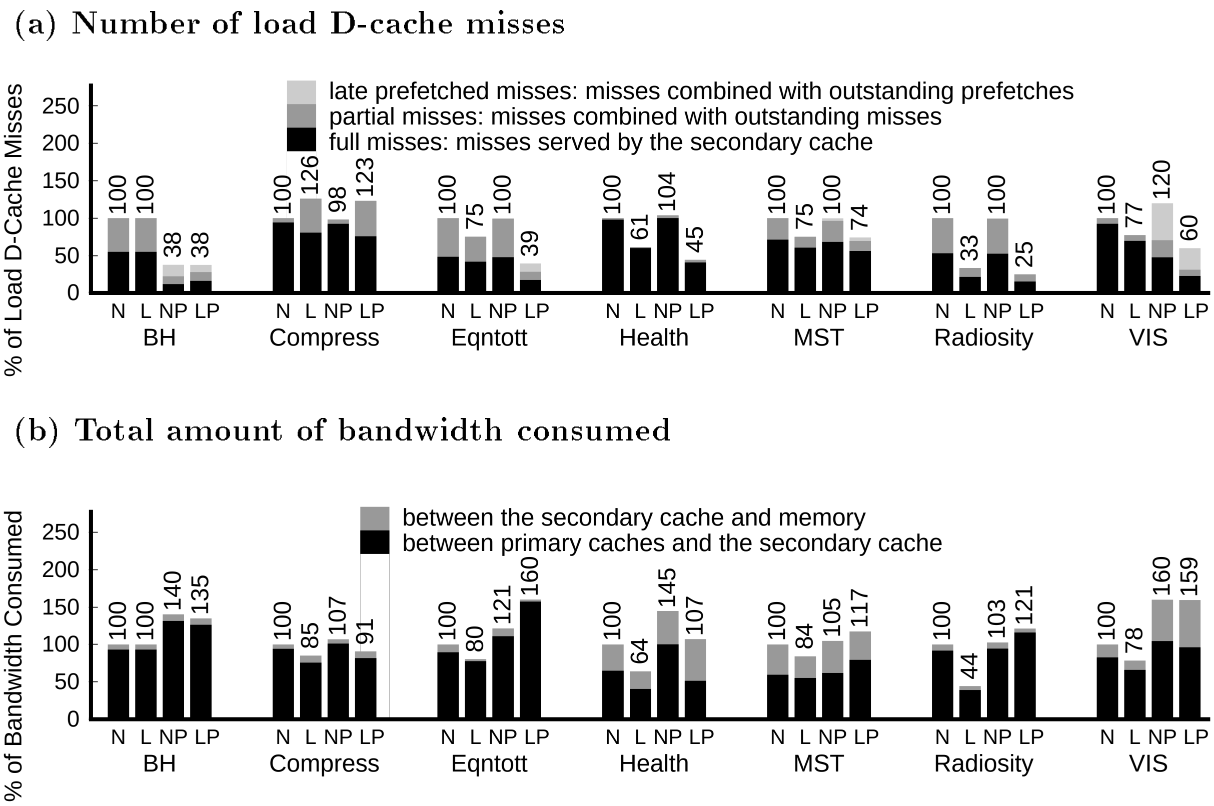


Figure 4.9: Additional performance metrics for the impact of locality optimizations on prefetching (**N** = not optimized, **L** = locality optimized, **NP** = prefetching without locality optimizations, **LP** = prefetching with locality optimizations). The line size is fixed at 32B. The y-axes are normalized to the **N** cases.

Health, MST, Radiosity, and VIS

We apply the same locality optimization to all four of these applications: *list linearization*. The structure of the linked lists used in these applications is modified throughout the program execution, and therefore list linearization is invoked periodically. To make our discussion more concrete, we use `VIS` as a representative example. `VIS` is a large application, consisting of more than 150,000 lines of C code. This program makes extensive use of a generic list library which implements many common list operations. Our optimizations are localized within this library. We optimize the locality of list processing as follows. We add a counter field to the head record of each list to count how many insertion or deletion operations have been performed on the list since the last time that the list was linearized. The list linearization procedure `ListLinearize()`—shown earlier in Figure 4.4(b)—is invoked whenever the list's counter exceeds a threshold, which was arbitrarily set to 50 in our experiments. The counter is reset after each linearization. De-

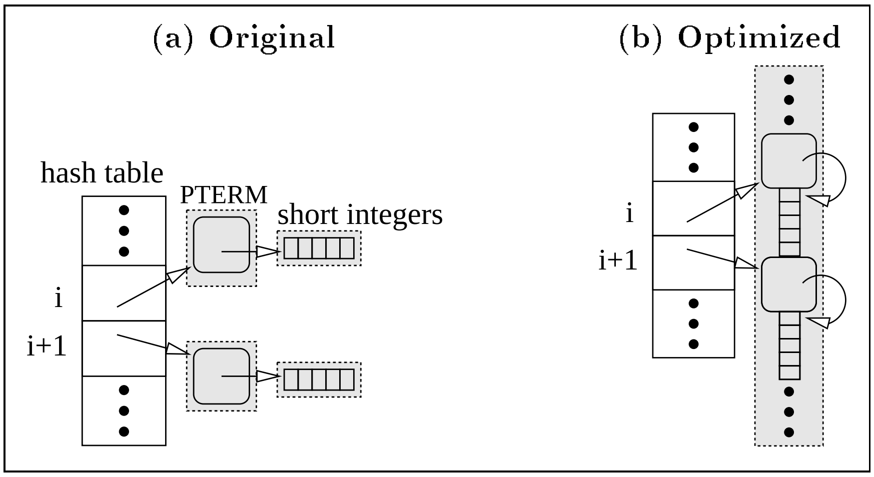


Figure 4.10: Locality optimization for `Eqntott` (objects in the same shaded region are allocated to contiguous memory).

spite the simplicity and usefulness of this optimization, performing it without the support of memory forwarding is dangerous due to the fact that most functions in this library return pointers to list elements, which can be scattered across any of the over hundred source files of `VIS`. The program behave incorrectly if after a list is linearized, it is later accessed using a pointer to the middle of the list that existed before the linearization. Fortunately, memory forwarding allows us to simply ignore this hazard, thereby safely resulting in an over twofold performance gain with 128B lines.

Eqntott

The most interesting data structure in `Eqntott` is a hash table which stores pointers to a record of type `PTERM`. A `PTERM` record in turn contains a pointer to an array of short integers. The original layout of this data structure is shown in Figure 4.10(a). We optimize the locality by (i) relocating a `PTERM` record and its short integer array into a single chunk of memory, and (ii) putting these chunks into contiguous memory locations in increasing order of the hash index. The optimized layout is shown in Figure 4.10(b). This relocation optimization is invoked only once in the program, immediately after the hash table is constructed. Therefore, this optimization costs only 0.1% more instructions than the original program.

BH

In `BH`, an octree is constructed and then traversed at each time step of the N-body force calculation. The octree is constructed in a depth-first order, but the traversal order is fairly random. We improve the locality of the traversal by clustering non-leaf nodes

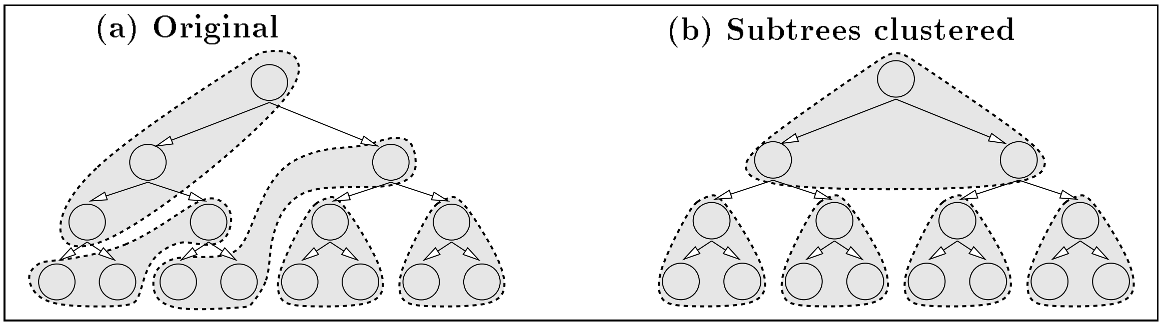


Figure 4.11: Example of the subtree clustering applied to BH (nodes in the same shaded region are in the same cache line).

of the tree. We do not cluster leaf nodes since they are actually linked together by a list and accessed via list traversals. Subtree clustering [31] attempts to pack nodes of a subtree into a cache line, in the most balanced form. Locality will be improved if the next node to be visited—which can be any of the children of the current node—is already in the current cache line. Figure 4.11 illustrates this optimization using a binary tree. Figure 4.11(a) shows the original memory layout of the tree, which was created using a pre-order traversal, and Figure 4.11(b) shows the memory layout after subtree clustering. Since a non-leaf node in `BH` is 78B long, we need cache lines of 256B or longer to do meaningful clustering.

Compress

The most relevant data structures in `Compress` are two hash tables, namely `htab` and `codetab`, which are implemented using arrays. Indices to `htab` are computed through hashing, but `codetab` always shares the same index values as `htab`. Therefore, spatial locality might be improved if `codetab[i]` could be next to `htab[i]` in the memory. We achieve this by copying the two tables into a single larger table `T` such that `htab[i]` and `codetab[i]` occupy adjacent elements in `T`. We also replace all explicit array references to `htab` and `codetab` by the appropriate array references to `T`. Forwarding addresses and bits are also set appropriately in `htab` and `codetab` to catch any unexpected references to them. However, as we have already seen in Figure 4.6, performance is in fact degraded by this optimization because more misses occur in the optimized code. This suggests that the original memory layout where `htab[i]` and `htab[i+1]` are adjacent (so are `codetab[i]` and `codetab[i+1]`) enjoys more locality than the optimized one.

4.5.4 Impact of Forwarding Overhead

In each of the applications that we have studied so far, we were successful enough at updating the appropriate pointers to point to a relocated object's new location that the forwarding mechanism was almost never invoked. (At the same time, we would like to point out that without memory forwarding support, we would not have been able to apply these optimizations because they were not provably safe.) As a result, the performance of dereferencing a forwarding address did not matter in these cases. To quantify the impact of forwarding overhead in a case where it does matter, we now focus on `SMV`, which is the only application we studied that experiences significant forwarding after data relocation.

`SMV` is a model checking program which makes extensive use of Binary Decision Diagrams (BDDs) [16]. The BDD nodes are connected both through a hash table and through binary trees. The hash table is organized as an array of buckets pointing to linked lists. Since more cache misses occur during hash table accesses than binary tree accesses, we attempted to improve locality by linearizing the lists stored in the hash table. Unfortunately, since our optimized code is not able to update the tree pointers to point to a relocated object's new address, forwarding does occur whenever relocated BDD nodes are accessed via the tree pointers.

Figure 4.12 shows our performance results for `SMV`. In addition to the cases without (**N**) and with (**L**) locality optimization, as shown in earlier graphs, we also show a case with locality optimization and *perfect* forwarding (**Perf**). We say that memory forwarding is perfect if all references to relocated objects access them directly at their new addresses, and hence no forwarding is actually required. While this latter case is not achievable, it represents a useful bound on performance. As we see in Figure 4.12(a), the performance of scheme **L** is degraded by forwarding in two ways. First, the act of dereferencing forwarding addresses incurs extra time. Second, when forwarding occurs, both the old and new locations of relocated data are accessed, thereby degrading cache behavior. With perfect forwarding, there is no forwarding overhead and the performance does improve. However, the improvement is only marginal due to the fact that we cannot optimize the layout to accelerate both the hash table and tree access patterns.

To provide further insight into the source of the forwarding overhead, Figure 4.12 presents three additional performance metrics. Figure 4.12(b) shows the impact of the schemes on the number of load and store data cache misses. As we see in this figure, scheme **L** suffers a noticeable increase in misses. Figure 4.12(c) shows that 7.9% of loads and 1.7% of stores require one forwarding hop under scheme **L**. Finally, Figure 4.12(d) shows the average number of CPU cycles needed to complete a load or store under

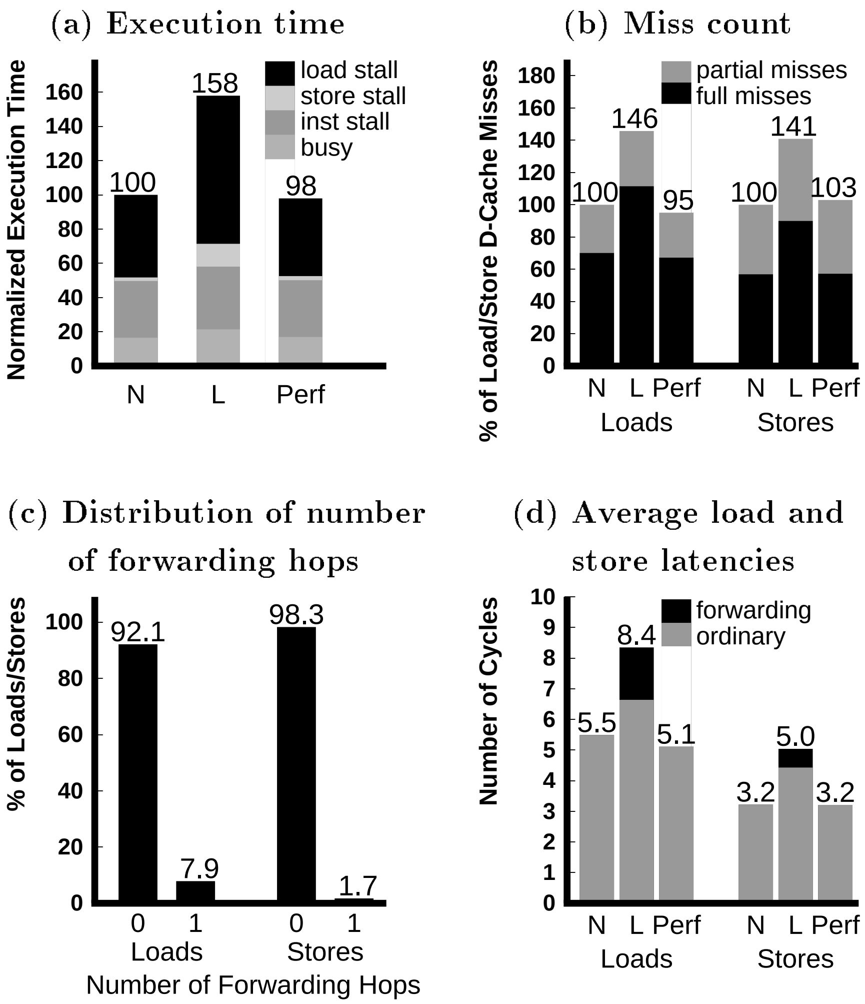


Figure 4.12: Performance results for `SMV`. (**N** = not optimized, **L** = locality optimized with realistic forwarding, **Perf** = locality optimized with perfect forwarding). The line size is fixed at 64B.

each scheme. Each bar in Figure 4.12(d) is divided into two sections explaining the reason for the stall. The *forwarding* section represents the time spent dereferencing forwarding addresses, and the *ordinary* section includes cache hit and miss latencies. The *ordinary* sections of scheme **L** increase due to the cache pollution effects of touching the forwarding pointers, as mentioned earlier. As we see in Figure 4.12(d), both the latency of dereferencing a forwarding address and its resulting cache pollution effects play significant roles in the overall performance degradation. A profiling tool based on user-level traps (as discussed earlier in Section 4.3.2) could potentially identify cases such as this where forwarding occurs too frequently.

4.6 Chapter Summary

As changes in technology continue to alter the landscape of what constitutes a major performance bottleneck, it is sometimes worth re-examining old architectural ideas that have fallen out of fashion to see whether they can be adapted to serve completely new purposes. In this chapter, we have examined such a technique: *memory forwarding*. Although the original concept was proposed to facilitate garbage collection in early Lisp machines, we have demonstrated that memory forwarding can be adapted to address the entirely modern problem of enhancing cache performance. In addition, we have shown that it is quite feasible to implement this mechanism within modern out-of-order superscalar processors, largely because forwarding can be treated as an exception.

To fully understand the potential of memory forwarding, we have modeled it in detail in a simulator of a modern processor, and used it to enable a number of aggressive locality optimizations on a set of non-numeric applications. Our experimental results provide us the following lessons:

- The locality optimizations that we applied require only small changes to the original program, but their safety is difficult to guarantee. Several of these optimization—in particular list linearization—can easily be implemented as library functions, provided that their safety can be guaranteed. Memory forwarding allows users of these optimizations to ignore correctness issues and focus performance instead.

- Our optimizations are useful not only for hiding memory latency, but also for reducing memory bandwidth consumption. The result is the significant performance gains across our applications: as much as over twofold speedups in two applications.

- Forwarding does not occur in seven of our eight applications. However, in the only one application where forwarding actually happens, the forwarding overhead is significant. Therefore, software should not apply memory forwarding blindly in situations where a large number of forwarding address dereferences are expected.

In summary, by liberating aggressive relocation-based data layout optimizations from concerns over violating program correctness, memory forwarding can enable impressive performance gains. Although one must still exercise caution not to use forwarding carelessly, a user-level trap mechanism can help identify and avoid cases where pointers have not been updated successfully. We demonstrate that memory forwarding is a powerful tool which makes a large class of optimizations that were promising in theory useful in practice.

Chapter 5

Correlation-Based Cache Miss Prediction

5.1 Introduction

In this chapter, we address the problem of predicting data cache misses in non-numeric applications. Knowing accurately in advance when cache misses will happen can improve the overall performance of latency tolerance techniques by reducing unnecessary overhead. While the benefit of a technique typically outweighs its overhead whenever a miss is tolerated, the overhead hurts performance in cases where the reference would have enjoyed a cache hit anyway. Therefore to maximize overall performance, we would like to apply a latency tolerance technique *only* to the precise set of dynamic references that would suffer misses.

5.1.1 Importance of Cache Miss Prediction

To get a clearer picture of the importance of cache miss prediction, we discuss in this section how it can be applied to improve the effectiveness of three major latency tolerance techniques: instruction scheduling, software-controlled prefetching, and multithreading.

Instruction Scheduling

The latency of a load L is tolerated in this technique by scheduling the "right" number of instructions that are not dependent on L in between L and its first use. Instruction scheduling can be done statically (by the compiler) or dynamically (by hardware). To maximize scheduling effectiveness, it is essential for the compiler or hardware to know the load latency prior to the actual scheduling. In static scheduling, scheduling a load using the *hit* latency will cause memory stall if the load is in fact a cache *miss*. In contrast, scheduling a load using the *miss* latency increases register lifetimes, which can lead to spilling. Moreover, in some scheduling algorithms such as trace scheduling [78] and hyperblock scheduling [84] where control speculation is allowed, loads scheduled using long latencies are likely to become speculative. If any of these loads are misspeculated, the total instruction count will be increased. In dynamic scheduling, cache miss prediction is particularly helpful for high-clock-rate, out-of-order machines such as the Alpha 21264 [66] where instruction scheduling itself is a multiple-cycle operation. In such machines, in order to minimize the load-and-use latency, the decision of when to schedule an instruction that consumes the result of a load L has to be made *before* actually knowing whether L is a cache hit. In other words, hardware has to make the scheduling decision based on cache miss prediction. If L is a *miss* but was optimistically predicted as a *hit*, all the instructions depending on L that were scheduled too early need to be squashed and then re-executed. On the other hand, if L is a *hit* but was pessimistically predicted as a *miss*, all the dependent instructions of L will be unnecessarily delayed. In summary, accurate prediction of data cache misses is important to both static and dynamic instruction scheduling.

Software-Controlled Prefetching

Software-controlled prefetching tolerates latency by bringing data lines into the cache before they are needed [20, 79, 96]. Compared with hardware-controlled prefetching [28, 60], software-controlled prefetching is better able to exploit application-specific knowledge about future data access patterns, but requires additional instructions to compute prefetch addresses and launch the prefetches themselves. If the data access patterns are irregular (and hence more computational effort may be needed to predict prefetch addresses), this instruction overhead can be significant even with modern superscalar processors. One such example is the *history-pointer prefetching* scheme we proposed in Chapter 3. Recall that this scheme adds extra pointers called *history pointers* to record data addresses seen in the past and uses them as prediction of future prefetch addresses. As we have already seen in Section 3.5.3, the instruction overhead of manipulating history pointers is substantial even on a four-issue superscalar machine—it can be as much as the number of instructions executed in the case without prefetching. Therefore, it is very helpful to have accurate miss prediction so that these relatively expensive prefetching schemes would not be applied to cases that already result in cache hits.

Multithreading

Multithreading [4, 117, 130] tolerates latency by overlapping a long-latency access from one thread with the computation from other parallel threads. Traditionally, the two main mechanisms involved in multithreading—deciding when to switch between threads and the actual thread switching itself—are entirely controlled by *hardware* [4, 7, 73, 117, 130]. Recently a scheme that controls these two mechanisms using *software*, with the help of informing memory operations [53], has been proposed [99]. A key concern of multithreading techniques is to minimize run-time thread switching penalty, which is classified into *instruction overheads* and *memory overheads*. Instruction overheads can be either *explicit* or *implicit*. Instruction overhead exists *explicitly* in software-controlled multithreading where thread switching is completely performed by software. In hardware-controlled multithreading, instruction overhead is *implicit*: all partially-executed instructions belonging to the current thread have to be squashed from the CPU pipeline upon a thread switch. The memory overheads arise from the fact that data and instruction caches/TLBs are stressed more by a multiple-threaded workload than a single-threaded workload, due to a decrease in locality [3, 70, 130]. To minimize the impact of instruction and memory overheads, thread switching should take place only at long-latency misses, which are nowadays typically secondary or tertiary cache misses. This kind of "selective" thread switching requires some means to accurately predict whether a primary cache miss is also going to be a miss in the next hierarchy level.

Remarks

We would like to emphasize that the overheads of latency tolerance techniques will continue to be a concern in spite of the ever-increasing speed discrepancy between processors and memory. There are two reasons for this. First, memory latency can be hidden only *partially* most of time due to the lack of parallelism in reality. Consequently, the effective ratio of overhead to latency is usually higher than the theoretical ratio. Second, we expect that future innovations in latency tolerance will involve larger overheads than what we experience nowadays due to the larger potential benefits when cache misses occur. Therefore, techniques to predict data cache misses will remain important.

5.1.2 Predicting Data Cache Misses in Non-Numeric Codes

While previous work has addressed the problem of predicting data cache misses for numeric codes [96], this study focuses on the more difficult but important case of isolating dynamic miss instances in *non-numeric* applications. To overcome the compiler's inability to analyze data locality in non-numeric codes, we can instead make use of profiling information. One simple type of profiling information is the precise miss ratios of all static memory references. Throughout the remainder of this chapter, we will refer to

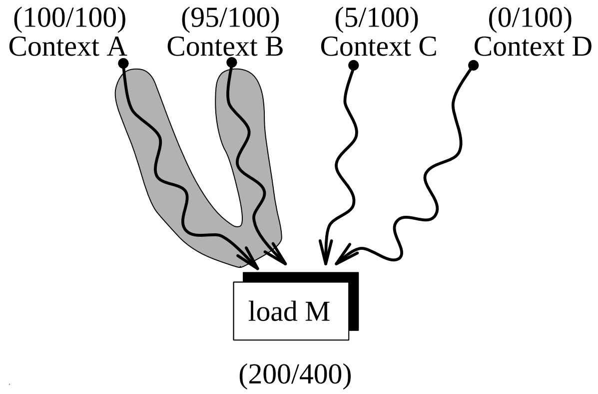


Figure 5.1: Example of how correlating cache misses with the dynamic context may improve predictability (X/Y means X misses out of Y dynamic references).

this approach as *summary profiling*, since the miss ratio of each memory reference is summarized as a single value.

If summary profiling indicates that all significant memory reference instructions (i.e. those which are executed frequently enough to make a non-trivial contribution to execution time) have miss ratios close to 0% or 100%, then isolating dynamic misses is trivial—we simply apply the latency tolerance technique only to the static references which always suffer misses. In contrast, if the important references have *intermediate* miss ratios (e.g., 50%), then we do not have sufficient information to distinguish which dynamic instances hit or miss, since this information is lost in the course of summarizing the miss ratio. The current state-of-the-art approach for dealing with intermediate miss ratios is to treat all static memory references with miss ratios above or below a certain threshold as though they always miss or always hit, respectively [2]. However, this all-or-nothing strategy will fail to hide latency when references are predicted to hit but actually miss, and will induce unnecessary overhead when references are predicted to miss but actually hit. Rather than settling for this sub-optimal performance, we would prefer to predict dynamic hits and misses more accurately.

Correlation Profiling

As profiling is becoming increasingly helpful for improving system performance, profiling techniques that are more informative than simple performance counters have been proposed recently. Such techniques include *informing memory operations* [53] and *ProfileMe* [39], which enable new classes of lightweight profiling tools that can collect more sophisticated information than simply the per-reference miss ratios. For example, cache misses can be correlated with information such as recent control-flow paths, whether recent memory references hit or missed in the cache, etc., to help predict dynamic cache miss behavior. We will refer to this approach as *correlation profiling*.

Figure 5.1 illustrates how correlation profiling information might be exploited. The load instruction shown in Figure 5.1 has an overall miss ratio of 50%. However, depending on the dynamic context of the load, we may see more predictable behavior. In this example, contexts **A** and **B** result in a high likelihood of the load missing, whereas contexts **C** and **D** do not. Hence we would like to apply a latency tolerance technique within contexts **A** and **B** but not **C** or **D**.

The dynamic contexts shown in Figure 5.1 should be viewed simply as non-overlapping sets of dynamic instances of the load which can be grouped together because they share a common distinguishable pattern. In this study, we consider three different types of information which can be used to distinguish these contexts. The first is *control-flow* information—i.e. the sequence of N basic block numbers preceding the load. The other two are based on sequences of cache access outcomes (i.e. hit or miss) for previous memory references: *self* correlation considers the cache outcomes of the previous N dynamic instances of the given static reference, and *global* correlation refers to the previous N dynamic references across the entire program.

5.1.3 Related Work

Abraham *et al.* [2] investigated using summary profiling to associate a single latency tolerance strategy (i.e. either attempt to tolerate the latency or not) with each profiled load. They used this approach to reduce the cache miss ratios of nine SPEC89 benchmarks, including both integer and floating-point programs. In a follow-up study [1], they also report the improvement in *effective cache miss ratio*. In contrast with this earlier work, our study focused on *correlation profiling*, which is a novel technique that provides superior prediction accuracy relative to summary profiling, as evidenced by our experimental results that will be shown in Section 5.4.

The three forms of correlation explored in this study were inspired by earlier work on using branch histories to enhance *branch prediction* accuracies [24, 88, 103, 140, 141]. History-based branch prediction schemes associate multiple predictors with each conditional branch (or set of conditional branches) in a program. At a branch prediction, one of the predictors associated with the branch encountered is selected via a branch history, which can be *local* or *global*. A local branch history [140] is a sequence of previous outcomes of a particular conditional branch. A global branch history [103, 140] is a sequence of previous outcomes of all conditional branches encountered. There are *static* and *dynamic* history-based branch prediction schemes. Static schemes [141] use code duplication and profile-based branch prediction information to transform a program to improve its branch prediction accuracy. On the other hand, dynamic schemes [88, 103, 140] implement the branch histories, predictors and indexing mechanisms all in hardware. While our self and global correlation are analogous to local and global history-based branch prediction, our exploitation of control-flow information in cache miss prediction does not have a counterpart in branch prediction.

Ammons *et al.*[8] studied the impact of flow and context sensitivity on a number of performance metrics that can be collected via hardware performance counters. One of their findings that is particularly related to our work is that a large fraction of primary data cache misses (for a 16KB direct mapped D-cache) in the SPEC95 benchmarks occur along a relatively small number of frequently executed *intraprocedural* paths. They also observed that D-cache misses are heavily concentrated in a small number of procedures. One of our correlation profiling techniques—control-flow correlation—also relates D-cache misses to paths but is more general because the paths we profiled can be *interprocedural*. In addition, our study also measured the sensitivity of correlation profiling to different input sets (see Section 5.4). Another unique feature of our study is the providence of detailed case studies (see Section 5.5) that map observed correlation behaviors back to the source codes.

5.1.4 Objectives and Overview

The primary goals of this study are twofold. First, we would like to quantify the *improvement in data cache miss prediction accuracy* offered by correlation profiling relative to summary profiling. Note that unlike branch prediction, where prediction accuracy can be summarized as a single value, we separate *false positives* from *false negatives* since the overheads of misprediction are asymmetrical (i.e. failing to hide the latency of a miss is generally more expensive than paying latency tolerance overhead for a reference that hits). Second, and more importantly, we would like to understand *why* correlation profiling works by relating its success to underlying program behaviors. Hence our analysis includes detailed case studies for many of the applications.

Although the focus of this study is on *understanding* correlation profiling rather than building tools to exploit it, we do discuss how correlation profiling can be used in practice, and we demonstrate that it can actually improve the performance of dynamic instruction scheduling and software-controlled prefetching. We focus specifically on predicting *load* misses in this study because load latency is fundamentally more difficult to tolerate than store latency (the latter can generally be hidden through buffering and pipelining). Although we rely on simulation to capture our profiling information in this study, correlation profiling is a practical technique because it could be performed with relatively little overhead using mechanisms such as *informing memory operations* [53] or *ProfileMe* [39].

The remainder of this chapter is organized as follows. We begin in Section 5.2 by discussing the three different types of history information that we use for correlation profiling, and in Section 5.3 we present a qualitative analysis of the expected performance benefits. In Section 5.4, we present our experimental results which quantify the performance advantages of correlation profiling in a collection of 21 non-numeric applications. In addition, in Section 5.5, we report the memory-access behaviors of individual applications which explain when and how correlation profiling is effective. We then demonstrate the practicality of correlation profiling in Sections 5.6.1 and 5.6.2. In Section 5.6.1, we apply correlation-based prediction to dynamic instruction scheduling. We discuss in Section 5.6.2 how one could use code duplication techniques to exploit correlation profiling statically, using software-controlled prefetching as an example application. Finally, we present conclusions in Section 5.7.

5.2 Correlation Profiling Techniques

In this section, we propose and motivate three new correlation profiling techniques for predicting cache outcomes: *control-flow correlation*, *self correlation*, and *global correlation*.

5.2.1 Control-Flow Correlation

Our first profiling technique correlates cache outcomes with the recent control-flow paths. To collect this information, the profiling tool maintains the N most recent basic block numbers in a FIFO buffer, and matches this pattern against the hit/miss outcomes for a given memory reference. Intuitively, control-flow correlation is useful for detecting cases where either *data reuse* or *cache displacement* is likely.

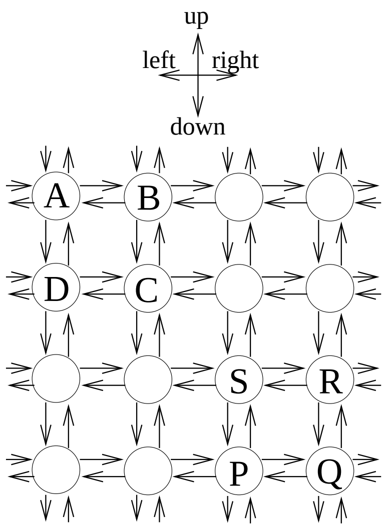
If we are on a path which leads to *data reuse*—either temporal or spatial—then the next reference is likely to be a cache hit. Consider the example shown in Figure 5.2(a)-(b), where a graph is traversed by the recursive procedure `walk()`. Any cyclic paths (e.g., A→B→C→D→A or P→Q→R→S→P) will result in temporal reuse of `p→data`. In this example, control-flow correlation can potentially detect that if the last four traversal decisions lead to a cycle (e.g., *right*, *down*, *left*, and *up*), then there is a high probability that the next `p→data` reference will enjoy a cache hit.

Some control-flow paths may increase the likelihood of a cache miss by displacing a data line before it is reused. For example, if the "`x > 0`" condition is true in Figure 5.2(c),

(a) Code with data reuse

```
struct node {
  int data;
  struct node *left,
    *right, *up, *down;
};
void walk(node* p) {
  work(p→data);
  if (go_left(p→data))
    p = p→left;
  elsif (go_right(p→data))
    p = p→right;
  elsif (go_up(p→data))
    p = p→up;
  elsif (go_down(p→data))
    p = p→down;
}
```

(b) Example graph



(c) Code with cache displacement

```
x = *p;
if  (x > 0) {
  for  (i = 0;
    i < 100000; i++)
    a[i] = foo(i);
}
y = *p;
```

Figure 5.2: Examples of how control-flow correlation can detect data reuse and cache displacement (control-flow profiled loads are underlined).

then the subsequent `for` loop is likely to displace `*p` from the primary cache before it can be loaded again. Note that whereas paths which access large amounts of data are obvious problems, the displacement might also be due to a mapping conflict.

5.2.2 Self Correlation

Under *self correlation*, we profile a load L by correlating its cache outcome with the N previous cache outcomes of L itself. This approach is particularly useful for detecting forms of spatial locality which are not apparent at compile time. For example, consider the case in Figure 5.3 where a tree is constructed in preorder, assuming that consecutive calls to the memory allocator return contiguous memory locations, and that a cache line is large enough to hold exactly two `treeNodes`. Depending on the traversal order (and the extent to which the tree is modified after it is created), we may experience spatial locality when the tree is subsequently traversed. For example, if the tree is also traversed in preorder, we will expect `p→data` to suffer misses on every-other reference as cache line boundaries are crossed. Therefore despite the fact that the overall miss ratio of `p→data` is 50% and the compiler would have difficulty recognizing this as a form of spatial locality, self correlation profiling would accurately predict the dynamic cache outcomes for `p→data`.

(a) Example Code

```
void preorder(treeNode* p) {
  if (p != NULL) {
    work(p→data);
    preorder(p→left);
    preorder(p→right);
  }
}
```

(b) Tree constructed and traversed both in preorder

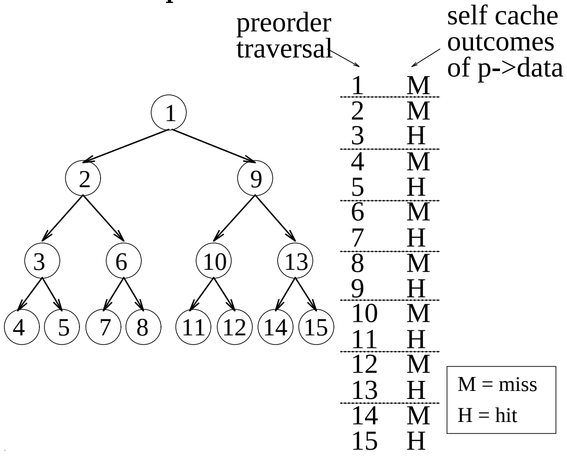


Figure 5.3: Example of using self-correlation profiling to detect spatial locality for `p→data` (consecutively numbered nodes are adjacent in memory).

5.2.3 Global Correlation

In contrast with self correlation, the idea behind *global correlation* is to correlate the cache outcome of a load L with the previous N cache outcomes regardless of their positions within the program. The profiling tool maintains this pattern using a single N-deep FIFO which is updated whenever dynamic cache accesses occur. Note that because earlier instances of L itself may appear in this global history pattern, global correlation may capture some of the same behavior as self correlation (particularly in extremely tight loops).

Intuitively, global correlation is particularly helpful for detecting *bursty* patterns of misses across multiple references. One example of this situation is when we move to a new portion of a data structure that has not been accessed in a long time (and hence has been displaced from the cache), in which case the fact that the first access to an object suffers a miss is a good indication that associated references to neighboring objects will also miss. Figure 5.4 illustrates such a case where a large hash table (too large to fit in the cache) is organized as an array of linked lists. In this case, we might expect a strong correlation between whether `htab[i]` (the list head pointer) misses and whether subsequent accesses to `curr→data` (the list elements) also miss. Similarly, if the same entry is accessed twice within a short interval (e.g., `htab[10]`), the fact that the head pointer hits is a strong indicator that the list elements (e.g., `A→data` and `B→data`) will also hit.

In summary, by correlating cache outcomes with the context in which the reference

(a) Example code

```
while (1) {
  register int i = hash(get());
  register listNode* curr = htab[i];
  while (curr != NULL) {
    work(curr→data);
    curr = curr→next;
  }
}
```

(b) Hash table accesses

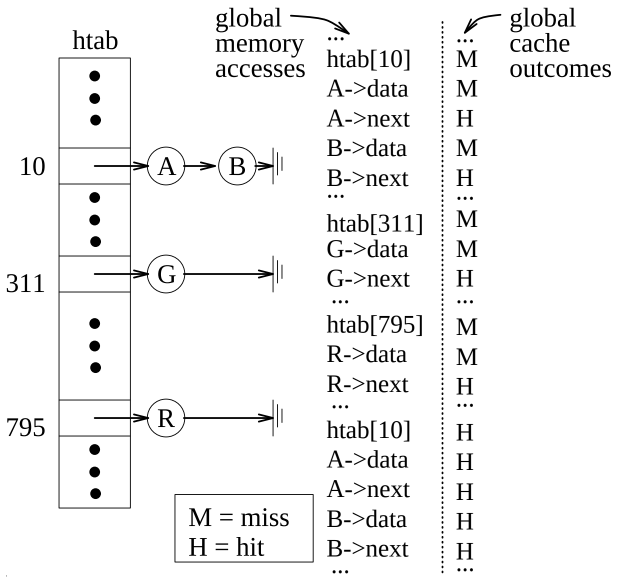


Figure 5.4: Example of using global-correlation profiling to detect bursty cache misses for `curr→data`.

occurs—e.g., the surrounding control flow or the cache outcomes of prior references—we can potentially predict the dynamic caching behavior more accurately than what is possible with summarized miss ratios.

5.3 Qualitative Analysis of Expected Benefits

Before presenting our quantitative results in later sections, we begin in this section by providing some intuition on how correlation profiling can improve performance. A key factor which dictates the potential performance gain is the ratio of the latency tolerance overhead (V) to the cache miss latency (L). In the extreme cases where $\frac{V}{L} = 0$ or $\frac{V}{L} = 1$, there is no point in applying the latency tolerance technique (T) selectively, since it either has no cost or no benefit. When $0 < \frac{V}{L} < 1$, however, applying T selectively may be important.

Figure 5.5(a) illustrates how the average number of effective *stall cycles per load* (CPL) varies as a function of $\frac{V}{L}$ for various strategies for applying T. (Note that our CPL metric includes any overhead associated with applying T, but does not include the cycles for executing the load instruction itself.) If T is never applied, then the CPL is simply mL, where m is the average miss ratio. At the other extreme, if we *always* apply T, then the latency will always be hidden, but *all* references (even those that normally hit) will suffer the overhead V: hence the $CPL = V$. Note that when $\frac{V}{L} > m$, it is better

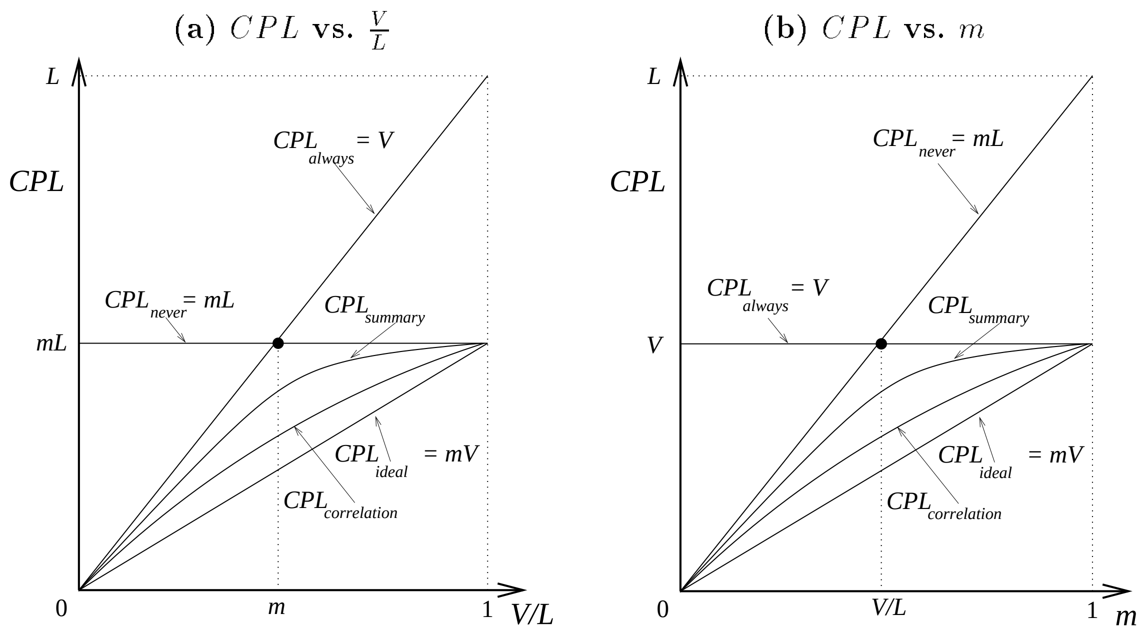


Figure 5.5: Illustration of the CPL for different approaches of applying a latency tolerance scheme (m = overall average load miss ratio, V = latency tolerance overhead, and L = load miss latency). The $CPL_{summary}$ and $CPL_{correlation}$ curves are chosen arbitrary for the sake of illustration.

to never apply T rather than always applying it. Figure 5.5(b) shows an alternative view of CPL, where it is plotted as a function of m for a fixed $\frac{V}{L}$. Again, we observe that the choice of whether to always or never apply T depends on the value of m relative to $\frac{V}{L}$.

To achieve better performance than this all-or-nothing approach, we apply the same decision-making process (i.e. comparing the miss ratio with $\frac{V}{L}$) to more refined sets of loads. In the ideal case, we would consider and optimize each dynamic reference individually (the resulting CPL of mV is shown in Figure 5.5). However, since this is impractical for software-based techniques, we must consider aggregate collections of references. Since summary profiling provides only a single miss ratio per static reference, the finest granularity at which we can decide whether or not to apply T is once for all dynamic instances of a given static reference. Figure 5.5 illustrates the potential shape of the summary profiling curve, which is bounded by the cases where T is never, always, and ideally applied. Since correlation profiling distinguishes different sets of dynamic instances of a static load based on context information, it allows us to make decisions at a finer granularity than with summary profiling. Therefore we can potentially achieve even better performance, as illustrated in Figure 5.5.

Below we formulate the five CPL's shown in Figure 5.5. Denote the CPL under a particular approach S by CPL_S. Let CPL_S^i be the CPL_S of load i in the program and f_i be the fraction of references made by load i out of the total references of all loads. Let m_i be the miss ratio of load i and m be the overall miss ratio of all loads. Then:

$$CPL_S = \sum_i CPL_S^i \times f_i \tag{5.1}$$

<u>CPL_{never}:</u> A load reference is stalled only when it is a cache miss, so:

$$CPL_{never} = mL \tag{5.2}$$

<u>CPL_{always}:</u> T fully tolerates the latencies of all load references but always incurs the overhead, so:

$$CPL_{always} = V \tag{5.3}$$

<u>$CPL_{summary}$:</u> The miss ratio m_i decides whether T should be applied to load i:

$$CPL_{summary}^i = \begin{cases} m_i L & \text{if } m_i \leq \frac{V}{L} \text{ (i.e. not apply } T) \\ V & \text{otherwise (i.e. apply } T) \end{cases} \tag{5.4}$$

$$\begin{aligned} CPL_{summary} &= \sum_{i \in A} CPL_{summary}^i \times f_i + \sum_{i \in NA} CPL_{summary}^i \times f_i \\ &= V \sum_{i \in A} f_i + L \sum_{i \in NA} m_i f_i \qquad \text{by (5.4)} \end{aligned} \tag{5.5}$$

where A is the set of loads with miss ratios $> \frac{V}{L}$ and NA is the set of loads with miss ratios $\leq \frac{V}{L}$.

<u>$CPL_{correlation}$:</u> T is only applied to references of load i that belong to contexts with miss ratios $> \frac{V}{L}$. The formula for $CPL_{correlation}^i$ can be simply obtained by adding an extra level to Equation (5.5) to capture the notion of contexts within load i. That is:

$$CPL_{correlation}^i = V \sum_{j \in A_i} f_{i,j} + L \sum_{j \in NA_i} m_{i,j} f_{i,j} \tag{5.6}$$

where A_i is the set of contexts of load i with miss ratios $> \frac{V}{L}$, NA_i is the set of contexts of load i of miss ratios $\leq \frac{V}{L}$, $m_{i,j}$ is the miss ratio of context j of load i, and $f_{i,j}$ is the fraction of references of load i that are on context j. $CPL_{correlation}$ can be obtained by substituting $CPL_{correlation}^i$ into Equation (5.1).

<u>CPL_{ideal}:</u> Under this ideal scheme, load miss latencies are fully tolerated and the overhead is only incurred to miss references:

$$CPL_{ideal} = mV \tag{5.7}$$

5.4 Quantitative Evaluation of Performance Gains

In this section, we present experimental results to quantify the performance benefits offered by correlation profiling. We begin by measuring and understanding the potential performance advantages for a generic latency tolerance scheme. Later, we will focus on dynamic instruction scheduling and software-controlled prefetching as specific case studies in Sections 5.6.1 and 5.6.2, respectively.

5.4.1 Experimental Methodology

We measured the impact of correlation profiling on the following 21 non-numeric applications: the entire SPEC95 integer benchmark suite, the additional integer benchmarks contained in the SPEC92 suite, the uniprocessor version of a graphics application from SPLASH-2 [135], eight applications from Olden [109] (the suite of pointer-intensive benchmarks we used in Chapter 3), and the standard UNIX utility `awk`. Table 5.1 briefly summarizes these applications, including in each case the *testing* input set and an alternate *training* input set that differs from the testing one. All input data sets were run to completion.

We compiled each application with `-O2` optimization using the standard MIPS C compilers under IRIX 5.3. We used the MIPS `pixie` utility [119] to instrument these binaries, and piped the resulting trace into our detailed performance simulator. To increase simulation speed and to simplify our analysis, we model a perfectly-pipelined single-issue processor (similar to the MIPS R2000) in this section. (Later, in Sections 5.6.1 and 5.6.2, we model modern superscalar processors.)

To reduce the simulation time, our simulator performs correlation profiling only on a selected subset of load instructions. Our criteria for profiling a load is that it must rank among the top 15 loads in terms of total cache miss count, and its miss ratio must be between 10% and 90%. Using this criteria, we focus only on the most significant loads which have intermediate miss ratios. We will refer to these loads as the *correlation-profiled loads* (the fraction of total dynamic load references in each application that are correlation profiled is shown under "CP Loads" in Table 5.2).

We attempt to maintain as much history information as possible for the sake of correlation. For control-flow correlation, we typically maintained a path length of 200 basic blocks—in some cases this resulted in such a large number of distinct paths that we were forced to measure only 50 basic blocks. For the self and global correlation experiments, we maintained patterns of 32 previous cache outcomes (either self or global).

We focus on the predictability of a single level of data cache (two levels makes the

Table 5.1: Benchmark characteristics. For the input data sets of SPEC95 programs, "TRAIN" and "TEST" are the training and testing data sets provided by SPEC, respectively.

Suite	Name	Description	Input Data Sets		Cache Size
			Testing	Alternate Training	
SPEC95 Integer	compress	Compresses and decompresses file in memory	TRAIN	"10000 l 1321"	16 KB
	gcc	GNU C compiler	amptjp.i	stmt.i	64 KB
	go	Computer game "Go"	2stone9.in	null.in	8 KB
	ijpeg	Graphic compression and decompression	vigo.ppm	specmun.ppm	8 KB
	li	LISP interpreter	TRAIN	TEST	8 KB
	m88ksim	Motorola 88000 CPU simulator	TRAIN	TEST	8 KB
	perl	Unix script language Perl	scrabbl.pl	jumble.pl	128 KB
	vortex	Database program	TRAIN	shortened TEST	8 KB
SPEC92 Integer	eqntott	Translation of boolean equations into truth tables	int_pri_3.eqn	fx2fp.eqn (8-bit fix to floating point encoder)	8KB
	espresso	Minimization of boolean functions	cps	bca	16 KB
	sc	Spreadsheet program	loada1	loada2	128 KB
SPLASH-2	raytrace	Ray-tracing program	car	teapot	4KB
Olden	bh	Barnes-Hut's N-body force-calculation	4K bodies	2K bodies	16KB
	bisort	Sorts and merges bitonic sequences	250,000 integers	125,000 integers	8KB
	em3d	Simulates the propagation of E.M. waves in a 3D object	2000 H-nodes, 100 E-nodes	1000 H-nodes, 50 E-nodes	32KB
	health	Simulation of the Columbian health care system	max. level = 10, max. time = 100	max. level = 5, max. time = 50	16KB
	mst	Finds the minimum spanning tree of a graph	512 nodes	256 nodes	8KB
	perimeter	Computes perimeters of regions in images	4K x 4K image	2K x 2K image	16KB
	tsp	Traveling salesman problem	100,000 cities	200,000 cities	8KB
	voronoi	Computes the voronoi diagram of a set of points	20,000 points	10,000 points	8KB
UNIX Utilities	awk	Unix script language AWK	Extensive test of AWK's capabilities	Simple scanning and printing	32KB

Table 5.2: General run-time statistics. Column "Insts" is the total dynamic instruction count; "Loads" is the total dynamic load count (its percentage out of "Insts" is also given); "Load Miss Rate" is the average miss rate of loads; "CP Loads" is the fraction of total dynamic loads that are correlation profiled; "CP Load Misses" is the fraction of total load misses that are correlation profiled.

Suite	Name	Dynamic Statistics				
		Insts	Loads	Load Miss Rate	CP Loads	CP Load Misses
SPEC95 Integer	compress	40M	7M (18%)	4.0%	6%	85%
	gcc	275M	62M (22%)	1.1%	1%	19%
	go	544M	116M (21%)	7.3%	10%	23%
	ijpeg	1382M	253M (18%)	2.5%	3%	22%
	li	209M	46M (22%)	4.3%	9%	75%
	m88ksim	124M	21M (17%)	1.0%	2%	37%
	perl	46M	11M (24%)	1.0%	1%	36%
	vortex	2708M	794M (29%)	2.5%	8%	48%
SPEC92 Integer	sc	929M	178M (19%)	7.2%	11%	80%
	espresso	551M	110M (20%)	2.0%	5%	65%
	eqntott	940M	200M (21%)	5.5%	14%	74%
SPLASH-2	raytrace	2019M	609M (30%)	4.2%	9%	48%
Olden	bh	1404M	343M (24%)	2.3%	7%	81%
	bisort	1091M	271M (25%)	2.6%	6%	63%
	em3d	334M	55M (16%)	1.9%	6%	80%
	health	553M	130M (23%)	3.7%	7%	35%
	mst	72M	11M (16%)	7.4%	21%	67%
	perimeter	103M	15M (14%)	1.3%	3%	40%
	tsp	1078M	142M (13%)	1.6%	8%	58%
	voronoi	175M	60M (34%)	1.7%	5%	55%
UNIX Utilities	awk	45M	7M (15%)	0.4%	1%	19%

analysis too complicated). The choice of data cache size is important because if it is either too large or too small relative to the problem size, predicting dynamic misses becomes too easy (they either always hit or always miss). Therefore we would like to operate near the "knee" of the miss ratio curve, where predicting dynamic hits and misses presents the greatest challenge. Although we could potentially reach this knee by altering the problem size, we had greater flexibility in adjusting the cache size within a reasonable range. We chose the data cache size as follows. We first used summary profiling to collect the miss ratios of all loads within the application on different cache sizes ranging from 4KB to 128KB. We then chose the cache size which resulted in the largest number of significant loads having intermediate miss ratios—these sizes are shown in Table 5.1. In all cases, we model a two-way set-associative with 32 byte lines and cache miss latency of 20 cycles.

Table 5.2 shows some general run-time statistics of these applications. One important observation from it is that, in many cases, a substantial portion of misses are covered by a relatively small amount of correlation-profiled loads. This happens because it is common that only a few static loads account for most misses in the program (a similar observation was made by Abraham *et al.* [2]), and this characteristic is essential for collecting correlation profiling information in practice.

5.4.2 Improvements in Prediction Accuracy and Stall Time

Figures 5.6 and 5.7 show how the three correlation profiling schemes—control-flow (**C**), self (**S**), and global (**G**)—improve the prediction accuracy of correlation-profiled loads. As discussed earlier in Section 5.3, our threshold for deciding whether to apply latency tolerance to a reference is that its miss ratio must exceed $\frac{V}{L}$, where V is the latency tolerance overhead and L is the miss latency. For summary profiling, this threshold is applied to the overall miss ratio of an instruction; for correlation profiling, it is applied to groups of dynamic references sharing individual contexts. Figures 5.6 and 5.7 show the results with two values of $\frac{V}{L}$: 0.1 and 0.25, respectively.

Two training input sets were used in each application: **same** means the training and the testing inputs were identical whereas **diff** means the alternate training input was used. Each bar in Figures 5.6 and 5.7 corresponds to the misprediction rate of the scheme it represents, and is broken down into two categories. The top section ("*Predict HIT / Actual MISS*") represents a *lost opportunity* where we predict that a reference hits (and thus do not attempt to tolerate its latency), but it actually misses. The "*Predict MISS / Actual HIT*" section accounts for *wasted overhead* where we apply latency tolerance to a reference that actually hits.

At first glance, the misprediction rates are surprisingly higher than usual *branch* misprediction rates, especially for $\frac{V}{L} = 0.10$. This phenomenon can be explained by two reasons. First, by our definition correlation-profiled loads are those that have intermediate miss ratios and hence will not have strong hit or miss biases. Second, unlike the case in branch prediction, since failing to hide a miss is more expensive than wasting overhead, it is reasonable to predict misses more often in order to achieve higher performance even if this will result in a larger number of "*Predict MISS / Actual HIT*" mispredictions.

It is clear from Figures 5.6 and 5.7 that correlation profiling improves prediction accuracy[1] over summary profiling in nearly all cases. Among the three correlation pro-

[1]Again, because the penalties of mispredicting a hit and a miss is asymmetric, it is possible to improve performance by replacing more expensive with less expensive mispredictions, even if the total misprediction count increases (e.g., go with correlation profiling when $\frac{V}{L} = 0.25$).

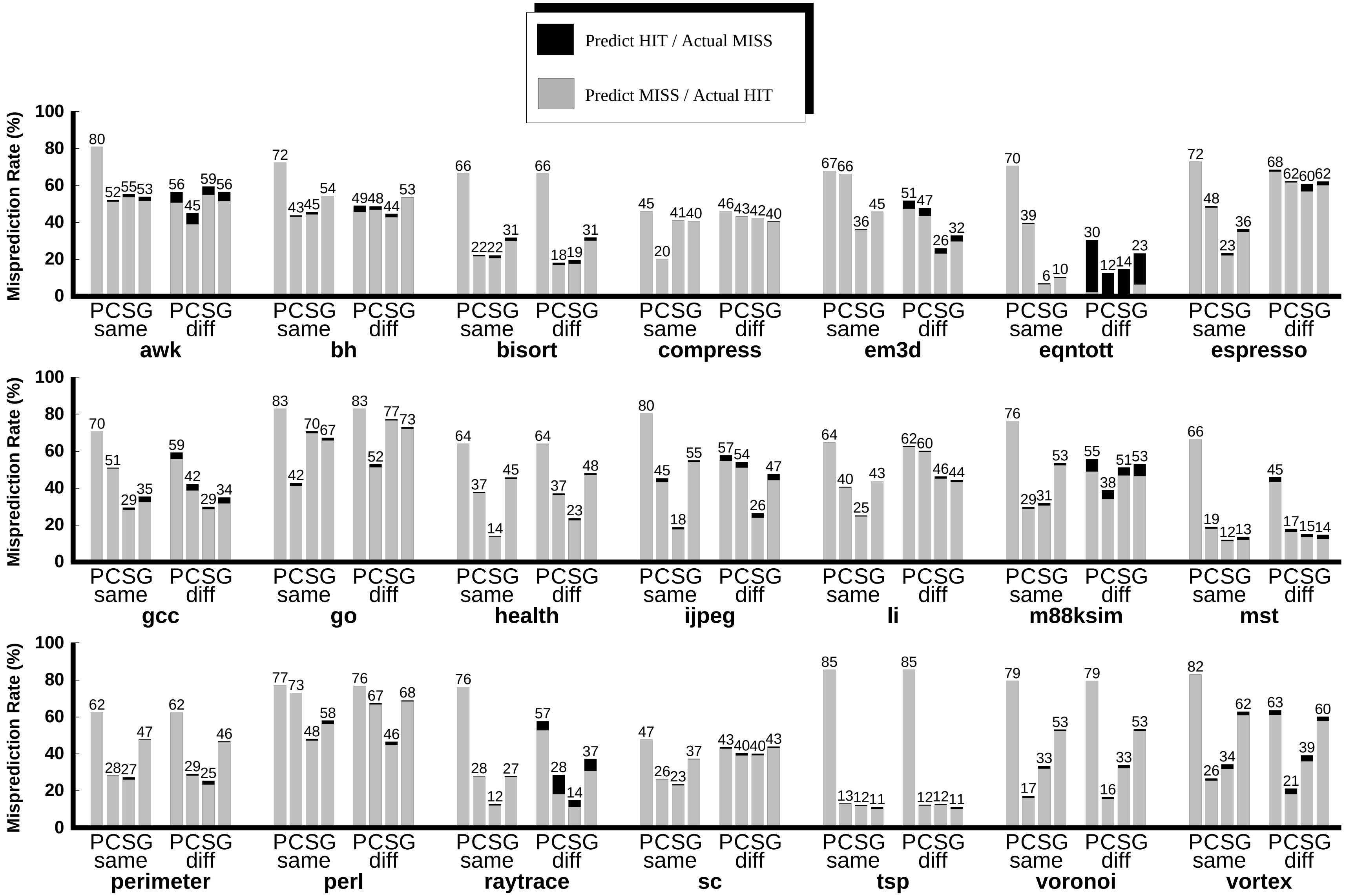


Figure 5.6: Misprediction rates of correlation-profiled loads with $\frac{V}{L} = 0.10$. For each application, training inputs that are identical to (**same**) and different from (**diff**) the testing input are both used. Four prediction schemes (**P** = summary profiling, **C** = control-flow correlation, **S** = self correlation, **G** = global correlation) are shown for each training input.

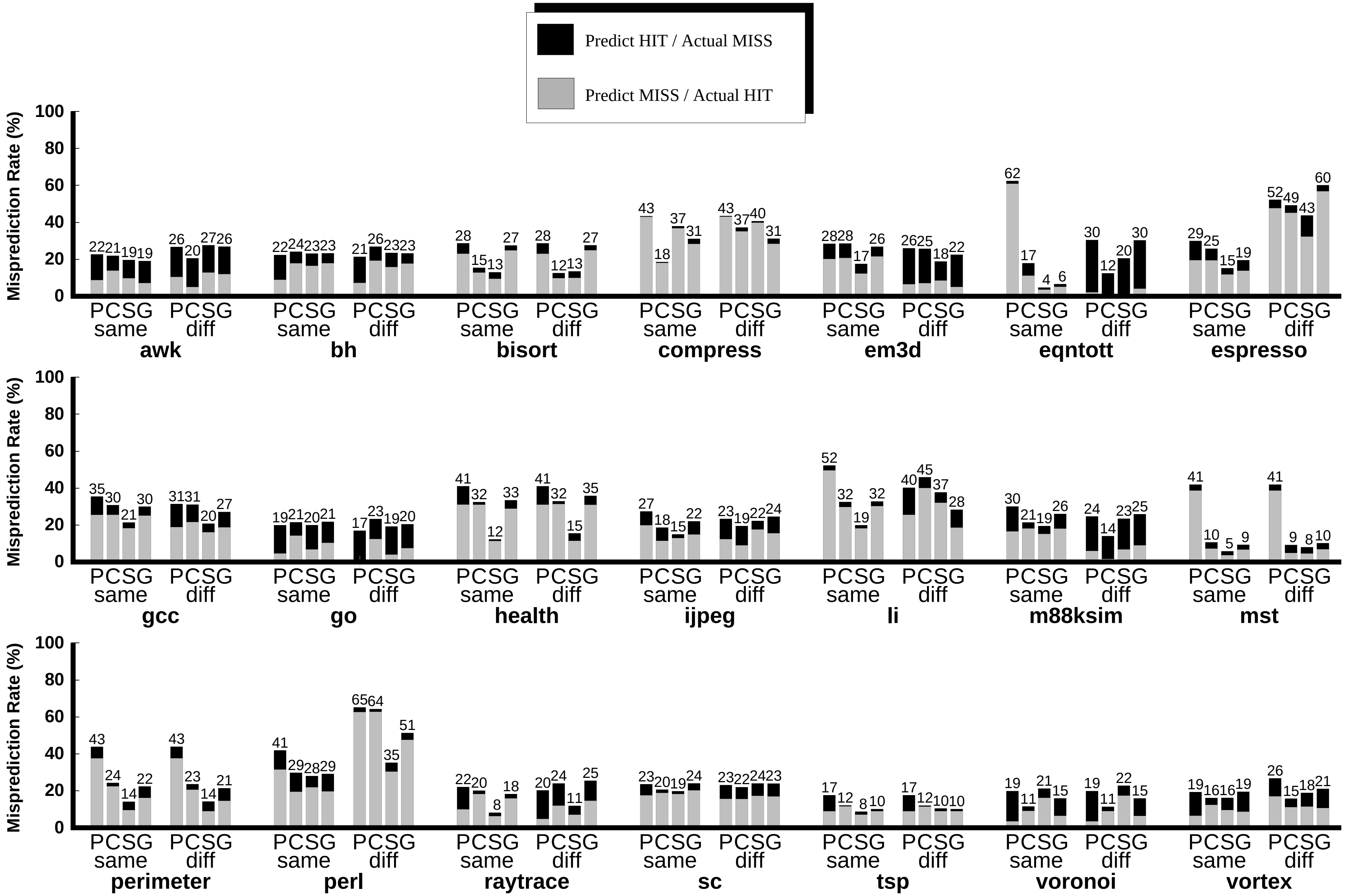


Figure 5.7: Misprediction rates of correlation-profiled loads with $\frac{V}{L} = 0.25$. For each application, training inputs that are identical to (**same**) and different from (**diff**) the testing input are both used. Four prediction schemes (**P** = summary profiling, **C** = control-flow correlation, **S** = self correlation, **G** = global correlation) are shown for each training input.

filing schemes, self correlation achieves the highest prediction accuracy in the largest number of cases, followed by control-flow correlation and then by global correlation. With $\frac{V}{L} = 0.1$, summary profiling tends to apply latency tolerance aggressively, thus resulting in a large amount of wasted overhead. In contrast with $\frac{V}{L} = 0.25$, summary profiling tends to be more conservative, thereby resulting in more untolerated misses. Fortunately, for both ratios, correlation profiling successfully bridges the gap between summary profiling and an ideal prediction.

As we expect, the effectiveness of these profiling schemes relies on how closely the miss behaviors in the training run resemble those in the testing run. Fortunately, though the prediction accuracy of correlation profiling is generally lower when the alternative training input set is used, the improvement over summary profiling is still significant. Overall, correlation profiling does not appear to be overly sensitive to the training input.

To quantify the performance benefits of this increased prediction accuracy, Figures 5.8 and 5.9 show the potential resultant effective load stall time (recall that effective stall time is the ordinary stall time plus any overhead associated with latency tolerance) of applying the four profiling schemes to a generic latency tolerance technique. In this experiment, we assume an ideal exploitation of correlation profiling, where isolating different dynamic contexts incurs no run-time overhead. Later in Sections 5.6.1 and 5.6.2, when we apply correlation profiling to dynamic instruction scheduling and prefetching, we will also include the overheads (if any) of isolating dynamic contexts.

Each bar in Figures 5.8 and 5.9 is normalized to the load stall time without latency tolerance, and is broken down into three categories. The bottom section ("*Predict MISS / Actual MISS*") is the *useful* overhead paid for tolerating references that normally miss. The top two sections represent the *misprediction penalty*, including wasted overhead ("*Predict MISS / Actual HIT*") and untolerated miss latency ("*Predict HIT / Actual MISS*").

As we see in these two figures, the increased prediction accuracy offered by correlation profiling translates into noticeable to significant reductions in load stall time over summary profiling. Such reductions are generally greater when $\frac{V}{L} = 0.25$, a more intermediate threshold, with which summary profiling tends to mispredict more often. The extent to which improved prediction accuracy translates into reduced overall load stall time also depends on the fraction of load misses that are correlation profiled. When this factor and the improvement in prediction accuracy are both favorable (e.g., `eqntott` and `tsp`), we see large reductions in load stall time—when either factor is small (e.g., `compress` and `gcc`), the reductions are only modest.

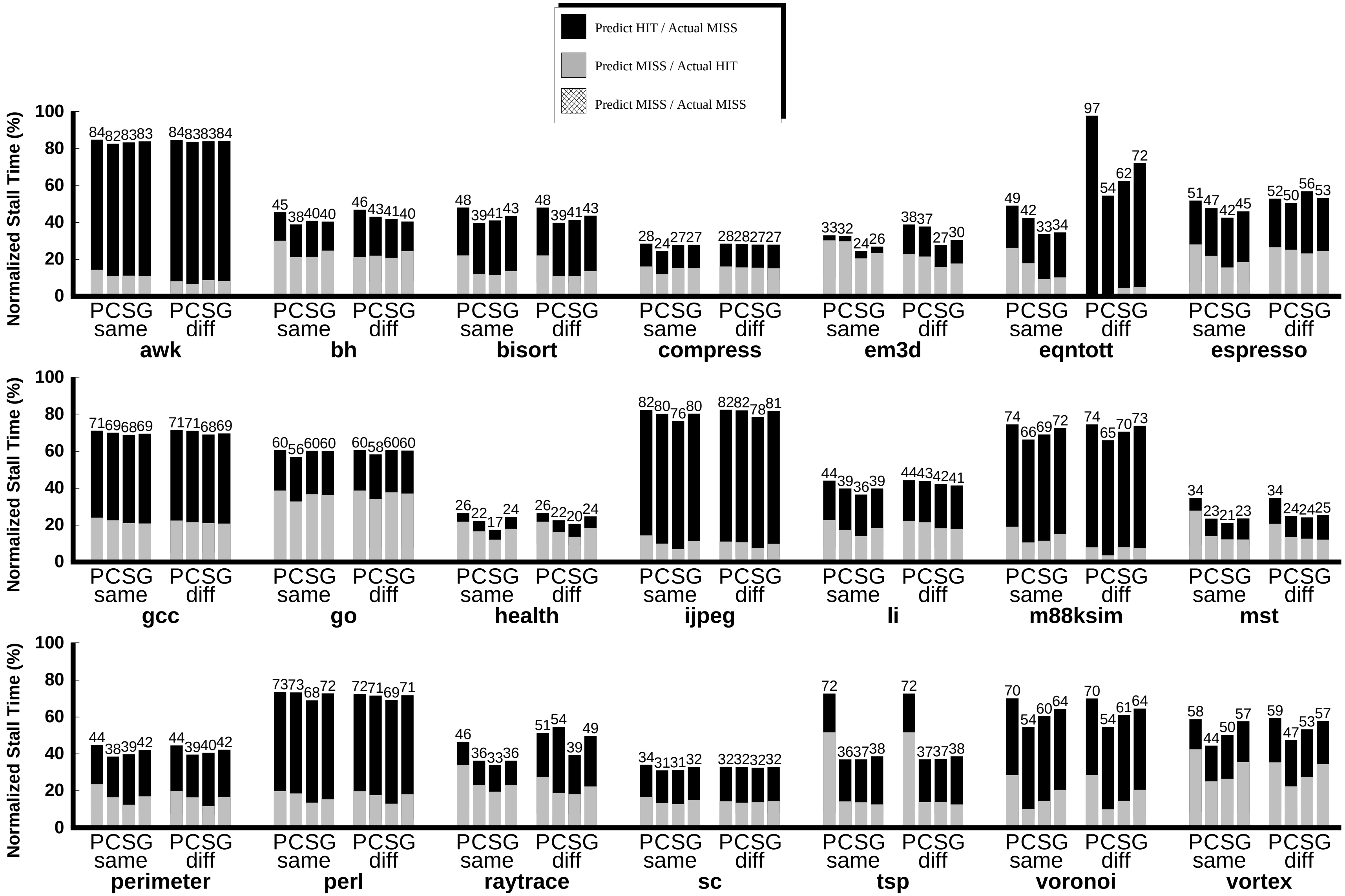


Figure 5.8: Potential impact of correlation profiling on load stall time with $\frac{V}{L} = 0.10$. For each application, training inputs that are identical to (**same**) and different from (**diff**) the testing input are both used. Four prediction schemes (**P** = summary profiling, **C** = control-flow correlation, **S** = self correlation, **G** = global correlation) are shown for each training input.

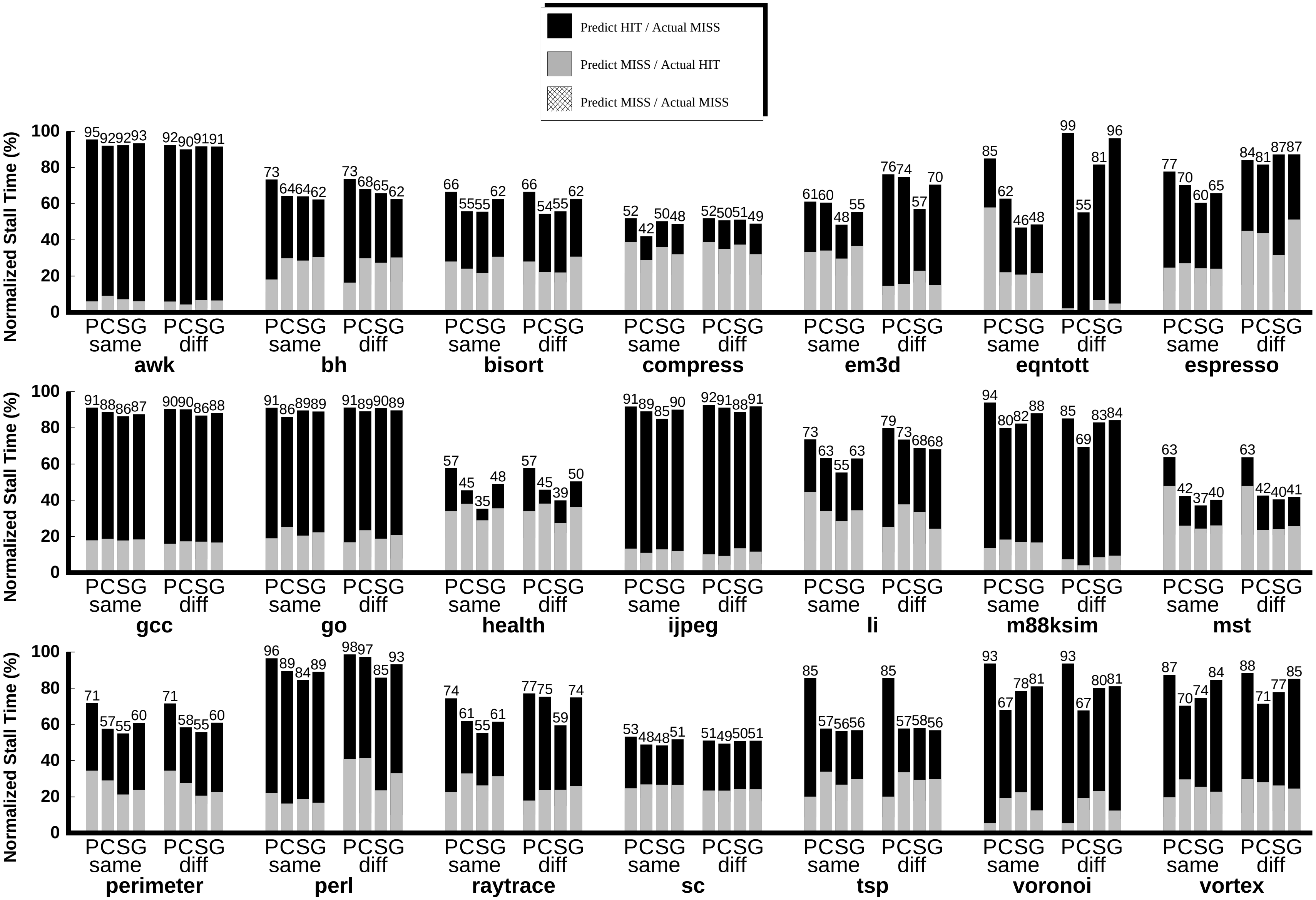


Figure 5.9: Potential impact of correlation profiling on load stall time with $\frac{V}{L} = 0.25$. For each application, training inputs that are identical to (**same**) and different from (**diff**) the testing input are both used. Four prediction schemes (**P** = summary profiling, **C** = control-flow correlation, **S** = self correlation, **G** = global correlation) are shown for each training input.

5.5 Case Studies

To develop a deeper understanding of when and why correlation profiling succeeds, we now examine a number of the applications in greater detail. We discuss the relevant memory access patterns in each of these applications. For the sake of demonstration, we present in the study of `li` the *miss ratio distributions* under different profiling schemes, which give us insight into how effectively correlation profiling has isolated the dynamic hit and miss instances of static load instructions.

5.5.1 li

Over half of the total load misses are caused by two pointer dereferences: `this`→`n_flags` in `mark()`, and `p`→`n_flags` in `sweep()`, as illustrated by the pseudo-code in Figure 5.10.

The access patterns behave as follows. The procedure `mark()` traverses a binary tree through the three `while` loops shown in Figure 5.10(a). Starting at a particular node, the first inner `while` loop continues descending the tree—choosing either the left or right child as it goes—until it reaches either a marked node or a leaf node. At this point, we then backup to a node where we can continue descending through a search performed by the second inner `while` loop. The tree is allocated in preorder, similar to the one shown in Figure 5.3, but much bigger. Therefore we enjoy spatial locality as long as we continue following left branches in the tree, but spatial locality is disrupted whenever we backup in the second inner `while` loop, as illustrated by Figure 5.10(c).

All three types of correlation profiling provide better cache outcome predictions than summary profiling for the `this`→`n_flags` reference in `mark()` for `li`. Self correlation detects this form of spatial locality effectively. Global correlation is more accurate than summary profiling but less accurate than self correlation in this case because the cache outcomes of other references (which do not help to predict this reference) consume wasted space in the global history pattern. Control-flow correlation also performs well because it observes that `this`→`n_flags` is more likely to suffer a miss if we begin iterating in the first inner `while` loop immediately following a backup performed in the second inner `while` loop (in the preceding outer `while` loop iteration).

Finally, the reference `p`→`n_flags` in `sweep()` (shown in Figure 5.10(b)) is in fact an array reference written in pointer form. Both self correlation and global correlation detect the spatial locality caused by accessing consecutive elements within the array. (Although the compiler could potentially recognize this spatial locality through static analysis if it can recognize that `p`→`n_flags` is effectively an array reference, this is not always possible for all such cases.)

(a) Procedure `mark()`

```
void mark(NODE *ptr) {
  NODE *this,*prev,*tmp;
  if (ptr == NIL) return;
  prev = NIL; this = ptr;
  ...
  while (TRUE) { /* outer while loop */
    while (TRUE) {/* 1st inner while loop */
      if (this→n_flags & MARK)
        break; /* a marked node */
      else {
        ...;
        if (livecar(this)) {
          ...;
          prev = this;
          this = car(prev); /* go left */
          ...;
        }else if (livecdr(this)) {
          ...;
          prev = this;
          this = cdr(prev); /* go right */
          ...;
        }else break; /* a leaf node* /
      } /* ends if-else */
    } /* ends 1st inner-while */
    while (TRUE) {/* 2nd inner while loop */
      /* backup to a point where we
      can continue descending */
    } /* ends 2nd inner while */
  } /* end 1st outer while */
```

(b) Procedure `sweep()`

```
LOCAL sweep(){
  struct segment *seg;
  NODE *p; int n;
  ...
  for (seg = segs; seg != NULL;
    seg = seg→sg_next) {
    p = &seg→sg_nodes[0];
    for (n = seg→sg_size; n--; p++)
      if (!(p→n_flags & MARK)) {
        ...
      }
  }
}
```

(c) Tree traversal order in `mark()`

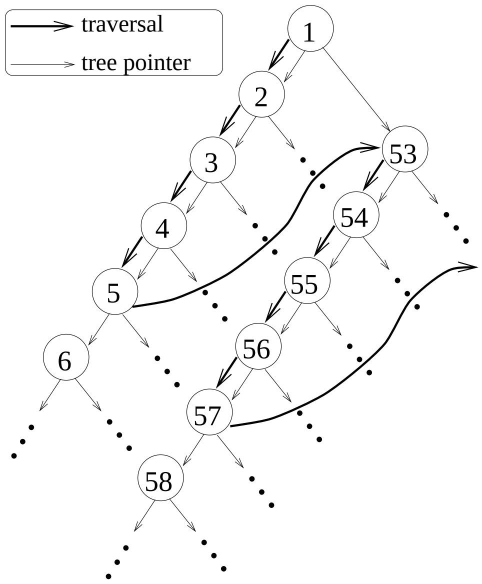


Figure 5.10: Procedures `mark()` and `sweep()` in `li`, and the memory access patterns of `mark()`. (Note: consecutively numbered nodes in part (c) correspond to adjacent addresses in memory.)

Figures 5.11 (a)-(d) show the miss ratio distribution of `li` under the four profiling schemes. Each graph is divided into 20 ranges of miss ratios, each of which contains a bar corresponding to the fraction of total dynamic correlation-profiled load references that fall within this range. The bars for summary profiling (Figure 5.11(a)) represent the

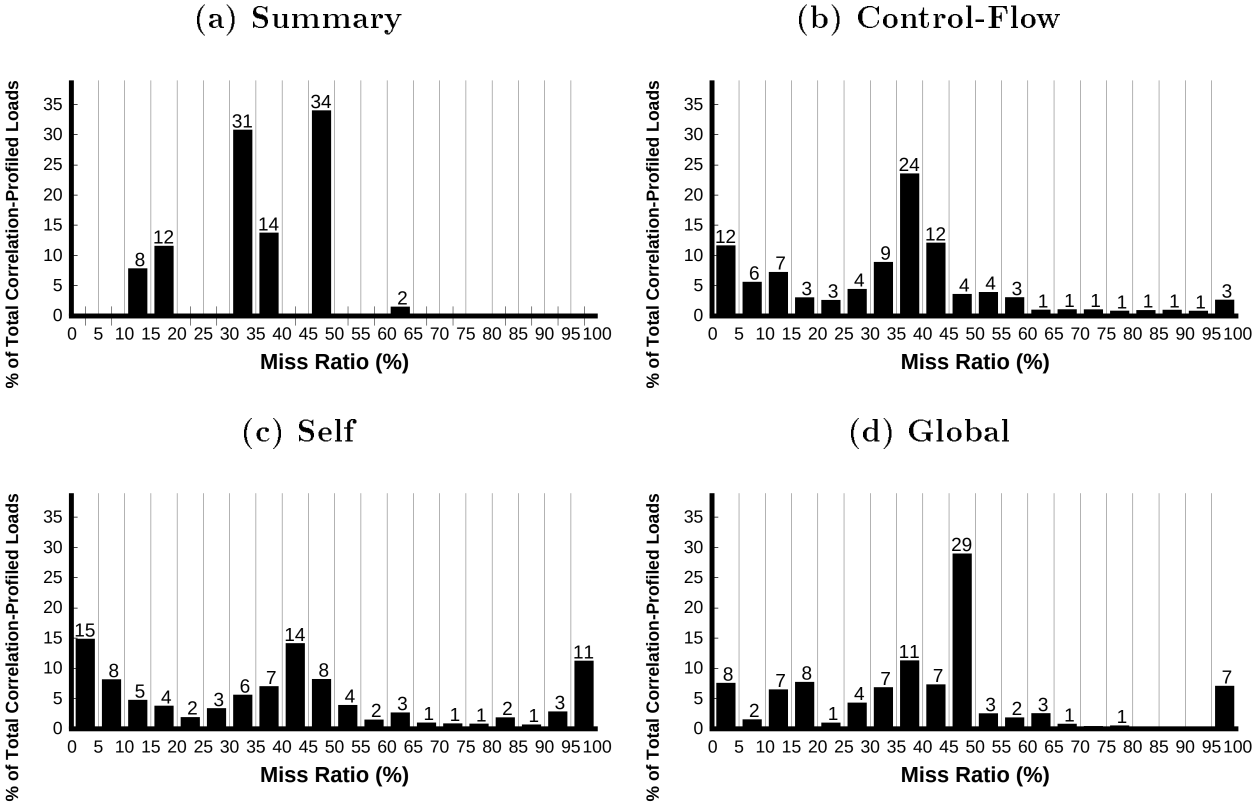


Figure 5.11: Miss ratio distribution of `li` under the four profiling schemes.

inherent miss ratios of these load instructions, and the other three cases represent the degree to which correlation profiling can effectively group together dynamic instances of the loads into separate contexts with similar cache outcome behavior. For a correlation scheme to be effective, we would like to see a "U-shaped" distribution where references have been isolated such that they always have very high or very low miss ratios—we refer to such a case as being *strongly biased*. In contrast, if most of the references are clustered around the middle of the distribution, we say that this is *weakly biased*. Correlation profiling can outperform summary profiling by increasing the degree of bias, which we do observe in Figures 5.11(b)-(d). With summary profiling, 79% of the loads that we profile[2] have miss ratios in the range of 30-50% (these include the `this→n_flags` and `p→n_flags` references shown earlier in Figure 5.10). In contrast, with self correlation profiling only 35% of the isolated loads have miss ratios in the 30-50% range, and over 37% are either below 10% or above 90%. All three correlation schemes increase the degree

[2]Recall that we only profile loads with miss ratios between 10% and 90% among the top 15 ranked loads in terms of their contributions to total misses. Therefore the summary profiling case will never have loads outside of this miss ratio range.

(a) Procedure `cmppt()` which causes most load misses

```
extern int ninputs, noutputs;
int cmppt(a, b)
PTERM *a[], *b[]; {
  register int i, aa, bb;
  register int* a_ptand, *b_ptand;
  a_ptand = a[0]→ptand;
  b_ptand = b[0]→ptand;
  for (i = 0; i < ninputs; i++) {
    aa = a_ptand[i];
    bb = b_ptand[i];
    /* the famous correlated branches */
  }
  return (0);
}
```

(b) Call-site dependent cache outcome patterns

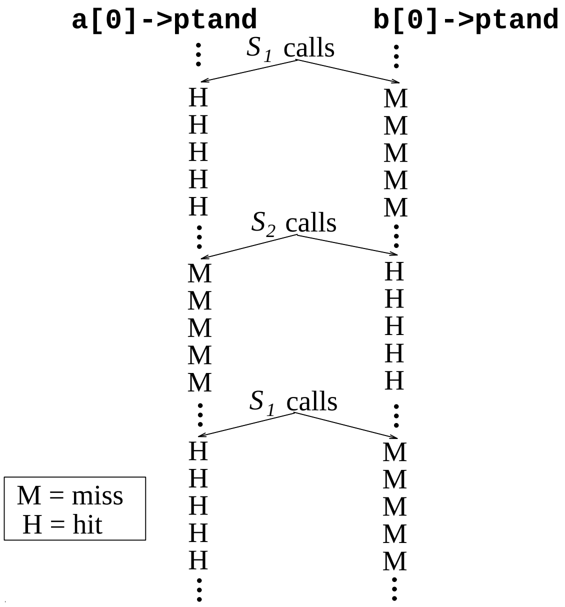


Figure 5.12: The memory access behavior in `eqntott`. To make all loads explicit, we rewrite the two expressions `a[0]→ptand[i]` and `b[0]→ptand[i]` in the original `cmppt()` into the four loads (i.e. `a[0]→ptand`, `a_ptand[i]`, `b[0]→ptand`, and `b_ptand[i]`) shown in (a).

of bias in this case.

5.5.2 eqntott

Most of the load misses in `eqntott` are caused by the four loads in `cmppt()` shown in Figure 5.12(a), two of which are array references (`a_ptand[i]` and `b_ptand[i]`). Clearly the spatial locality enjoyed by these two array references can be detected through self correlation (and hence global correlation). However, the access patterns of the other two loads (`a[0]→ptand` and `b[0]→ptand`) are more complicated. The procedure `cmppt()` has multiple call sites, and two of them, say S_1 and S_2, invoke it very frequently. Whenever `cmppt()` is called at S_1, `a[0]` will very likely be unchanged but `b[0]` will have a new value. In contrast, whenever `cmppt()` is called at S_2, `b[0]` will very likely be unchanged but `a[0]` will have a new value. Moreover, both S_1 and S_2 repeatedly call `cmppt()`. This call-site dependent behavior results in the streams of cache outcomes illustrated in Figure 5.12(b). Self correlation captures these streaming behavior, and control-flow correlation also predicts the cache outcomes accurately by distinguishing the two call sites of `cmppt()`.

```
ELEMENT **prims_in_box2(pepa, ...){
ELEMENT **pepa;
  ...
  k = 0;
  npepa = alloc(...);
  for (j = 0; j < n_in; j++){
    tmp = pepa[j];
    bb = tmp→bv;
    ...
    /* computes ovlap */
    /* no change in pepa[j] */
    if (ovlap == 1) {
      npepa[k++] = pepa[j];
      ...
    }
  }
  return (npepa);
}
```

```
void subdiv_bintree(BTNODE* btn, ...){
  ...
  /* btn1 and btn2 are btn's children */
  btn1→pe = prims_in_box2(btn→pe, ...);
  ...
  btn2→pe = prims_in_box2(btn→pe, ...);
  ...
}
void create_bintree(BTNODE* root, ...){
  ...
  if (...)  {
    subdiv_bintree(root, ...);
    create_bintree(root→btn[0], ...);
    create_bintree(root→btn[1], ...);
    ...
  }
  ...
}
```

Figure 5.13: Pseudo codes drawn from `raytrace`.

The cache outcomes of `a[0]→ptand` also help predict those of `a_ptand[i]`—if `a[0]→ptand` is a hit, it implies that the array `a_ptand[]` has been loaded recently, and therefore the `a_ptand[i]` references are likely to also hit. (Similar correlation also exists between `b[0]→ptand` and `b_ptand[i]`). Hence global correlation is quite effective in this case. Control-flow correlation also predicts the cache outcomes of `a_ptand[i]` and `b_ptand[i]` in an indirect fashion, by virtue of predicting those of `a[0]→ptand` and `b[0]→ptand`.

5.5.3 raytrace and tsp

In `raytrace`, over 30% of load misses are caused by the pointer dereference of `tmp→bv` in `prims_in_box2()` (see Figure 5.13). In `subdiv_bintree()`, the two calls to `prims_in_box2()` copy part of the array `pe` of the current node `btn` to the arrays `btn1→pe` and `btn2→pe`, where `btn1` and `btn2` are the children of `btn`. This process of copying `pe` is performed recursively on the whole tree by `create_bintree()`. As a result, when `prims_in_box2()` is called upon a node n, we may have used all values in the array `pe` (referred to as `pepa` in `prims_in_box2()`) of n before at some antecedent of n and hence hopefully most data loaded by `tmp→bv` is already in the cache. In this case, most references of `tmp→bv` will hit in the cache. In contrast, if the values in `pepa` are new, all `tmp→bv` references will miss. Hence self correlation captures these streams of hits and streams of

```
Tree tsp(Tree t,int sz, ...)  {
  ...
  if (t→size <= sz) return conquer(t);
  ...
  leftval = tsp(t→left, sz, ...);
  rightval = tsp(t→right,sz, ...);
  return merge(leftval, rightval, t, ...);
}
```

```
static Tree conquer(Tree t) {
  ...
  l = makelist(t);
  for (; l; l=donext) {
    work(l→data);
    donext = l→next;
  }
  ...
}
```

Figure 5.14: Pseudo codes drawn from `tsp`. Procedure `makelist(Tree t)` slings `t` into a list consisting of all nodes of t.

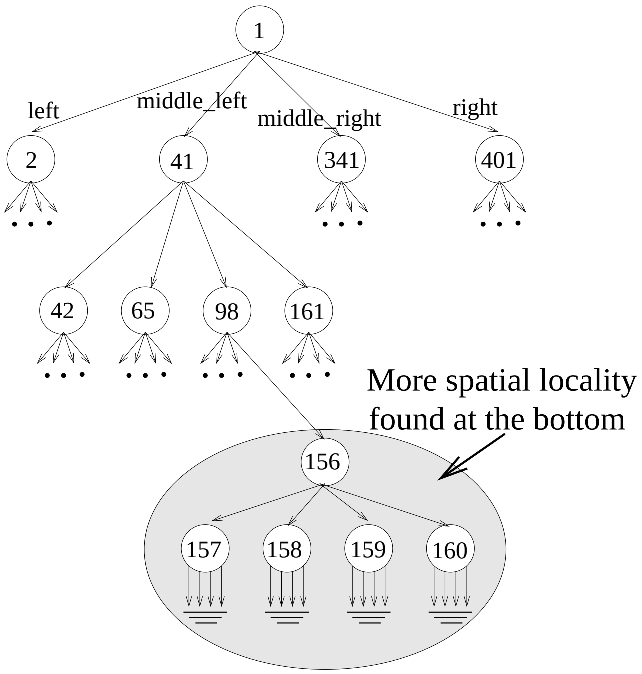
misses. In theory, control-flow correlation could also achieve good predictions by observing whether any copying occurred in the parent node—unfortunately, the profiling tool cannot record enough state across the many control-flow changes in `subdiv_bintree()` and `prims_in_box2()` to know what decisions were made in the parent node.

Similar to `raytrace`, `tsp` also traverses a binary tree recursively, and some data which is read by the current node will be read again by its descendents. As illustrated in Figure 5.14, the procedure `tsp()` recursively traverses the tree `t` and calls `conquer(t)` if the size of `t` is not greater than `sz`. The procedure `conquer(t)` uses `makelist(t)` to sling *every* node of `t` into a list which is then traversed by the `for` loop. Therefore since all descendents of `t` are brought into the cache whenever `conquer(t)` is called, subsequent recursion down `t→left` and `t→right` within `tsp()` results in many cache hits. Hence the `l→data` references either mainly hit or mainly miss for a given node `t`. Self correlation captures this pattern effectively. Control-flow correlation is also quite effective because it can observe the number of times `conquer()` has been called in a given recursive descent—most misses occur the first time it is invoked.

5.5.4 perimeter and bisort

The main data structures used in both `perimeter` and `bisort` are trees: quadtrees in `perimeter`, and binary trees in `bisort`. These trees are allocated in preorder, but the orders in which they are traversed are rather arbitrary. As a result, we do not see very regular cache outcome patterns (such as the one illustrated in Figure 5.3) for these applications. Nevertheless, there is still a considerable amount of spatial locality among consecutively accessed nodes while we are traversing around the *bottom* of a tree that has been allocated in preorder. For example, if we traverse a quadtree using the procedure `middle_first()` shown in Figure 5.15, we will only miss twice upon accessing nodes 156

(a) A quadtree allocated in preorder



(b) Code for traversing the quadtree in (a)

```
void middle_first(quadTree* p) {
  if (p == NULL)
    return;
  work(p→data);
  middle_first(p→middle_left);
  middle_first(p→middle_right);
  middle_first(p→left);
  middle_first(p→right);
}
```

Figure 5.15: Example of a case where more spatial locality is found at the bottom of a tree. This example assumes that one cache line can hold three tree nodes and the tree is allocated in preorder. Nodes having consecutive numbers are adjacent in the memory.

through 160 at the tree's bottom, assuming that nodes 156 through 158 are in one cache line and nodes 159 through 161 are in another. In contrast, there is relatively little spatial locality while we are traversing the middle of the tree. Self correlation (and hence global correlation) can discover whether we are currently in a region of the tree that enjoys spatial locality. Control-flow correlation can also potentially detect whether we are close to the bottom of the tree by noticing the number of levels of recursive descent.

5.5.5 mst

Most of the misses in `mst` are caused by loads in `HashLookup()` and the `tmp→edgehash` load in `BlueRule()`, as illustrated in Figure 5.16. The `mst` application consists of two phases: a creation phase and a computation phase. Both phases invoke `HashLookup()`, but the creation phase causes most of the misses when it calls `HashLookup()` to check whether a key already exists in the hash table before allocating a new entry for it. During the computation phase, much of the data has already been brought into the cache, and hence there are relatively few misses. Both self correlation and global correlation

```
void *HashLookup(int key, Hash hash) {
  int j;
  HashEntry ent;
  j = (hash→mapfunc)(key);
  for (ent = hash→array[j];
    ent && ent→key !=key;
    ent = ent→next);
  if (ent) return ent→entry;
  return NULL;
}
```

```
static BlueReturn BlueRule(...)  {
  ...
  for (tmp=vlist→next; tmp;
    prev=tmp,tmp=tmp→next) {
    ...
    hash = tmp→edgehash;
    ...
  }
}
```

Figure 5.16: Pseudo codes drawn from `mst`.

accurately predict the cache outcomes of these two distinct phases, since they appear as repeated streams of either hits or misses. Control-flow correlation is also effective since it can distinguish the call chains which invoke `HashLookup()`.

The load of `tmp→edgehash` in `BlueRule()` accesses a linked lists whose nodes are in fact allocated at contiguous memory locations. Consequently, self correlation detects this spatial locality accurately, but control-flow correlation is not helpful.

5.5.6 voronoi and compress

Control-flow correlation offers the best prediction accuracy in both of these applications. Most of the misses in `voronoi` are caused by loading `b→next` in `splice()`, which is called from three different places in `do_merge()`, as illustrated in Figure 5.17(a). When `splice()` is called from *call site 1*, `b→next` will hit since `ldi→next` loaded this same data into the cache just prior to the call. When `splice()` is called from the other two call sites, `b→next` is more likely to miss. Hence control-flow correlation distinguishes the behavior of these different call sites accurately. Self correlation is less effective since `b→next` does not have regular cache outcome patterns.

In `compress` roughly half of the misses are caused by the hash table access `htabof[i]` in the procedure `compress()` (see Figure 5.17(b)). The index `i` to the hash table `htab` is a function of the combination of the prefix code `ent` and the new character `c`. If this combination has been seen before, the hash probe test (`(htab[i] == fcode)`) will be true—if it has been seen *recently*, the load of `htab[i]` is likely to hit in the cache. Since the input file we use (provided by SPEC) is generated from a frequency distribution of common English texts, some strings will appear more often than others. Because of this, we expect that the condition `(htab[i] == fcode)` should be true quite frequently once many common strings have been entered into `htab`. If the last few tests of `(htab[i]`

(a) Code fragment in `voronoi`

```
EDGE_PAIR do_merge(...) {
  ...
  v = ldi→next;
  b = ldi;
  splice(a, b) /* call site 1 */
  ...
  /* no dereferences of ldj before */
  b = ldj;
  splice(a, b) /* call site 2 */
  ...
  /* no dereferences of ldk before */
  b = ldk;
  splice(a, b) /* call site 3 */
  ...
}
splice(QUAD_EDGE a, QUAD_EDGE b) {
  ...
  beta = rot(b→next);
  ...
}
```

(b) Code fragment in `compress`

```
compress() {
  ...
  while ((c = getbyte()) != EOF) {
    ...
    fcode = (((long) c << maxbits) + ent);
    i = (xor((c << hshift), ent));
    if (htab[i] == fcode) {
      ent = codetab[i];
      continue;
    } else {
      ... /* store fcode into htab */ ...
    }
  }
  ...
}
```

Figure 5.17: Pseudo codes drawn from (a) `voronoi` and (b) `compress`.

`== fcode)` are false, the probability that the next one is true will be high, which also implies that the next reference of `htab[i]` is more likely a hit. Therefore, control-flow correlation can make accurate predictions by examining the last several outcomes of this branch.

5.5.7 espresso, vortex, m88ksim, and go

For these four applications, correlation profiling mainly improves the cache outcome predictions for *array* references. In `espresso`, many load misses are due to array references, written in pointer form, with variable strides. Figure 5.18(a) shows one such example. Inside the `for` loop, `p` is incremented by `BB→wsize`, whose value depends on the call chain of `setup_BB_CC()` and ranges from 4 to 24 bytes. Different values result in different degrees of spatial locality, but all can be captured by self correlation (and hence global correlation). Control-flow correlation can also make enhanced predictions by exploiting the call-chain information.

In `vortex`, `m88ksim`, and `go`, many load misses are caused by array references located

(a) Code fragment in `espresso`

```
void setup_BB_CC(pcover BB,
          pcover CC){
  ...
  for(p=BB→data,
    last=p+BB→count*BB→wsize;
    p<last;p+=BB→wsize)
      p[0] = p[0] | ACTIVE;
  ...
}
```

(b) Code fragment in `vortex`

```
boolean ChkGetChunk(numtype ChunkNum, ...){
  ...
  if (((Theory→Flags[ChunkNum] & ...))
      && ...
  ...
}
```

Figure 5.18: Pseudo codes drawn from (a) `espresso` and (b) `vortex`.

inside procedures, where array indices are passed as procedure parameters. See Figure 5.18(b) for an example drawn from `vortex`. Each of these procedures have multiple call sites, and the cache outcomes of those array references are mainly call-site dependent. This explains why control-flow correlation offers the highest cache outcome prediction accuracy for these three benchmarks. In `vortex`, the array index parameter values at a given call are very close or even identical most of the time, but values passed at different call sites are quite different. Consequently, references made through the same call sites will enjoy temporal and/or spatial locality, but those made through different call sites will not. Since a procedure is usually invoked multiple times by the same call site before being invoked by another call site, this results in a streaming pattern of a miss followed by several hits—hence self correlation also performs well in `vortex` by capturing these cache outcome patterns.

5.5.8 Lessons Learned from All Case Studies

Although global correlation makes excellent predictions in some cases by correlating behavior across different load instructions (e.g., `eqntott`), in most cases it essentially assimilates self correlation, but does not perform quite as well since it records less history for a given load. Self correlation is often successful since it recognizes forms of spatial locality which are not recognizable at compile time (e.g, `li`, `perimeter`, `bisort`, and `mst`), and also long runs of either all hits or all misses (e.g., `eqntott`, `mst`, `tsp`, and `raytrace`). We often find that as few as four previous cache outcomes per reference are sufficient to achieve good predictability with self correlation. By capturing call chain information, control-flow correlation can distinguish behavior based on call sites (e.g., `eqntott`, `espresso`, `vortex`, `m88ksim`, `go`, `mst` and `voronoi`) and the depth of the recursion while traversing a tree (e.g., `perimeter`, `bisort`, and `tsp`). Another interesting finding is that array-like references with regular strides still play an important role in a considerable number of these non-numeric applications (e.g., `eqntott`, `li`, `espresso`, `vortex`, `m88ksim` and `go`). However, these array references are often written in more complex forms than in numeric applications, including using pointer arithmetic to compute addresses and passing array indices as procedure parameters.

Over half of the applications enjoy significant improvements from *both* control-flow and self correlation, and in many of these cases we observe that the same load references can be successfully predicted by both forms of correlation. This is good news, since control-flow correlation profiling is the easiest case to exploit in software by using *procedure cloning* [34] to distinguish call-chain dependent behavior (we will discuss this further in Section 5.6.2).

5.6 Exploiting Correlation Profiling in Practice

Now that we have demonstrated correlation profiling's potential for predicting cache outcomes more accurately than summary profiling, the next question is *how do we exploit these potential performance gains in practice?* Any practical exploitation of correlation profiling must consist of the two key components: context-based cache miss profiling and isolation of contexts at run time. In this section, we propose how correlation profiling could be exploited in *hardware* and in *software*, in light of those proven hardware and software history-based branch prediction techniques. In addition, we apply correlation profiling to two latency tolerance techniques to further demonstrate its practicality.

5.6.1 Exploiting Correlation Profiling in Hardware

Let us consider some possibilities for hardware-based miss predictors that utilize the three types of correlation. For control-flow correlation, a straightforward hardware-based predictor would be to use *branch histories* to predict cache outcomes. An advantage of this approach is that branch histories are free if they are already used for branch prediction. However, the main drawback is that only branch *directions* are used in most history-based branch predictors, not the actual *control-flow paths.* Without control-flow path information (especially procedure call contexts), prediction accuracy would not be improved much. For this reason, we do not exploit control-flow correlation in hardware. Instead, we will consider exploiting control-flow correlation in software later in Section 5.6.2. Nevertheless, control-flow correlation would be more readily exploitable in hardware if the branch predictor already makes use of path information, such as the one proposed by Nair [100].

For self and global correlation, we can borrow the designs of existing history-based branch predictors to construct our own cache miss predictors: we simply need to replace the notion of taken/not-taken in branch prediction by that of hit/miss. In such predictors, both context-based cache miss profiling and run-time context isolation are implemented entirely in hardware. Cache miss profiling is essentially achieved through an array of hardware counters (typically saturating), which keep track of recent cache hit/miss outcomes and are also used to make prediction in the future. These counters are indexed by history patterns stored in hardware registers, which essentially isolate run-time contexts. As proof of this concept, we implement both self and global correlation-based miss predictors in hardware and apply them to *instruction scheduling*, as detailed in the next section. But, as we discussed before, they would also be useful for other latency tolerance techniques, such as multithreading.

Applying Cache Miss Prediction to Dynamic Instruction Scheduling

Consider the pipelines of a generic out-of-order machine shown in Figure 5.19(a). In this machine, an instruction takes s cycles to get scheduled to the corresponding function unit. Suppose that the scheduling of a load L begins at time t. Whether L is a cache hit is unknown until the D-cache access of L nearly completes at time $t + s + e + h$, where e and h are the execution latency and cache hit latency, respectively. To minimize load-and-use latency, an instruction C that consumes the result of L must be considered for scheduling before knowing the hit/miss outcome of L. There are two scenarios where performance may be degraded due to misprediction: (i) If L is predicted as a *hit*, the scheduling of C can start as soon as $t + e + h$. However, if L turns out to be a *miss* but C was already scheduled, C and all its dependent instructions that were scheduled in the window of s cycles must be squashed and re-executed (note that this can mean more than s instructions in superscalar machines), as illustrated in Figure 5.19(b). (ii) If L is predicted as a *miss* when C is considered for scheduling, the scheduling of C will not start until the misprediction is discovered. Consequently, C and its dependent instructions are unnecessarily delayed by s cycles (see Figure 5.19(c)). To avoid both kinds of misprediction, we study six hardware-based cache miss predictors. Three of them—namely *self*, *global*, and *tournament*—are correlation-based.

The *self* predictor is shown in Figure 5.20(a), which contains a history table and a prediction table. The history table holds the last 12 cache hit/miss outcomes for up to 4096 loads, indexed by the instruction address. The 12-bit self history is used to select one of the 4096 two-bit prediction counters. A "HIT" prediction is made if the most significant bit of the prediction counter is one. All prediction counters are initialized to

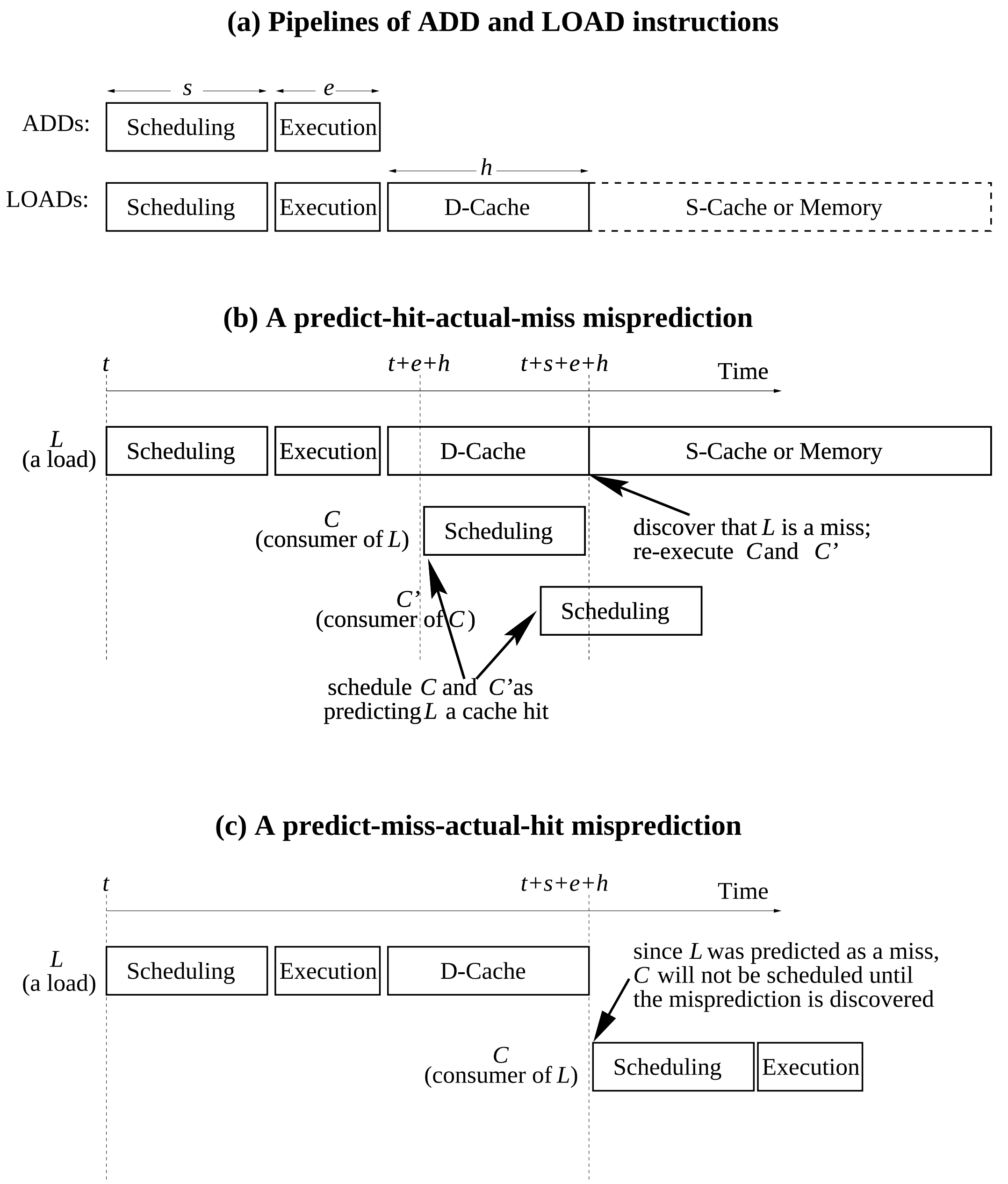


Figure 5.19: Illustration of the two kinds of misprediction penalties in dynamic instruction scheduling. Irrelevant pipeline stages such as fetch and decode are not shown in these figures.

two (i.e. just "HIT"). After a load graduates, the true cache outcome is inserted in the history table and the prediction counter is updated (using saturating arithmetic).

Figure 5.20(b) shows the *global* predictor, which is a 32K-entry table of two-bit prediction counters that is indexed by the global history of last 15 load hit/miss outcomes. The global history is updated when a load is issued speculatively but is backed up and

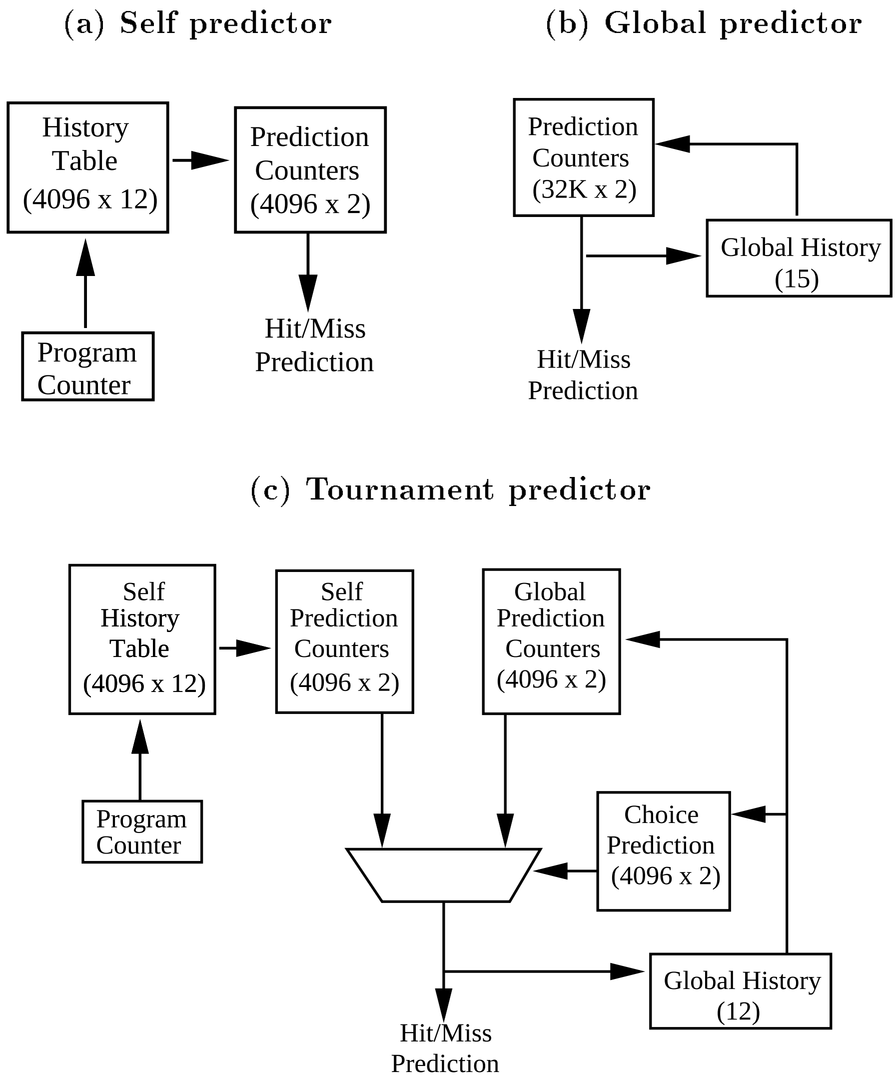


Figure 5.20: Three hardware correlation-based cache miss predictors.

corrected if the load is squashed eventually. The prediction counter is updated when the load graduates. Again, a "HIT" prediction is the most significant bit of the prediction counter, which is initialized to two.

The *tournament* predictor is in fact a combination of the self predictor and a smaller global predictor, as shown in Figure 5.20(c). It dynamically chooses between self and global history to predict the hit/miss outcome of a given load. The chooser table consists of 4096 two-bit saturation counters and is indexed by the global history. When the load graduates and if the self and global predictions differ, the selected choice entry is updated to support the correct predictor. The tournament predictor is included here because tournament *branch* predictors [88] have higher prediction accuracy than larger branch

Table 5.3: The seven cache miss predictors studied. The structures of the bimodal, self, global, and tournament are picked in a way that their resultant sizes are comparable.

Name	Structure	Size
Always-Hit	None	0
EV6	A 4-bit saturation counter	Four bits
Bimodal	32K 2-bit saturation counters indexed by the PC	8K bytes
Self	4K 12-bit self-history entries indexed by the PC, 4K 2-bit saturation counters indexed by self history	7K bytes
Global	A 15-bit global history register, 32K 2-bit saturation counters indexed by the global history	8K bytes
Tournament	4K 12-bit self-history entries indexed by the PC, 4K 2-bit self-prediction counters indexed by self history; a 12-bit global history register, 4K 2-bit global-prediction counters indexed by the global history; 4K 2-bit choice-prediction counters indexed by the global history	9K bytes
Ideal	None	0

Table 5.4: Simulation parameters of the two pipelines.

	Current Machines	Future Machines
Scheduling Latency	2 cycles	5 cycles
Cache Hit Latency	2 cycles	5 cycles
Issue Width	4 insts.	8 insts.
Functional Units	2 Int, 2 FP, 2 Memory, 2 Branch	4 Int, 4 FP, 4 Memory, 4 Branch
Reorder Buffer Size	64 entries	128 entries
Integer Multiply	12 cycles	
Integer Divide	76 cycles	
All Other Integer	1 cycle	
FP Divide	15 cycles	
FP Square Root	20 cycles	
All Other FP	2 cycles	
Branch Prediction Scheme	The tournament branch predictor in the Alpha 21264 [66]	

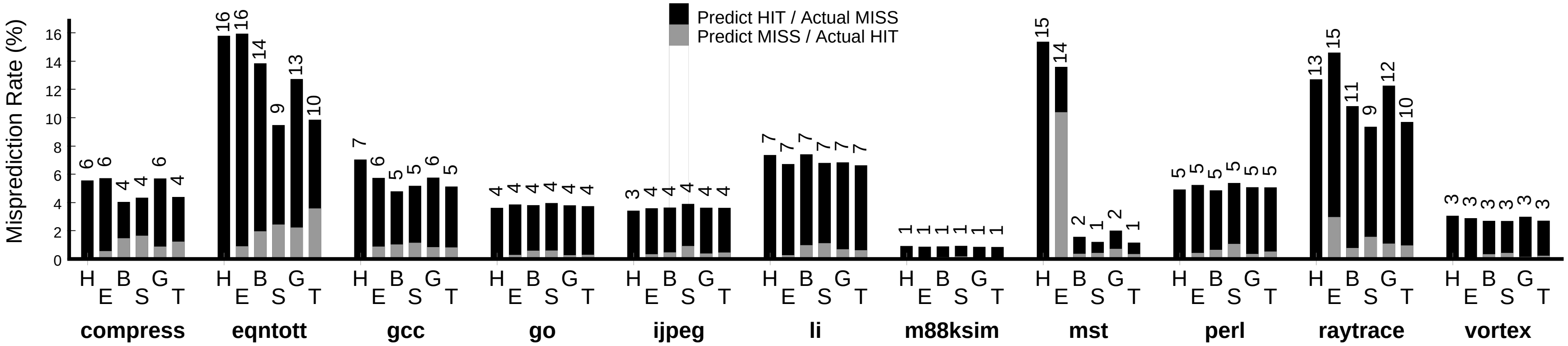
predictors that use either local or global history alone.

In addition to these three correlation-based predictors, we have four other predictors: *bimodal*, *EV6*, *always-hit*, and *ideal*. The *bimodal* predictor is an array of 32K two-bit saturation counters, indexed by the program counter. A "HIT" prediction is the most significant bit of the prediction counter, which is initialized to two (i.e. just "HIT"). It is updated when the load graduates. The *EV6* predictor is the one used by the Alpha 21264 (EV6) [66], which is the most significant bit of a 4-bit counter that tracks the hit/miss behavior of recent loads. The saturation counter is decremented by two upon a load miss and is incremented by one upon a hit. The *always-hit* predictor is the default predictor used in most processors—it simply assumes that all loads are cache hits. Finally, the *ideal* predictor always knows the hit/miss outcome in advance and is included for evaluating the absolute performance of realistic predictors. Table 5.3 summarizes these seven predictors with the amount of storage consumed by each.

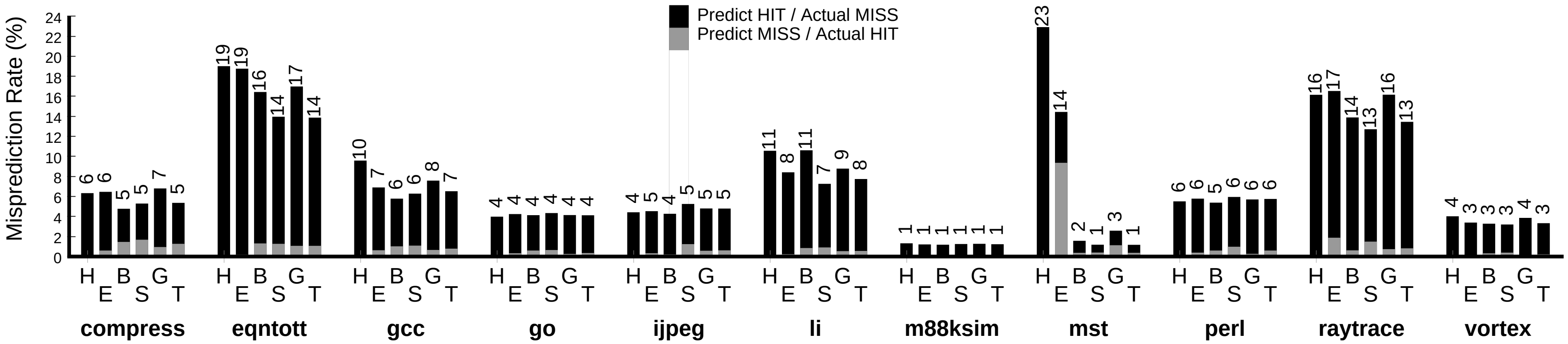
We experimented these predictors with two different pipelines: one is based on those of some recent processors such as the Alpha 21264 [66], whereas the other is a projected pipeline for some future processor. The key parameters are the scheduling latency and the cache hit latency (i.e. s and h in Figure 5.20, respectively). Table 5.4 lists the parameters of the two pipelines. The memory subsystem is identical to the one shown earlier in Table 3.5 of Chapter 3. Half of the applications studied in Section 5.4 were selected for this experiment. They are the eight SPEC95 integer benchmarks plus `eqntott`, `raytrace`, and `mst`.

Our first set of results is included in Figure 5.21 which shows the misprediction rate of the various cache miss predictors (the ideal predictor always has no misprediction and so is not shown). Like before, we divide mispredictions into two categories. We make the following observations from this figure. First, the overall misprediction rate in Figure 5.21 is much lower than the ones we observe in Figures 5.6 and 5.7. The difference is due to the fact that Figure 5.21 shows the misprediction rate for *all* loads executed, instead of focusing on loads that have intermediate miss ratios (i.e. correlation-profiled loads), as Figures 5.6 and 5.7 do. Second, among the predictors, the EV6 predictor is only slightly more accurate than the always-hit predictor (except in the case of `mst`). This is not a surprise given the very low hardware cost of the EV6 predictor. The bimodal and self predictors have similar prediction accuracy in many cases, but the self predictor achieves a significantly lower misprediction rate in `eqntott`, `li`, and `raytrace`. The global predictor does not predict as accurately as the self predictor. As a result, the tournament predictor mostly follows the self predictor. Third, the misprediction rate is generally higher in the future pipeline case. With a higher degree of speculation in the future pipeline, the data cache needs to accommodate a larger data set (including data accessed on wrong paths). Therefore the number of *Predict HIT/Actual MISS* mispredictions is increased along with the overall miss rate.

The improvement in prediction accuracy translates into the performance gain shown in Figure 5.22, where the instruction stall (*inst stall*) component has been reduced by cache miss prediction. As we expect, the performance potential is larger in the future pipeline case. For example, the ideal speedup of `eqntott` is 28% with the future pipeline but is only 18% with the current pipeline. The three correlation-based predictors achieve

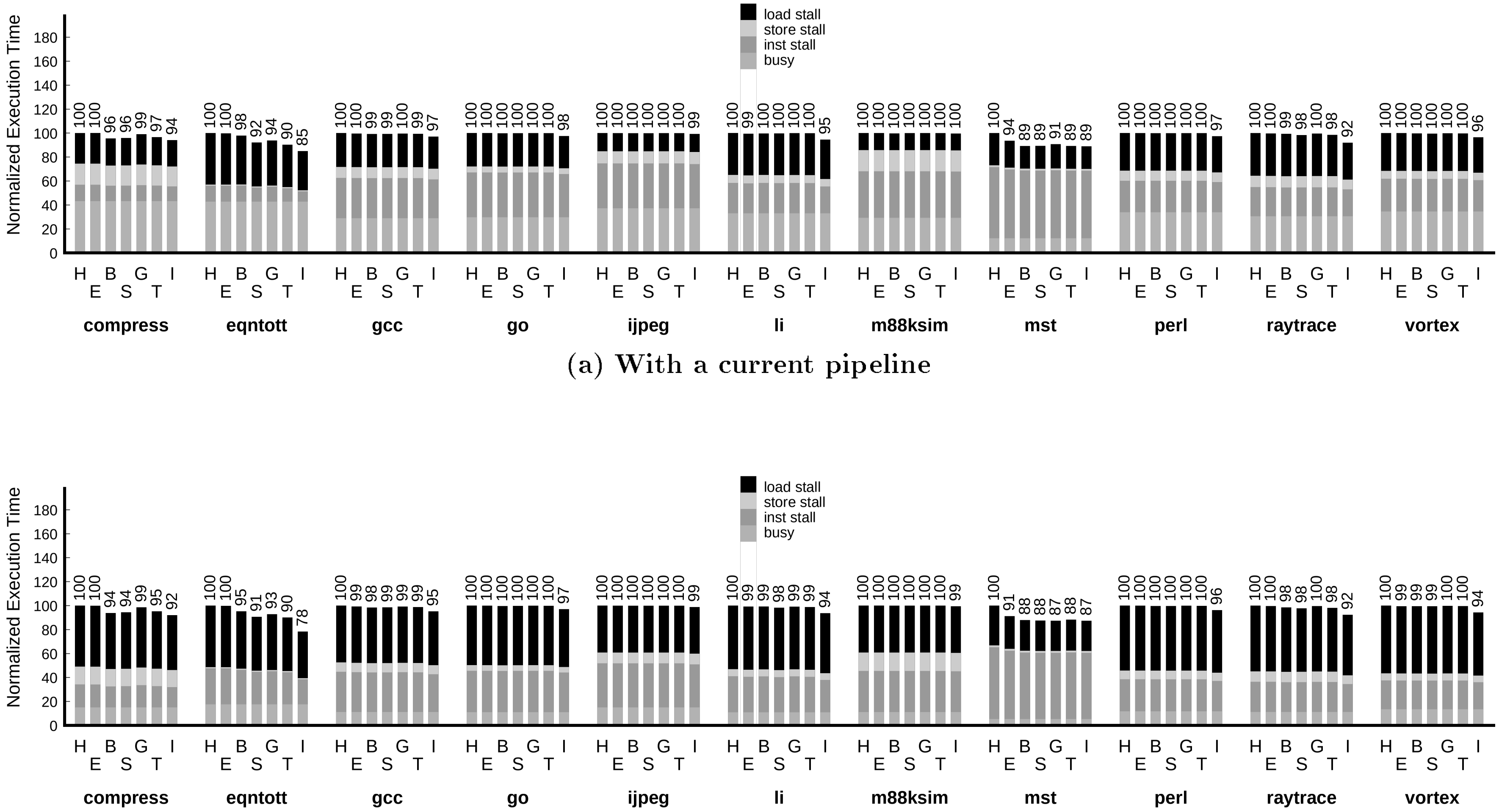


(a) With a current pipeline



(b) With a future pipeline

Figure 5.21: Misprediction rate of the various cache miss predictors used for dynamic instruction scheduling (**H** = always-hit, **E** = EV6, **B** = bimodal, **S** = self, **G** = global, **T** = tournament).



(a) With a current pipeline

(b) With a future pipeline

Figure 5.22: Performance of the various cache miss predictors used for dynamic instruction scheduling (**H** = always-hit, **E** = EV6, **B** = bimodal, **S** = self, **G** = global, **T** = tournament, **I** = ideal).

noticeable to significant speedups in seven of these 11 benchmarks. They range from 1% to 12% and from 1% to 15% for the current and future pipelines, respectively.

We would like to point out that our correlation-based predictors have not been optimized for performance—they are mainly for demonstration purpose and their design is largely based on existing branch predictors. Since there is still substantial room for improvement between correlation-based predictors and the ideal predictor in a few cases (especially for the future pipeline), it may be worthwhile to study more sophisticated correlation-based predictors (e.g., using more complex indexing mechanisms, adding tags to tables, or enlarging history or prediction tables, etc.) in order to achieve further performance gains.

5.6.2 Exploiting Correlation Profiling in Software

The first step of exploiting correlation profiling in software is to collect the correlation profiles themselves, which can be done either through simulation (as we have done in this study) or through some informative profiling techniques such as ProfileMe [39] and informing memory operations [53]. To minimize the overhead of correlation profiling, we found it useful to first use summary profiling information to focus only on the most significant loads with intermediate miss rates. With so few loads to profile (as evidenced by Table 5.2), it was reasonable to maintain relatively long sequences of basic block numbers or previous cache outcomes in the history patterns.

Assuming that we discover that interesting correlations do exist, the next step is to transform the code to statically isolate the contexts (either control-flow paths or different patterns in previous cache outcomes) which lead to the different cache outcome predictions. One viable approach for doing this is to exploit *code duplication*, which has been used to implement static correlated branch prediction [141]. We could also potentially duplicate sections of code to isolate contexts with similar cache outcome behavior, and apply individual latency tolerance actions to each context. Applying code duplication to separate control-flow paths is very similar to what has been done for static correlated branch prediction. Using code duplication to distinguish contexts which differ based on previous cache outcomes requires an architectural mechanism that enables software to observe cache outcomes directly, such as informing memory operations.

Exploiting Control-Flow Correlations

While we can reuse the code duplication algorithms developed for correlated static branch prediction [141], we may also potentially suffer from the same kinds of performance overhead due to *code expansion*. Fortunately, our results suggest that only a relatively

(a) Original procedure `foo()`

```
foo(tree* root){
  work(root→data);
  foo(root→left);
  foo(root→right);
}
```

$\stackrel{\text{clones}}{\Longrightarrow}$

(b) n clones of `foo()`

```
foo_0(tree* root){
  work(root→data);
  foo_1(root→left);
  foo_1(root→right);
}

foo_1(tree* root){
  work(root→data);
  foo_2(root→left);
  foo_2(root→right);
}

foo_2(tree* root){
  work(root→data);
  foo_3(root→left);
  foo_3(root→right);
}
⋮
foo_n(tree* root){
  work(root→data);
  foo_n(root→left);
  foo_n(root→right);
}
```

Figure 5.23: Example of using procedure cloning to determine the current tree level.

small amount of code expansion would be required in many applications, since the key distinctions are *call sites* involved in the calling chain. By cloning a small number of procedures, we can effectively isolate these paths.

We can also potentially use *procedure cloning* [34] to determine the current tree level for predicting tree-level dependent cache outcomes. Figure 5.23 illustrates how a recursive procedure `foo()` could be cloned n times, where each copy corresponds to a particular depth in the tree. If most misses occur near the root of the tree (as we have observed in several of our cases studies), we could apply a latency tolerance scheme at higher levels in the tree (`foo_0()`, `foo_1()`, etc.), but disable it near the bottom (`foo_n()`). The number of clones that we should make can potentially be determined from the control-flow correlation profiling information.

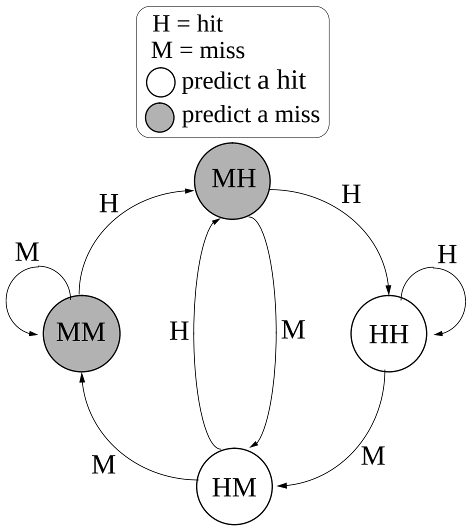
Exploiting Self Correlations or Global Correlations

A viable technique for exploiting self correlations or global correlations is to use *informing memory operations* [53], which are essentially memory operations, each combined with a conditional branch-and-link operation that is taken only if the reference suffers a cache miss. With informing memory operations, we can apply code duplication to separate contexts based on previous cache outcomes. Figure 5.24 shows an example of how this might be accomplished. Assume that self correlation has detected three common cache outcome patterns for the load of `*p` in Figure 5.24(a): (i) a long sequence of hits, (ii) a long sequence of misses, and (iii) an alternating sequence of hits and misses. Given these patterns, we could use the previous two cache outcomes to predict the next cache outcome, as illustrated by the state diagram in Figure 5.24(b). Figure 5.24(c) shows how

(a) Original code

```
loop_begin: if (p == 0) goto loop_exit
                    :
            R <- load *p
            R <- R + 10
                    :
            goto loop_begin
loop_exit:
```

(b) Predicting the next cache outcome based on the last two



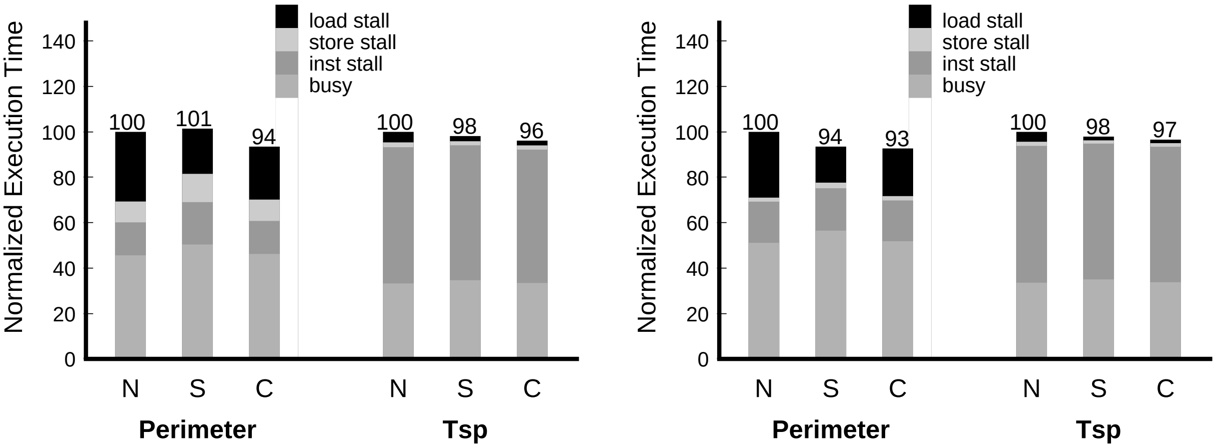
(c) Duplicated code with informing memory operations

```
HH_loop_begin: if (p == 0) goto loop_exit
                      :
               R <- info_load *p
               R <- R + 10      /*just seen HH*/
jump upon a miss      :
               goto HH_loop_begin
               R <- R + 10      /*just seen HM*/
                      :
HM_loop_begin: if (p == 0) goto loop_exit
                      :
               R <- info_load *p
               R <- R + 10      /*just seen MH*/
jump upon a miss      :
               goto MH_loop_begin
               R <- R + 10      /*just seen MM*/
                      :
MM_loop_begin: if (p == 0) goto loop_exit
                      :
               R <- info_load *p
               R <- R + 10      /*just seen MH*/
jump upon a miss      :
               goto MH_loop_begin
               R <- R + 10      /*just seen MM*/
                      :
               goto MM_loop_begin
MH_loop_begin: if (p == 0) goto loop_exit
                      :
               R <- info_load *p
               R <- R + 10      /*just seen HH*/
jump upon a miss      :
               goto HH_loop_begin
               R <- R + 10      /*just seen HM*/
                      :
               goto HM_loop_begin
loop_exit:
```

Figure 5.24: Example of using informing memory operations to implement self correlated prediction. The single loop in (a) is duplicated into the four loops shown in (c), each of them corresponds to a different state in (b).

the original loop could be replicated into four copies, where each copy encodes one of the four states. The original load of `*p` is replaced by an *informing* version of the same load. As a result of each informing load, one of two state transitions occur: (i) if the load hits, we will continue as normal and jump directly to next state associated with a hit; (ii) if the load *misses*, the informing mechanism will directly trigger a branch which we will set to take us to the next state associated with a miss.

Again, code expansion is a concern here. In the worst case, we might expand the code exponentially with respect to the length of the history pattern that is needed to provide the correlation. Fortunately, our experimental results indicate that relatively



(a) **With one memory unit** (b) **With two memory units**

Figure 5.25: Impact of correlation profiling on prefetching performance (**N** = no prefetching, **S** = prefetching directed by summary profiling, **C** = prefetching directed by correlation profiling).

short history lengths (four or fewer) are sufficient in many applications. In addition, we do not expect it to be necessary to enumerate all possible contexts of a particular history length, as we do in our example. In many situations, we only need to differentiate contexts that occur frequently but have substantially different miss ratios. Enumerating only these contexts can reduce the code expansion factor significantly.

Applying Correlation Profiling to Software-Controlled Prefetching

To exemplify how correlation profiling can be exploited in software, we used both summary and correlation profiling to guide the manual insertion of prefetch instructions into two applications: (`perimeter` and `tsp`). In the case of correlation profiling, we used procedure cloning to isolate different dynamic instances of a static reference, and adapted the prefetching strategy accordingly with respect to the call sites. Two processor models, each with one or two memory functional units, were used in this experiment; other simulation parameters are identical to those used in Chapter 3 (refer back to Table 3.5 for the details). As we discussed in Section 3.5.6, changing the number of memory units alters the effective prefetching overhead on superscalar machines. Doing so allows us to experiment with different degrees of overhead without modifying the prefetching code. The input data sets are the "testing" sets shown in Table 5.1.

Figure 5.25 shows the resultant execution time, normalized to the case without prefetching. Note that the execution time of correlation-profiling directed prefetching has already included any possible overheads due to procedure cloning such as additional instruction cache misses and branching, etc. As we observe from the figure, correlation profiling improves the performance of prefetching over summary profiling by isolating dynamic hits and misses more effectively, thereby achieving similar benefits with significantly less overhead. This is especially the case with one memory unit where prefetching overhead has a bigger impact on performance. For example, for `perimeter`, correlation profiling speeds up prefetching by 7% over summary profiling with one memory unit but by only 1% with two memory units. Overall, these results demonstrate that software can exploit correlation profiling through code duplication to further increase the benefit of latency tolerance.

5.7 Chapter Summary

To achieve the full potential of latency tolerance techniques, we have proposed *correlation profiling*, which is a technique for isolating which dynamic instances of a static memory reference are likely to suffer cache misses. We have evaluated the potential performance benefits of three different forms of correlation profiling on a wide variety of non-numeric applications. Our experiments demonstrate that correlation profiling techniques outperform summary profiling in most cases by increasing the degree of bias in the miss ratio distribution, and this improved prediction accuracy can translate into significant reductions in the memory stall time for roughly half of the applications we study. We also find that correlation profiling is fairly robust with respect to input data sets. Detailed case studies of individual applications show that *self correlation* works well because the cache outcome patterns of individual references often repeat in predictable ways, and that *control-flow correlation* works mainly because many cache outcomes are call-chain dependent. Although *global correlation* offers superior performance in some cases, for the most part it mainly assimilates self correlation. Finally, we propose exploitation of correlation profiling based on well-known history-based branch prediction techniques. We demonstrate that hardware correlation-based predictors improve the effectiveness of dynamic instruction scheduling. We also suggest that code duplication techniques could be used to exploit correlation profiling in software and observe that correlation profiling offers superior performance over summary profiling when prefetching on a superscalar processor.

Chapter 6

Conclusions

The memory latency of accessing *instructions* and *data* has become a critical performance bottleneck in many important computer applications. Caches are a crucial first step toward addressing this problem, but they are not a panacea. As a result, a number of latency-hiding techniques have been proposed. Among them, *locality optimizations* and *prefetching* have demonstrated significant success in regular numeric codes but their potential in irregular *non-numeric applications* has been largely unexplored. This dissertation attempts to provide viable techniques that help unlock these techniques' potential, thereby improving the memory performance of non-numeric codes.

The techniques investigated and the key results of this dissertation are summarized as follows:

Cooperative Instruction Prefetching: We began with a study on using prefetching to hide the latency of fetching *instructions*. Our detailed evaluation of previous instruction prefetching schemes reveals that only about one-third of the instruction cache miss latency in a modern processor can be hidden by these schemes because prefetches are usually launched too late. To address this problem, we proposed *cooperative instruction prefetching*, which uses the compiler to automatically prefetch non-sequential accesses far enough in advance while using the hardware to prefetch sequential accesses aggressively under the guidance of a prefetch filter. The results of this cooperation are that, first, over 70% of the miss latency is successfully hidden by converting late prefetches into cache hits and that, second, the number of useless prefetches is six times less than the best previous scheme. Moreover, the compiler is able to schedule prefetches early enough while at the same time limiting code expansion to only 9% of the original program. We also demonstrated that the effectiveness of cooperative prefetching can be further increased by using profiling information to help reduce conflict misses and unnecessary prefetches. From an architectural perspective, the performance advantages of cooperative prefetching are sustained over a range of common miss latencies and bandwidth. Finally, cooperative prefetching is cost-effective as well, since it delivers performance comparable to (or even better than) that of larger caches, but requires a much smaller hardware budget.

Compiler-Based Prefetching for Recursive Data Structures: We continued by exploiting the potential of *data* prefetching in an important class of non-numeric codes that contain *pointer-based data structures*. To overcome the *pointer-chasing problem* associated with prefetching these data structures, we designed three prefetching schemes and automated them in an optimizing research compiler. Detailed evaluation of these schemes on a dynamically-scheduled processor demonstrates that compiler-inserted prefetching can significantly improve the execution speed of pointer-based codes—up to more than twofold for those applications that suffer the most from cache misses. While *greedy prefetching* is the most widely applicable scheme, the other two schemes can provide superior performance when the traversal orders of the data structures are predictable. In a few cases where prefetching overhead outweighs the benefit, the performance degradation is slight (usually less than 2%). Our results also indicate that these prefetching schemes perform well over wide ranges of miss latencies, memory bandwidth, and cache sizes. In addition, through increasing the prefetching distance, *history-pointer prefetching* and *data-linearization prefetching* can cope with very large miss latencies. Compared with the only other compiler-based pointer prefetching scheme found in the literature, our schemes offer substantially better performance by avoiding unnecessary overhead and hiding more latency.

Memory Forwarding: In addition to *tolerating* latency through prefetching, *reducing* latency by re-optimizing data layout at run-time provides another rich opportunity to improve memory performance. Unfortunately, it is extremely difficult to guarantee that such optimizations are safe in practice on today's machines, since accurately updating all pointers to an object requires perfect alias information, which is well beyond the scope of the compiler for languages such as C. To overcome this limitation, we proposed a technique called *memory forwarding* which effectively adds a new layer of indirection within the memory system whenever necessary to guarantee that data relocation is safe. Because actual forwarding rarely occurs (it works as a safety net), the mechanism can be implemented as an exception in modern superscalar processors. Our experimental results demonstrate that the aggressive layout optimizations enabled by memory forwarding can result in significant speedups—more than twofold in a few cases—by reducing the number of cache misses, improving the effectiveness of prefetching, and conserving memory bandwidth.

Correlation-Based Cache Miss Prediction: Finally, we investigated cache miss prediction in non-numeric codes. An important use of such prediction is to reduce the unnecessary overhead of prefetching and other latency tolerance techniques by applying them only to the dynamic references that are likely to suffer cache misses. We proposed a new class of prediction techniques that *correlate* future cache misses with recent control-flow paths or with recent cache miss behavior. Our experimental results show that roughly half of the 21 non-numeric applications we study can potentially enjoy significant reductions in memory stall time by exploiting at least one of the three forms of correlation we consider: *control-flow correlation*, *self correlation*, and *global correlation*. In addition, our detailed case studies illustrate that self correlation succeeds because a given reference's cache outcomes often contain repeated patterns, and control-flow correlation succeeds because cache outcomes are often call-chain dependent. Lastly, we demonstrated that correlation-based miss prediction can boost the performance of dynamic instruction scheduling and software-controlled prefetching.

6.1 Future Work

The techniques developed in this research target non-numeric codes in general. Nonetheless, most of the workloads we experimented with were drawn from the technical community. Perhaps the most obvious next step is to extend this research to directly address *commercial* workloads. Although there are already a number of studies on measuring the performance of such workloads, studies on actually improving their performance are rare. We are encouraged by some preliminary research [64] which extends our data prefetching techniques to handle more commercially-oriented applications, and we believe that a thorough study on the applicability of our techniques to these applications is warranted.

We are also interested in further exploring the design space of memory forwarding. In particular, we hope to eliminate the need for *physical* forwarding bits to distinguish forwarded data. By using some combination of data encoding and exception handling, we might be able to emulate these forwarding bits mostly in *software*, and thereby implement memory forwarding without changing the memory hierarchy. In addition, compiler techniques that utilize memory forwarding are desirable. Ultimately, we want to automate dynamic data layout optimizations (for common data structures) in the compiler. With correctness guaranteed by memory forwarding, the next problem will be how to accurately estimate the cost/benefit tradeoff of these optimizations. The use of dynamic information through profiling feedback or adaptive code appears to be essential for addressing this problem.

So far, we have been focusing on optimizing *existing* programs. Another research direction is to develop tools that can help programmers write new programs that are "memory friendly". To maximize performance, we may need to attack the memory latency problem at an *algorithmic* level. That is, memory latency should be taken into account when we estimate the running time of an algorithm. For example, there have been some studies on designing searching and sorting algorithms with the consideration of memory latency [72]. However, this approach requires programmers to fully understand the memory hierarchy, which is highly unlikely in general, since the memory hierarchy is increasingly complex and is also machine-dependent. An intermediate solution between this "programmer-centric" extreme and the "compiler-centric" extreme is to develop libraries of *intelligent, memory-conscious data structures*. These structures would collect run-time information about memory behavior. Based on this information, they can decide themselves which implementation of a data structure should be used, how often data relocation should happen, and which prefetching strategy should be taken. We envision that such libraries could be implemented for a wide range of languages, including C, C++, and Java. In addition, these new libraries can share the same interfaces to some "standard" libraries of these languages (e.g., the Standard Template Library (STL) [124] for C++, the `java.util` package [23] for Java) so that programmers would have less difficulty in using them.

Finally, the techniques that we developed for prefetching, improving locality, and predicting cache misses can potentially be extended to cope with other forms of latency. Important candidates include accessing file systems and communicating across networks, including the World-Wide Web.

Appendix A

Overall Experimental Methodology

In this appendix, we discuss the overall experimental methodology used throughout this dissertation. Since each of the techniques we studied addresses different aspects of the problem (of improving the cache performance of non-numeric codes), different experimental setups may be required to evaluate different techniques. Therefore, the detailed setups are reported individually along with the corresponding technique in each chapter. Here, we instead focus on the rationales for configuring our experiments—in particular, those used for running simulations and selecting benchmarks.

A.1 Simulations

Since some of our proposed techniques involve new hardware features that do not exist in any real processors, using simulation is the most viable approach to performing quantitative evaluation of these techniques. In addition, using simulation has two advantages over running on real machines: (1) Through simulations we can obtain extremely detailed or tailored statistics that are not available on real machines; (2) Simulations allow us to experiment with a range of hardware configurations. However, the disadvantages are that simulations are several orders of magnitude slower than the actual execution and that simulation results need to be validated. We will shortly discuss how these two issues were addressed in our experiments. Let us first briefly describe our simulator.

Figure A.1 shows the organization of our simulation system. Our simulator can be driven by either the *pixie* [119] or *mable* [38] traces generated on MIPS machines. These traces list all of the instruction and data addresses in the order that they appeared in the actual execution.

Our simulator performs detailed cycle-by-cycle simulations of a generic dynamically-scheduled, superscalar processor with the structure shown in Figure A.2. It models the rich details of the processor including the pipeline, register renaming, the reorder buffer,

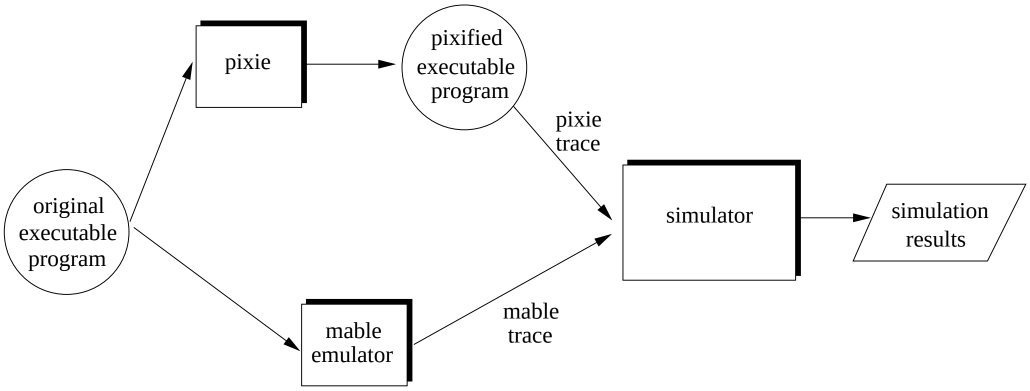


Figure A.1: Organization of our simulation system. Each simulation run is driven by either a *pixie* or a *mable* trace (but not both).

branch prediction, branching penalties, speculative instruction fetching (including incorrect execution paths), the memory hierarchy (including tag, bank, and bus contention), etc.

To increase simulation speed, the simulator uses an *event-driven* approach, where only the actions that really need to be performed in a particular simulation cycle are considered. This is more efficient than the polling-based approach which considers *all* possible actions in each cycle to determine what need to be performed.

Three approaches were used together to validate simulation results. First, to ensure that the simulator correctly implemented the desired functionality, an extensive number of assertions were added to the simulator source code to detect unexpected events and conditions. Second, different parts of the simulation statistics were checked against each other to make sure that they were consistent. Third, to know whether our simulator correctly predicted the run-time behavior of an application running on an equivalent real processor, simulation results were compared to those reported by other researchers using their own simulators. We found that the discrepancies between our results and theirs were small.

A.2 Benchmarks

The benchmarks used in this dissertation were all non-numeric applications which were either publicly available or drawn from some common benchmark suites. The selection of the benchmarks for evaluating a particular technique was based on their relevance to the performance characteristic being investigated. For example, we focused only on

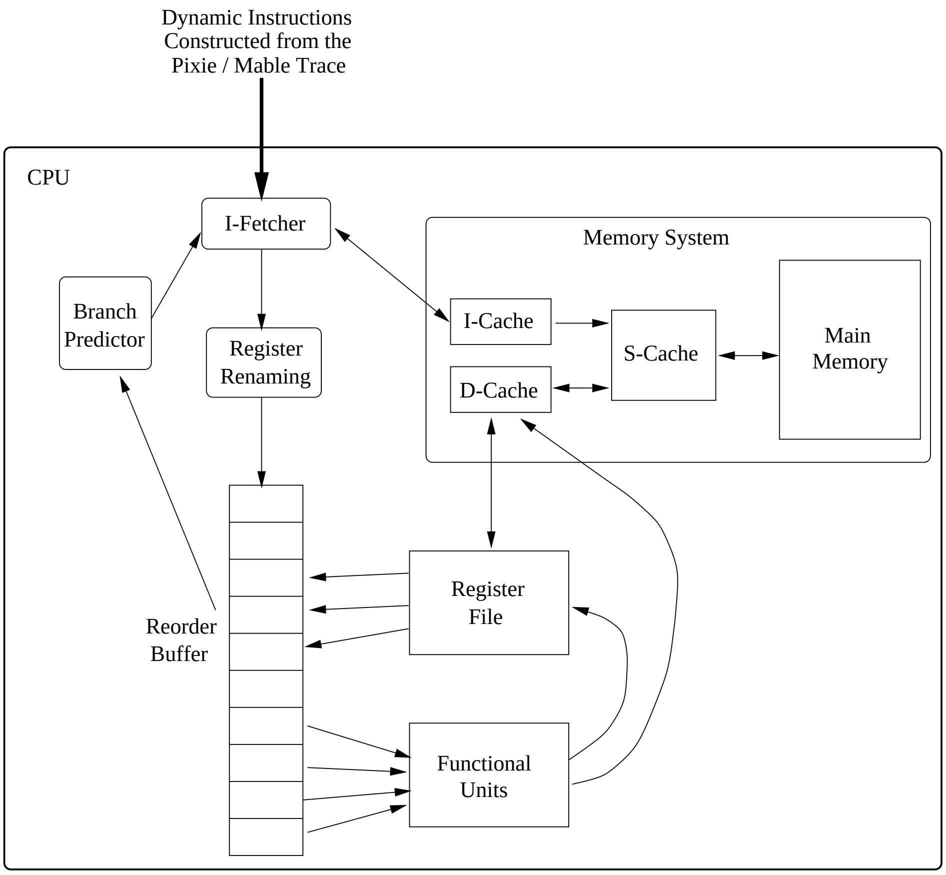


Figure A.2: Structure of the dynamically-scheduled, superscalar processor being simulated.

applications that have large instruction footprints in the instruction prefetching study. Details of the benchmarks used for each technique are provided in the corresponding chapter. Below, we give an overview of our benchmarks.

Similar to other studies, we used the "standard" benchmarks provided by SPEC. These included the entire SPEC95int suite [36] and three applications (*eqntott*, *espresso*, *sc*) from the SPEC92int suite [35]. Nevertheless, SPEC benchmarks alone might not be sufficient for our purpose since some studies already revealed that the cache performance of many SPEC benchmarks was not as problematic as that of many important non-numeric programs [26]. Therefore, we used additional benchmarks including all of the ten applications from the Olden benchmark suite [109], a uniprocessor version of a SPLASH-2 benchmark [135] (*raytrace*), and eight applications from the public domain (*awk* [5], *porky* [133], *postgres* [142], *radiosity* [90], *skweel* [133], *smv* [89], *tcl* [102], and *vis* [15]). Altogether, our benchmark collection covered a wide range of applications including database servers, graphics programs, hardware verification systems, interpreters, compilers, simulators, etc. All of these applications were written in C except that two were in C++. Their size ranged from several hundreds to over a hundred thousand source lines. All of them were run to completion in our experiments.

The input data for our experiments was chosen from those provided along with the benchmarks or those that had been used to produce some previously published results. When there were more than one such data sets, we attempted to pick the one that resulted in the longest execution time. Nevertheless, to make our experiments feasible, we only used input that took less than one week to simulate (which took less than three minutes or so in the actual execution).

Bibliography

[1] S. G. Abraham and B. R. Rau. Predicting load latencies using cache profiling. Technical Report HPL-94-110, Hewlett-Packard Company, November 1994.

[2] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 139–152, December 1993.

[3] A. Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[4] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[5] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison Wesley, 1988.

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

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

[8] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 97 Conference on Programming Language Design and Implementation*, June 1997.

[9] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[10] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 176–186, 1991.

[11] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[12] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[13] D. Bernstein, D. Cohen, A. Freund, and D. E. Maydan. Compiler techniques for data prefetching on the PowerPC. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, pages 19–26, June 1995.

[14] D. G. Bobrow and D. W. Clark. Compact encodings of list structure. *ACM Transactions. on Programming Languages and Systems*, 1(2):267–286, October 1979.

[15] R. K. Brayton, G. D. Hachtel, A. S. Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. R. Shilpe, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification*, July 1996.

[16] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[17] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 244–255, October 1996.

[18] D. Burger, J. R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[19] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[20] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[21] M. C. Carlisle and Anne Rogers. Software caching and computation migration in olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 29–38, July 1995.

[22] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, October 1994.

[23] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries: Second Edition, Volume 1 Supplement for the Java 2 Platform, Standard Edition, v1.2.* Addison-Wesley, June 1999.

[24] P. Chang, E. Hao, T. Yeh, and Y. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, November 1994.

[25] P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Transactions on Computers*, 44(4):481–494, April 1995.

[26] M. J. Charney and T. R. Puzak. Prefetching and memory system behavior of the SPEC95 benchmark suite. *IBM Journal of Research and Development*, 41(3):265–285, May 1997.

[27] T.-F. Chen. *Data Prefetching for High-Performance Processors*. PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, July 1993.

[28] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5), May 1995.

[29] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69–73, 1991.

[30] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, Nov 1970.

[31] T. M. Chilimbi, J. R. Larus, and M. D. Hill. Tools for cache-conscious data structures. In *Proceedings of the ACM SIGPLAN 99 Conference on Programming Language Design and Implementation*, May 1999.

[32] G. Chrysos and J. Emer. Memory dependency prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 142–153, June 1998.

[33] D. W. Clark. *List structure: measurements, algorithms, and encodings*. PhD thesis, Carnegie-Mellon University, August 1976.

[34] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2), April 1993.

[35] Standard Performance Evaluation Corporation. *The SPEC92 benchmark suite*. http://www.specbench.org.

[36] Standard Performance Evaluation Corporation. *The SPEC95 benchmark suite*. http://www.specbench.org.

[37] Z. Cvetanovic and D. Bhandarkar. Performance characterization of the Alpha 21164 microprocessor using TP and SPEC workloads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 60–70, April 1994.

[38] P. Davies, P. Lacroute, J. Heinlein, and M. Horowitz. Mable: A technique for efficient machine simulation. Technical Report CSL-TR-94-636, Stanford University, September 1994.

[39] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 292–302, December 1997.

[40] A. Deutsch. A storeless model of aliasing and its abstractions using finite representation of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13, April 1992.

[41] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.

[42] R. Ghiya and L. J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, January 1996.

[43] N. Gloy, T. Blackwell, M. Smith, and B. Calder. Procedure placement using temporal ordering information. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 303–313, December 1997.

[44] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.

[45] E. H. Gornish. Compile time analysis for data prefetching. Master's thesis, University of Illinois at Urbana-Champaign, December 1989.

[46] R. Greenblatt. The LISP Machine. Technical Report Working Paper 79, M.I.T. Artificial Intelligence Laboratory, November 1974.

[47] L. Gwennap. Alpha 21364 to ease memory bottleneck. *Microprocessor Report*, 12(14):12–15, 1998.

[48] L. Gwennap. Intel outlines high-end roadmap. *Microprocessor Report*, 12(14):16–19, 1998.

[49] R. H. Halstead, Jr. and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, June 1988.

[50] W. J. Hansen. Compact list representation: Definition, garbage collection, and system implementation. *Communications of the ACM*, 12(9):499–507, September 1969.

[51] L. J. Hendren, J. Hummel, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, pages 218–229, June 1994.

[52] Hewlett Packard. *HP PA-8500 Microprocessor White Paper*, 1998.

[53] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing Memory Operations: Memory Performance Feedback Mechanisms and Their Applications. *ACM Transactions on Computer Systems*, 16(2):170–205, May 1998.

[54] A.S. Huang and J. P. Shen. A limit study of memory requirements using value reuse profiles. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 71–81, Dec 1995.

[55] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *IEEE CompCon'95*, March 1995.

[56] W. Hwu and P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 242–251, May 1989.

[57] IBM. *PowerPC 620 Risc Microprocessor Technical Summary*, October 1994.

[58] T.E. Jeremiassen and S.J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *7th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, July 1995.

[59] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-time spatial locality detection and optimization. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 57–64, December 1997.

[60] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[61] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[62] D. Kaeli and P. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 34–42, May 1991.

[63] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[64] M. Karlsson, F. Dahlgren, and P. Stenström. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, January 2000.

[65] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4), Nov 1992.

[66] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the International Conference on Computer Design*, October 1998.

[67] A. C. Klaiber and H. M. Levy. Architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–63, May 1991.

[68] J. S. Kowalik, editor. *Parallel MIMD Computation : The HEP Supercomputer and Its Applications*. MIT Press, 1985.

[69] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–85, 1981.

[70] H. Kwak, B. Lee, A. R. Hurson, S.-H. Yoon, and W.-J. Hahn. Effects of multithreading on cache performance. *IEEE Transactions on Computers*, 48(2):176–184, February 1999.

[71] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[72] A. G. LaMarca. *Caches and Algorithms*. PhD thesis, University of Washington, 1996.

[73] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, October 1994.

[74] D. Leibholz and R. Razdan. The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor. In *IEEE CompCon'97*, February 1997.

[75] G. Lesartre and D. Hunt. PA-8500: The continuing evolution of the PA-8000 family. In *IEEE CompCon'97*, 1997.

[76] K. Li and P. Hudak. A new list compaction method. *Software - Practice and Experience*, 16(2):145–163, February 1986.

[77] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 231–236, 1995.

[78] P. G. Lowney, S. Freudenberger, T. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multoflow trace scheduling compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[79] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[80] C.-K. Luk and T. C. Mowry. Compiler and hardware support for automatic instruction prefetching: A cooperative approach. Technical Report CMU-CS-98-140, Carnegie Mellon University, June 1998.

[81] C.-K. Luk and T. C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 182–193, December 1998.

[82] C.-K. Luk and T. C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers (Special Issue on Cache Memory)*, 48(2):134–141, February 1999.

[83] C.-K. Luk and T. C. Mowry. Memory forwarding: Enabling aggressive data layout optimizations by guaranteeing the safety of data relocation. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 88–99, May 1999.

[84] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 45–54, December 1992.

[85] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[86] A. Maynard, C. Donnelly, and B. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, October 1994.

[87] S. McFarling. Program optimization for instruction caches. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[88] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[89] K. L. McMillan. *The SMV system.* Carnegie-Mellon University, February 1992.

[90] D. Meneveaux, K. Bouatouch, and E. Maisel. Memory management schemes for radiosity computation in complex environment. Technical Report PI 1097, IRISA/INRIA, 1996.

[91] M. L. Minsky. A Lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, M.I.T., Cambridge, Mass., 1963.

[92] D. A. Moon. Architecture of the symbolics 3600. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 76–83, 1985.

[93] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[94] A. Moshovos and G. Sohi. Streamlining inter-operation memory communication. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 235–245, December 1997.

[95] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching.* PhD thesis, Stanford University, March 1994.

[96] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on*

Architectural Support for Programming Languages and Operating Systems, pages 62–73, October 1992.

[97] T. C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 314–320, December 1997.

[98] T. C. Mowry and C.-K. Luk. Understanding why correlation profiling improves the predictability of data cache misses in non-numeric applications. *IEEE Transactions on Computers*, to appear.

[99] T. C. Mowry and S. R. Ramkissoon. Software-controlled multithreading using informing memory operations. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, January 2000.

[100] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 15–23, 1995.

[101] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the International Conference on Software Engineering*, pages 338–348, 1997.

[102] John Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[103] S. Pan, K. So, and J. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, October 1992.

[104] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM: IRAM. *IEEE Micro*, April 1997.

[105] K. Pettis and R. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[106] J. Pierce and T. Mudge. Wrong-path prefetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[107] A. R. Pleszkun and M. J. Thazhuthaveetil. An architecture for efficient Lisp list access. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 191–198, 1986.

[108] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668, June 1990.

[109] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions. on Programming Languages and Systems*, 17(2):233–263, March 1995.

[110] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.

[111] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 111–121, May 1999.

[112] V. Santhanam, E. Gornish, and W.-C. Hsu. Data prefetching on the HP PA8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, June 1997.

[113] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: the case for processor/memory integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 90–101, May 1996.

[114] D. P. Siewiorek, C. G. Bell, and A. Newell, editors. *Computer Structures: Principles and Examples*, chapter Design of the B 5000 System, pages 129–134. McGraw-Hill, New York, 1982.

[115] A. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(2):7–21, 1978.

[116] A. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, Sept. 1982.

[117] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 298:241–248, 1981.

[118] J. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Supercomputing'92*, pages 588–597, 1992.

[119] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[120] G. S. Sohi, E. S. Davidson, and J. H. Patel. An efficient Lisp-execution architecture with a new representation for list structures. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 91–98, 1985.

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

[122] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *In Proceedings of the Sixth International Conference on Compiler Construction*, pages 136–150, April 1996. Also available as LNCS 1060.

[123] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM SIG PLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[124] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[125] C. J. Stephenson. Fast fits. In *Proceedings of the ACM 9th Symposium on Operating Systems*, pages 30–32, October 1983.

[126] G. S. Taylor, P. N. Hilfinger, J. R. Larus, D. A. Patterson, and B. G. Zorn. Evaluation of the SPUR Lisp architecture. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 444–452, 1986.

[127] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing'93*, pages 410–419, November 1993.

[128] J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[129] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the 1st International Symposium on High Performance Computer Architecture*, pages 360–369, January 1995.

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

[131] G. Tyson and T. Austin. Improving the accuracy and performance of memory communication. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 218–227, December 1997.

[132] M. V. Wilkes. The memory wall and the CMOS end-point. *ACM Computer Architecture News*, 23(4), 1995.

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

[134] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[135] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–38, June 1995.

[136] Wm. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture News*, 23(1), 1995.

[137] C. Xia and J. Torrellas. Instruction prefetching of system codes with layout optimized for reduced cache misses. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 271–282, June 1996.

[138] Y. Yamada, J. Gyllenhaal, G. Hand, and W-M Hwu. Data relocation and prefetching for programs with large data sets. In *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 217–227, 1994.

[139] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, pages 28–41, April 1996.

[140] T.-Y. Yeh and Y. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993.

[141] C. Young and M. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–241, October 1994.

[142] A. Yu and J. Chen. *The Postgres95 User Manual v1.0.* University of California at Berkeley, Sept 1996.

[143] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 188–200, June 1995.