COOPERATIVE HARDWARE/SOFTWARE CACHING FOR NEXT-GENERATION MEMORY SYSTEMS

A Dissertation Presented

by

ZHENLIN WANG

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2004

Department of Computer Science

COOPERATIVE HARDWARE/SOFTWARE CACHING FOR NEXT-GENERATION MEMORY SYSTEMS

A Dissertation Presented

by

ZHENLIN WANG

Approved as to style and content by:

Kathryn S. McKinley, Chair

Charles C. Weems, Member

J. Eliot B. Moss, Member

Csaba Andras Moritz, Member

Doug Burger, Member

W. Bruce Croft, Department Chair
Department of Computer Science

To my parents, my wife, and my daughter

ACKNOWLEDGMENTS

I am deeply indebted to my advisor, Kathryn McKinley, for her research guidance as well as moral support. She kindly adopted me when I joined UMass six years ago and could barely express myself in English. She has been an outstanding advisor and I have been sufficiently guided even when we only had remote contact for three years.

I must thank Eliot Moss and Chip Weems for their leadership in ALI group. They have been providing us an enjoyable research environment. A special thank goes to Chip who has been acting as my local advisor when Kathryn was a thousand miles away. Both Eliot and Chip served as members of my committee. I thank them for their thoughtful suggestions and comments. I thank Eliot for his careful reading of every detail of my dissertation.

I'd like to thank all members of ALI group, particularly James Burrill, Brendon Cahoon, Steve Blackburn, Xianglong Huang, Steve Dropso, Chris Hoffmann, Matthew Hertz, and John Cavazos, whom I always come to for help on both technical issues and English usage. It is my pleasure to work with such a wonderful team. Also I owe thanks to Benyuan Liu, Wei Wei, and Tian Bu whom I went downstairs to talk with when I was bored by research.

I enjoyed doing research with Doug Burger, Steve Reinhardt, and Csaba Andras Moritz. Their insights in system research inspired me and will benefit my career in the long run. As my committee members, Doug and Andras also suggested quite a few technical updates indispensable to my thesis.

I am grateful to Zhuoqun Xu who was my master thesis advisor in China. I built a solid background in computer science during my work with him in Beijing University.

I thank my parents who had always encouraged me to move on for a higher degree. I cannot imagine a life without the big, warm family they had created. I must thank my

wife, Ruihong, and my daughter, Maggie. I could not have finished my thesis without their support, love, and forbearance.

ABSTRACT

COOPERATIVE HARDWARE/SOFTWARE CACHING FOR NEXT-GENERATION MEMORY SYSTEMS

FEBRUARY 2004

ZHENLIN WANG

B.S., BEIJING UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS, AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Kathryn S. McKinley

The memory system remains a major performance bottleneck in modern and future architectures. In this dissertation, we propose a hardware/software cooperative approach and demonstrate its effectiveness. This approach combines the global yet imperfect view of the compiler with the timely yet narrow-scope context of the hardware. It relies on a light-weight extension to the instruction set architecture to convey compile-time knowledge (hints) to the hardware. The hardware then uses these hints to make better decisions.

Our work shows that a cooperative hardware/software approach to (1) cache replacement, (2) prefetching, and (3) their combination eliminates or tolerates much of the memory performance bottleneck. (1) Our work enhances cache replacement decisions using compiler hints. The compiler detects which data will or will not be reused and annotates loads accordingly. The compiler sets one bit (the *evict-me* bit) to denote a preferred eviction candidate. On a miss, the cache replacement algorithm preferentially replaces a cache

line with its evict-me bit set. Otherwise, it follows the LRU policy. The evict-me replacement scheme improves cache replacement decisions and is effective in both L1 and L2 caches. (2) We also use compiler hints to direct aggressive hardware region prefetching and content-aware pointer prefetching. The original SRP (scheduled region prefetching) engine queues prefetching requests on every outstanding L2 miss and tolerates latencies at the cost of dramatically increasing the memory traffic. GRP (guided region prefetching) enhances SRP by restricting prefetching to compiler-marked loads. Our compiler algorithms effectively mark spatial reuses across the SPEC CPU2000 benchmarks, and thus GRP achieves the performance of SRP with only one eighth of the additional traffic. (3) The evict-me cache replacement scheme helps alleviate the side effects of cache pollution introduced by useless region prefetches. The combination of evict-me caching and region prefetching further improves cache performance. These results demonstrate significant promise for overcoming the memory bottleneck with cooperative hardware/software techniques.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

In this dissertation we propose a hardware/software cooperative approach to improve computer system performance. We demonstrate that this approach is very effective and promising for the memory system, improving cache replacement, data prefetching, and their combination.

Modern processor speed continues to outpace memory speed [45]. Researchers project the performance gap will be even larger in the next ten years [3]. Hennessy and Patterson [45] illustrate the trend of memory-processor speed disparity as shown in Figure 1.1. The figure is based on the assumption that processor performance increases 55% per year from 1987 on, and 35% per year until 1986. In contrast, memory speed shows only 7% growth each year. This figure more or less reflects the reality of commercial systems. Figure 1.2 shows the gap of an Intel family [37], where each bar displays the peak number of instructions per memory access. It is easy to observe that there is an exponential increase of the bar height.

Both Figure 1.1 and Figure 1.2 describe the worst case scenario. In reality, a typical commercial architecture relies on a memory hierarchy to alleviate memory bottlenecks. A modern system usually contains two or more levels of cache, starting with the fastest Level 1 cache, which is closest to the CPU. Caches, which exploit program locality, perform well for some applications but are not always effective. Applications that have poor locality or large working sets often show poor cache performance. Several modern architectural features, such as out-of-order execution and lock-up free caches, effectively hide the Level 1 latencies for many programs [17, 100], though not all. These techniques, however, cannot

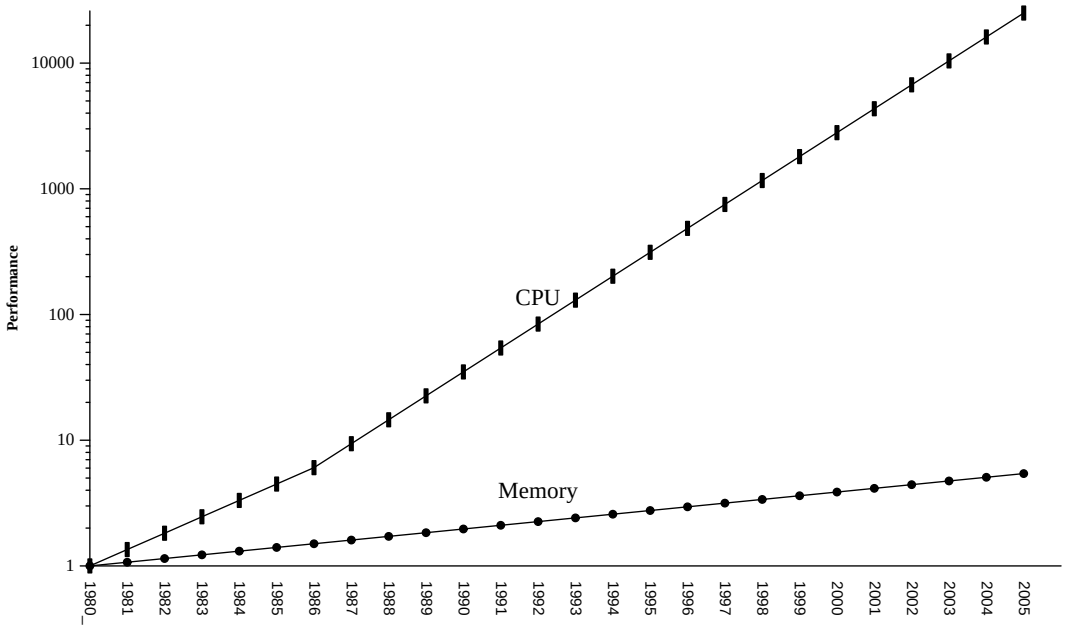

Figure 1.1. Processor and memory performance trend

hide the latencies of the Level 2 cache and beyond. Hundreds of cycles that result from DRAM accesses cannot be tolerated, thus causing significant performance degradation. For the SPEC CPU2000 benchmarks running on a modern high-performance microprocessor, over half of the time is spent stalling for loads that miss in the Level 2 cache [69]. We observe similar results in our simulations for a subset of SPEC CPU2000 benchmarks and *sphinx*, a speech recognition application [68]. Figure 1.3 compares the performance of a system with a configuration of a modern processor and a realistic memory hierarchy with two levels of cache versus a system with a perfect L1 cache and one with a perfect L2 cache with the stacked bar for each benchmark. The benchmarks are sorted by the size of the gap between a realistic system and one with a perfect L2 cache. The geometric mean of this performance gap is 33.7%.

Despite an enormous amount of research, memory stalls remain a challenge to computer performance. One reason is that past work does not fully exploit the strengths of both software and hardware. As we shall discuss in Chapter 2, existing techniques dominantly lean towards either a pure software solution, such as loop transformations and software

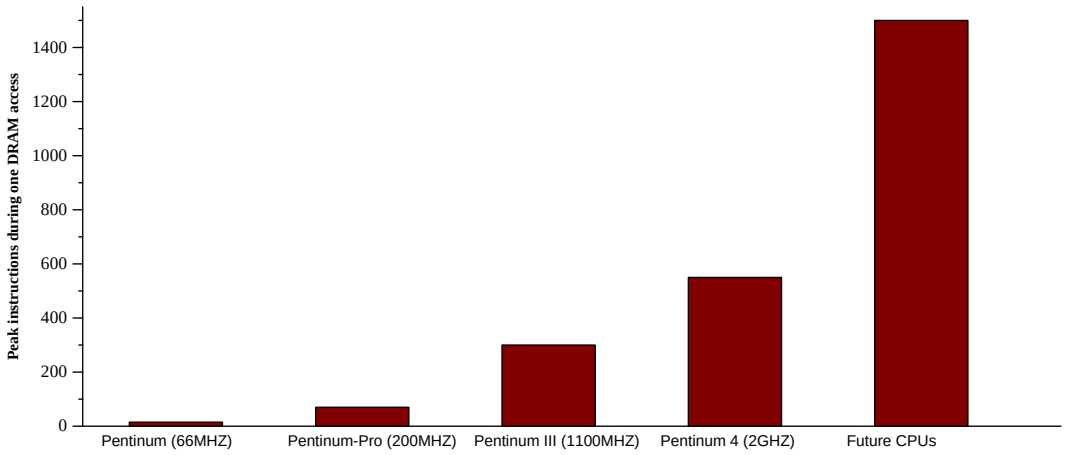

Figure 1.2. Peak instructions per memory access of Intel family

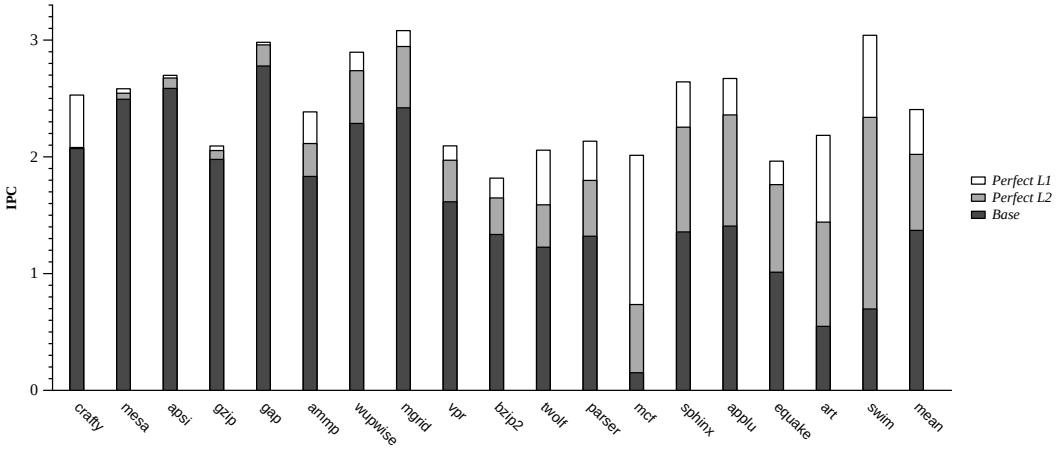

Figure 1.3. Processor performance

prefetching, or a pure hardware solution, such as the victim cache and hardware prefetching. Even the limited amount of research on hardware/software cooperation that we discuss in Section 2.3.3 is typically restrained by the current hardware/software interfaces. In this dissertation, we propose a novel hardware/software cooperative approach with a new hardware/software interface to address the increasing performance gap between the main memory and the processor.

1.1 Our Hardware/Software Cooperative Approach

Memory system performance can be improved with closer cooperation between software and hardware. Software has the advantage of its global view of the whole program, which is obtained through static compiler analysis. This static knowledge is typically coarse-grained and imprecise. For example, at compile time it is easy to detect if an array is accessed in a loop nest and reused in the following nest. But it is hard or impossible to calculate exactly how many array elements are reused if the loop bounds are unknown. However, even this imprecise view can serve as valuable guidance for run-time decisions. Run-time knowledge is typically precise but its scope is very limited. The information of the future execution at a point is unknown. Prediction from past behavior is usually restricted by a limited history because retaining a complete history of run-time status is prohibitive due to its very high storage and retrieval cost. By combining the strengths of software and hardware, the run-time system attains greater power to predict. Relying on compiler prediction combined with current run-time status and limited history, the run-time system can achieve high performance with relatively low cost. This dissertation investigates the memory system, but this approach can also be used to improve instruction level parallelism in a processor core and in other circumstances.

In the past, a limited amount of research has used a similar hardware/software approach to improve performance. As we discuss in Chapter 2, that research is typically specific to a particular application domain and restricted by poor interfaces between hardware and software. In most modern computer architectures, the hardware/software interface is limited to simple load/store instructions. The compiler generates memory access instructions. At run time, the processor sends requests to the memory system upon the execution of these instructions. It has no control beyond that. A memory request contains an address used to look up a value in the main memory or the cache. The request does not specify more details such as which slot the accessed data should sit in a cache set in order to exploit the best cache performance.

Our work extends the instruction set architecture (ISA) with a few bits in the memory instructions. The compiler encodes its global view of the whole problem into these bits. Combined with precise run-time status, the hardware is enhanced to use these bits to control memory system behavior and thus is able to utilize system resources better. This dissertation shows that this ISA extension is effective given its light-weight hardware support and compiler implementation. However, as the memory-processor performance gap keeps getting larger, we expect to need a richer interface, which can convey more information.

The cooperative approach can be applied to improve almost every aspect of the memory system including cache replacement, data prefetching, memory disambiguation, and cache coherence, to name a few.

1. Cache replacement. A typical cache replacement decision is made based on run-time history. Using the cooperative approach, the decision can be improved with knowledge of the future access pattern detected statically through compiler analysis.

2. Data prefetching. Cooperative data prefetching can achieve both the accuracy of software prefetching and the high performance of hardware prefetching. We can enhance hardware prefetching with compile-time locality information. We can depend on the run-time status to schedule prefetch requests and the compiler analysis to select what to prefetch.

3. Memory disambiguation. The interaction of the compiler and the hardware can supply us with a cost-effective run-time memory disambiguation technique to increase parallelism of memory instructions. Specifically, compile-time dependence testing and alias analysis can speed up speculative execution by predicting if a RAW (read after write) dependence exists between a store and a load.

4. Cache coherence. One application is to use the compiler to mark if a read or write is non-shared. This can reduce false sharing misses and speed up parallel execution.

In this dissertation we focus on application of our approach to cache replacement and data prefetching, aiming to improve cache performance. To achieve high cache performance we can rely on 1) hardware advances, which reduce the cycle penalty of cache misses, 2) techniques to reduce cache misses, or 3) approaches to tolerate latencies [45]. Our cache replacement work falls into the second category and our prefetching work into the third. Our work emphasizes the importance of software/hardware interaction to improve memory system performance.

1.2 Guided Cache Replacement

To attain fast cache access times, current microarchitectures have direct-mapped or low, 2 or 4-way, set-associative organizations [45, 47]. This choice trades off lower cache hit rates for higher clock rates to achieve better total performance. In set-associative caches, cache replacement policies determine which line to evict on a miss and will cause extra misses when making poor decisions. Current cache replacement policies typically rely only on run-time knowledge to make replacement choices. These policies do not always use cache memory effectively; i.e., even though the cache has sufficient capacity to retain data that will be reused in the future, they do not retain it [3, 14, 79]. Using a cooperative approach, we propose a novel compiler and architecture mechanism that uses compiler prediction of future accesses to improve cache replacement decisions directly. We particularly focus on enhancing the widely used LRU (least recently used) replacement policy.

Our new compiler mechanism guides cache replacements by selectively predicting when data will or will not be reused. We encode the compile-time prediction into memory instructions. We develop a comparative model that uses dependence and array section analysis to determine static locality patterns in a program. In Chapter 3, we first prove that our model matches or improves hit rates when compared to LRU. We then present an implementation that uses a single tag bit called the *evict-me* bit. On a miss, the architecture replaces a line with this bit set. Our compiler algorithm aggressively marks data as evict-

me if the data volume accessed between its reuse is (or it predicts the reuse is) greater than twice the cache size. The compiler can mark data aggressively since, if all the data fits in the cache, there will be no replacements. By applying the evict-me bit to both Level 1 and Level 2 caches, we observe up to 21% simulated performance improvements for current technology on a selection of scientific benchmarks and 34% for a technology prediction for 5 years from now [3]. On average, we reduce simulated execution time on our benchmarks by 5% to 16%, depending on the cache configuration. These results suggest that run-time cache replacements can benefit from the static compiler oracle, which tells data reuse patterns, to reduce cache misses and thus improve overall performance.

1.3 Guided Region Prefetching

The cache replacement techniques discussed in Section 1.2 help to reduce cache misses. But they cannot eliminate them all. For example, they help little on compulsory misses, which are caused by the first run-time accesses in an application. Prefetching is a popular technique to tolerate memory latencies. Researchers have proposed a large number of software and hardware prefetching schemes. Each of these two classes of prefetch solutions have distinct advantages and drawbacks. Pure software prefetching is typically highly accurate, but incurs run-time overhead and cannot issue prefetches sufficiently far in advance of a load to hide main memory access latencies [69]. Hardware-only schemes can prefetch spatial regions [23, 24, 54, 84, 92], pointer chains [32, 53, 89], or recurring patterns [66]. While these schemes can hide much of the main memory access time, they can also consume substantial amounts of memory bandwidth. This additional traffic does not always degrade uniprocessor performance, but it increases power consumption, and will likely degrade performance on multiprocessors. Since off-chip bandwidth will be the dominant limiter of scalability for future chip multiprocessors (CMPs) [51], prefetch schemes that consume bandwidth inefficiently will not be practical. While some schemes throttle

	Speedup	traffic increase	Performance gap from perfect L2 (%)
No prefetching	1	1	34
Stride prefetching	1.15	1.09	24
SRP	1.23	2.80	19
GRP	1.21	1.23	20

Table 1.1. Summary of prefetching performance and traffic

prefetching when the accuracy drops below a threshold, they then miss opportunities for issuing useful prefetches and thus trade performance against accuracy [36].

We propose an approach that builds on the strength of hardware and software prefetching, called Guided Region Prefetching (GRP). In GRP, sophisticated compiler analysis produces a rich set of load hints, including the presence or absence of spatial locality, pointer structures, or indirect array accesses. A run-time hardware engine, triggered by L2 cache misses, generates prefetches based on the compiler's hints. GRP thus benefits from compiler analysis of application reference patterns, but—unlike traditional software prefetching—the compiler is not required to generate or schedule individual prefetch addresses. Because the hardware generates the prefetches, it can run far ahead of the missing references. Because the compiler guides it, the hardware need not struggle to deduce future references with complex pattern matching on prior accesses stored in large tables.

Table 1.1 shows a summary of GRP results using the geometric mean of the SPEC CPU2000 benchmarks plus *sphinx*. Without prefetching, the mean performance across the benchmark suite is 34% lower than a perfect Level 2 cache. Stride prefetching (using the Sherwood et al. design [92]) provides a 15% speedup over no prefetching. SRP, which uses no compiler analysis, outperforms stride prefetching by 7%, but consumes excessive memory bandwidth, a 180% increase over a system with no prefetching. GRP provides near-equivalent performance to SRP but with substantially less traffic, an increase of only 23% over no prefetching. This reduction in traffic saves power and is more amenable to multiprocessor systems, where additional traffic can directly affect performance. To summarize, GRP as a cooperative prefetcher is able to make SRP, the hardware-only prefetcher,

practical by using compiler guidance. On the other hand, as is shown in previous work [69], SRP itself outperforms a state-of-the-art software prefetcher. The cooperative prefetcher thus provides a cost-effective solution for high performance.

1.4 Combining Cache Replacement and Region Prefetching

GRP is targeted to the Level 2 cache. Due to the complexity of the prefetching engine, it is impractical to implement a similar prefetching engine for the Level 1 cache. As shown in Figure 1.3, there is still a significant performance loss due to L1 stalls. Guided cache replacements alleviate this problem. However, even an optimal cache replacement policy usually cannot eliminate all misses. Given the high prefetching accuracy of GRP, most data prefetched to L2 will be used at L1. Accesses to these data at the Level 1 cache are L1 misses and thus suffer the L2 latencies. To tolerate these latencies, we design a prefetching engine between the Level 2 and Level 1 cache, which pushes the prefetched data into the Level 1 cache. Our results show that pushing prefetched data into the L1 cache can further improve memory system performance.

Typically, the Level 1 cache size is much smaller than the Level 2 cache. Pushing data into L1 introduces cache replacements and can pollute the cache. In Chapter 5, we examine the impact of the LRU and MRU placement policies. Furthermore, we combine guided cache replacement with L2 region prefetching and L1 data pushing. By marking a cache line as *evict-me*, the compiler optimistically predicts that the cache line will not be reused in the near future. The side effects of an unused prefetch or pushed line are reduced if it replaces an evict-me line. We find that compiler-guided replacement helps reduce the cache pollution of the push scheme by roughly half.

1.5 Dissertation Organization

We organize this dissertation as follows. In Chapter 2, we briefly cover some background material and discuss related work. We introduce basic notations for the memory

hierarchy and survey related techniques for tolerating cache latencies. We focus on the literature targeting cache replacement policies and data prefetching that are directly related to this dissertation. Finally, we introduce the Scale compiler infrastructure in which we implemented all our compiler algorithms.

In Chapter 3, we describe our cooperative cache replacement algorithms. We present a theoretical model in which we formulate our algorithms. We prove that the algorithms will generate hit rates at least matching LRU. We then present two implementation techniques. In one implementation, we use an impractical 16-bit hint to determine an upper bound of our approach. We then focus on a one-bit (evict-me) implementation and present the compiler analysis to generate this bit.

In Chapter 4, we introduce guided region prefetching and compare it with hardware-only region prefetching and stride prefetching. We present a series of compiler algorithms generating various compiler hints. We use these hints to enhance region prefetching and pointer prefetching. We show that our locality analysis is sufficient to catch most spatial reuses, so GRP is able to match the performance of SRP but reduce SRP's bus utilization to a practical level.

In Chapter 5, we first design a new prefetching engine, which pushes to the Level 1 cache the data prefetched into the Level 2 cache . We then put this push scheme, guided region prefetching, and guided cache replacement together. We show that the three methods interact well to improve performance further. We also observe that guided cache replacement can alleviate the pressure on cache replacement introduced by pushed data.

We conclude this dissertation in Chapter 6 by discussing the remaining problems and possible future work, and listing other applications of our cooperative approach.

1.6 Summary of Contributions

We make the following contributions in this dissertation:

1. Emphasizing the importance of hardware/software cooperation: There is only a limited amount of work that uses software/hardware collaboration to attack the memory wall problem. Within the research area, we emphasize the importance of the flexibility of using ISA extension to enrich the interface between software and hardware and propose practical applications of the extension. We develop and implement a series of compiler algorithms to manipulate the new ISA. We use simulators to explore the effectiveness of the compiler hints on performance and traffic.

2. A new cache replacement policy: We are the first to use those specific static compiler hints to direct run-time cache replacements. We describe a volume-based compile-time analysis to generate compiler hints and propose a practical cache architecture implementation based on a one-bit extension to the ISA and caches. We find that this new replacement policy is able to cut misses and often achieves miss rates close to optimal. Its performance depends on cache parameters and input data set sizes.

3. A new region prefetching technique: Our work distinguishes itself from previous region prefetching work in its compiler control. We use compiler hints to help hardware decide when to exercise prefetching and what is the appropriate prefetching region size. We find that compiler-guided region prefetching matches the performance of hardware-only region prefetching while reducing bus traffic to a practical level.

4. A thorough study of region prefetching and pointer prefetching: Our work thoroughly studies the interaction between region prefetching and pointer prefetching. We find region prefetching outperforms pointer prefetching in most cases, and their combination does not lead to a performance improvement.

5. A study of the interaction between cache replacement and prefetching: We study how a cache replacement policy can affect prefetching efficiency. Our results show that a well-tuned cache replacement policy can reduce the side effects of prefetching, such as cache pollution.

We show that our cooperative approach is an effective direction for addressing the memory wall problem. Through its applications for cache replacement and data prefetching, we demonstrate that compile-time analysis is able to supply copious information that the hardware can exploit to improve memory system performance. On the other hand, the run-time status tracked by hardware is critical for fully exploiting compiler hints. The limitation of this approach lies in its dependence on the ISA extension since the budget on ISA bits usage is very tight, particularly for RISC architectures. However, our work suggests it is a cost-effective way to improve memory system performance.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter provides background material and discusses related work. It first describes the most pertinent knowledge on how modern memory systems work and how our techniques benefit from state-of-the-art technologies. It then concentrates on related work in cache replacement, cache miss characteristics, and various prefetching techniques. We emphasize our contribution in both its underlying methodology and its breakthroughs in solving existing problems in memory systems. We also include a brief introduction to Scale, the compiler infrastructure we use to implement all of the compiler analyses described in this dissertation.

2.1 Memory System

In this section, we first address the role of the memory hierarchy in modern systems. We then focus on those recent advances in cache, DRAM, and processor core architectures that are most related to memory system performance and the hardware or software techniques discussed in this dissertation.

2.1.1 Cache Architecture and Cache Miss Classification

As latencies for accessing main memory keep growing, numerous techniques have been proposed and implemented to bridge the gap. Most of these techniques concentrate on the memory hierarchy. A typical *memory hierarchy* consists of register files, several levels of caches, the main memory, and the disk. The levels of a memory hierarchy usually follow an *inclusion* paradigm: all data in one level can be found in the level below. A higher level

(closer to the CPU) is faster but smaller than lower levels. Generally, the literature refers to all the levels between the CPU and the main memory as *caches*. We refer the reader to an early survey by Smith [98] for a comprehensive introduction to cache design and some techniques used to improve cache performance.

A *block* or a *line* is the minimum unit of information that can be present in the cache (*hit* in the cache) or not (*miss* in the cache). The restrictions on where a block can be placed in a cache create three categories of cache organizations. If each block has only one place it can appear in the cache, the cache is *direct mapped*. If it can be placed anywhere in the cache, the cache is *fully associative*. If it can be placed in a restricted set of places, the cache is *set associative*. In a set-associative cache, a *set* refers to a group of places each of which a block can be mapped to. A fully-associative cache can be considered as a special set-associative cache where the whole cache is a single set.

The memory hierarchy will speed up execution if accesses can be served at the upper levels (*hits*). Otherwise, it will suffer the longer latency due in *misses* to the higher levels. Hill and Smith classify cache misses into three categories: *compulsory*, *conflict*, or *capacity* [48]. A *compulsory* miss is the first access to a cache line. A *capacity* miss occurs when the cache size is too small to hold all the cache lines referenced by a program. With sufficient capacity, a *conflict* miss occurs when multiple cache lines are mapped to the same set in the cache, and the program subsequently references an evicted line. One can use the least recently used (LRU) replacement policy and a fully associative cache to define the three types of misses [46, 48, 67]. A capacity miss happens when a data item that is reused cannot be kept even in a fully associative cache. A conflict miss occurs when a reused data item *can* be kept in a fully associative cache, but is evicted due to limited cache associativity or a poor cache replacement decision in a given cache configuration. Using the LRU replacement policy makes statistics on different categories of cache misses simpler. Stricter statistics following the original definition require optimal cache replacement. Sugumar and

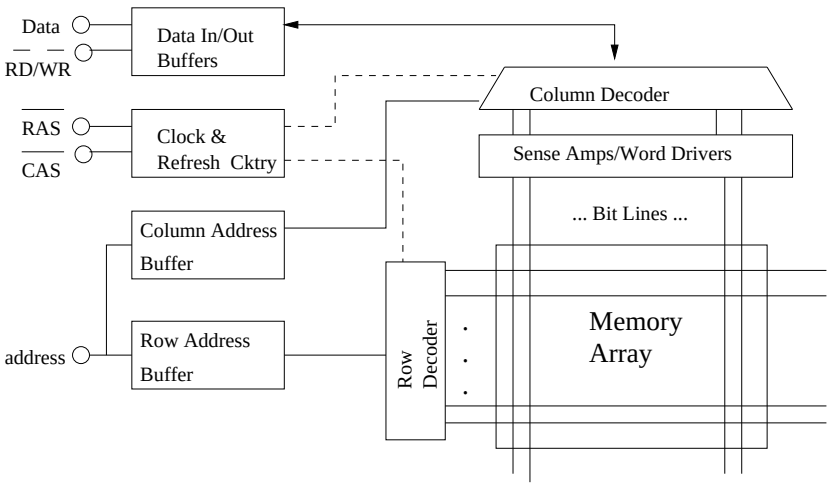

Figure 2.1. Conventional DRAM block diagram

Abraham suggest a measurement using optimal cache replacement [102] as we discuss in Section 2.2.2.

2.1.2 DRAM Architecture

Main memory is typically organized as DRAM (Dynamic Random Access Memory). Figure 2.1 illustrates a conventional DRAM. The term DRAM implies that an access to any randomly chosen location requires about the same amount of time. However, this is not the case since DRAM manufacturers have created several new DRAM architectures to respond to the memory wall problem. DRAM is conventionally arranged as a matrix of "cells". The memory accessing address is divided into a *row address* and a *column address*, which are then decoded to access the memory array. A data access sequence consists of a *row address strobe* ($\overline{RAS}$) signal followed by one or more *column address strobe* ($\overline{CAS}$) signals. The data in the storage cells of the decoded row address is moved into a bank of sense amplifiers during $\overline{RAS}$. In the following $\overline{CAS}$, the decoded column address selects data from the amplifiers.

In a conventional DRAM, there is only one $\overline{CAS}$ following each $\overline{RAS}$. In *fast page mode* DRAM, multiple $\overline{CAS}$ signals are allowed, and the amplifier set is called a *page* or a *hot row*. This DRAM is thus designed to exploit more spatial locality. We refer the reader to

Cuppu et al. [34] for characteristics of a list of representative DRAM techniques such as Synchronous DRAM (SDRAM) and Rambus DRAM (RDRAM).

Cuppu et al. [33] experiment with the performance effect of the system-level parameters of a DRAM system, such as the number of memory channels, burst sizes, queue sizes and organizations, turnaround overhead, memory controller page protocol, and algorithms for assigning request priorities and scheduling requests dynamically. They find that concurrency in the primary memory system is very important, even for a uniprocessor, and support for concurrent transactions improves performance by roughly a factor of two. They suggest that improving concurrency by subdividing the memory bus into multiple channels is risky, as it relies on the ability of the application to sustain the level of concurrency equal to the number of channels, otherwise the extra channels lie unused. From this point of view, the intra-channel concurrency is safer to exploit than inter-channel concurrency. Taking advantage of the Rambus design, Region Prefetching, as proposed by Lin et al. [69], issues prefetches only when a free channel is available and thus avoids channel contention with regular memory accesses. Our prefetching technique is an extension of this work and includes this optimization.

McKee et al. [76] find DRAM performance is largely dependent on the the order of accesses for streaming type workloads. They propose a stream buffer and a memory scheduling unit between the CPU/Cache and main memory. The compiler detects streams and generates stream access instructions. The memory scheduling unit is able to reorder the streaming accesses and the regular requests from caches and issue the accesses to main memory in an optimal order. The "optimality" comes from better hit rates to hot pages in modern DRAM systems. Generally, the access time of a page hit is a factor of two to three faster than a page miss. Modern DRAM designs, such as SDRAM and Rambus DRAM, make the access pattern and scheduling of main memory accesses more critical to the overall performance of the hierarchy. Varying cache replacements and prefetching policies change the data stream into and out of main memory. It is not the focus of this

dissertation to discuss the effect on main memory; however, both of our techniques aim to reduce the number of main memory accesses. Region prefetching takes direct advantage of the Rambus DRAM design. The prefetching requests are prioritized based on the availability of free channels.

2.2 Improving Cache Performance

In this section, we first introduce concepts of program locality, to which almost all cache improving approaches can be related. We list several theoretical studies of cache replacement policies. Then we discuss research on cache miss characteristics. We focus on hardware and software enhancements for improving cache performance and cache replacement algorithms. We single out related work in data prefetching and discuss it separately in Section 2.3. We conclude with some recent work on data remapping and cache coloring, and their impact on our work.

2.2.1 Program Locality

The performance benefits of a memory hierarchy stem from *program locality*. The classical notions of locality found in programs are: *temporal locality*—if an item is referenced, it will be referenced again soon; and *spatial locality*—if an item is referenced, an adjacent item will tend to be referenced soon [45]. When a program exhibits good locality, we expect most data will be available in the higher levels of the memory hierarchy; and thus we will avoid the longer latencies of the lower levels.

To improve cache performance, we can either improve program locality, exploit program locality better, or hide latencies of accessing data with poor locality. Examples of improving program locality are loop transformations, such as loop tiling and loop permutation. Our work on cache replacement policies is an example of exploiting program locality. Region prefetching exploits spatial locality and hides latencies.

2.2.2 Trace-based Cache Studies

A lot of theoretical cache work depends on having a complete program trace. Although our work does not depend on traces, some theoretical research motivates it. The notion of a program trace also facilitates our introduction to notations of data reuse.

Belady [10] pioneered research on replacement policies by comparing random cache replacement, LRU, and an optimal algorithm. His research originally targeted virtual memory page replacement, but the overall logic applies well to cache replacement. Given the trace of page accesses, the optimal algorithm should always replace the page with the largest reuse distance. A *reuse* of an access is the next access to the same address. *Reuse distance* is the number of distinct accesses between an access and its reuse. Belady proposed an optimal page replacement algorithm, called the MIN algorithm, given the entire program access sequence. Our cache replacement policy is inspired by the optimal cache replacement policy. Rather than relying on the whole program trace, we instead use static compiler analysis to predict the reuse distance.

Sugumar and Abraham [102] use Belady's algorithm to characterize capacity and conflict misses. They present three techniques for fast simulation of optimal cache replacement. Using a limited lookahead strategy, they are able to simulate multiple optimal caches with a one-pass scan. They also propose a tree-based fully-associative cache simulation and a partial inclusion scheme for simulating multiple set associativities. They find that the optimal miss rate is up to 70% lower than those under an LRU policy for 9 selected benchmarks. A simulator, *sim-cheetah*, implemented all these techniques. Sim-cheetah is the version adapted for SimpleScalar, a simulation tool set we use in this dissertation [15].

Temam [103] extends Belady's optimality result by simultaneously exploiting spatial and temporal locality. By considering both types of localities, the study evaluates the potential benefits of future memory optimizations and provides a performance upper bound for higher memory levels. Burger [13] uses the MIN algorithm to obtain a formal lower bound of the amount of bus traffic that a cache may produce.

All the studies discussed above seek to understand cache characteristics rather than to implement a real cache and related algorithms. Although our theoretical model in Section 3.2.1 is also based on static traces, we apply it to a real cache using compiler analysis. In Chapter 3, we also compare our algorithms to the optimal cache policy using Sugumar and Arbraham's simulation techniques [102].

2.2.3 Cache Miss Analysis

Ghosh et al. [39] suggest a set of miss equations for precisely analyzing cache misses for individual loop nests. Their framework enables compiler algorithms to find optimal solution for transformations like blocking, loop fusion, and padding. It also helps when reasoning about how different transformations work together.

Chatterjee et al. [22] set up a set of Presburger formulas to characterize and count cache misses. Chatterjee's model is powerful enough to handle imperfectly nested loops and various non-linear array layouts.

Both models could probably be extended to suggest evictions by calculating cache misses when applying different eviction schemes. They currently drive optimizations by comparing the number of misses between runs of the program compiled with different options. Our work uses heuristics and is less precise for an individual nest, but computes or estimates the data volume between nests and between reuses. A better cache miss analysis could improve our results.

2.2.4 Hardware Enhancement of Cache Replacement

Direct-mapped first level caches have been popular because of their low hit cycle time. They can yield good system performance, even though set-associative caches have lower miss rates [45, 47]. Due to rapid increases in miss cycle penalties, many recent architectures use at least 2-way set-associative L1 caches, e.g., the Compaq Alpha 21364 and Sun SPARC 2. To attain fast access time to L1 caches in future technologies, processors will probably have small L1 caches with a low degree of associativity [3]. We observe that the

industry is starting to deliver 3-cycle L1 caches. Some architectures trade higher associativity with a simpler cache replacement policy. For example, the IBM RS/6000 7043 has a 64K 128-way Level 1 cache that uses random replacement.

In Chapter 3, we propose an *evict-me* cache to enhance cache replacement. Each line is tagged with a bit called the *evict-me* bit. A line with the evict-me bit set is preferred for eviction on a cache miss. The hardware mechanisms of an evict-me cache do not increase cycle time and are effective only on set-associative caches; i.e., the hit time is unchanged. The replacement logic on a miss considers one more bit. Our work tries to achieve both fast hit cycle time and low miss rates.

The evict-me bit is similar to, but not the same as, the Alpha's *evict* instruction, which evicts a cache line immediately and thus cannot tolerate imprecision [58]. The evict instruction is designed to help maintain cache coherence, rather than to enhance locality. Our approach works for variable cache and data sizes because only when the data do not all fit in the cache will the replacement algorithm use our information. The Alpha's *prefetch and evict-next* instruction loads the line to the Level 1 cache and evicts it on the next miss to the cache set [58], but we instead tag actual loads, not speculative prefetches.

Numerous dynamic or hardware techniques have been proposed to reduce cache misses or alleviate cache pollution to improve hit rates, e.g., [2, 52, 54]. The victim cache was originally designed to enhance direct-mapped caches [54]. It is a small fully-associative buffer between the Level 1 and 2 cache, which stores replaced data to reduce conflict misses that occur close together in time. The evicted L1 cache line exchanges with the hit line in the victim buffer. The victim cache is probabilistic, rather than predictive.

Wong and Baer [112] enhance LRU with a temporal bit for each cache line. Temporal bits act oppositely to our evict-me bits: they specify lines to *retain* rather than lines to *evict*. Wong and Baer determine temporal bit settings using profiling or an online hardware history table. The temporal bit of a cache line is reset when the line is hit. To avoid a marked dead line polluting the cache, the temporal bit of the LRU line is reset when a non-

LRU line is evicted. Rivers et al. [87] use a (hardware) detection unit, similar to a history table, to track reuses at run time and to categorize access as temporal/non-temporal and cacheable/non-cacheable. A non-cacheable access bypasses the cache to avoid pollution. Lai et al. [66] use a hardware history table to predict when a cache block is dead and which block to prefetch to replace the dead one. Our technique is based on static compiler analysis and does not require substantially additional hardware.

Hannnor and Reinhardt [43] present a practical, fully associative, software-managed secondary cache. Their system consists of an *indirect index cache* (IIC) and a replacement algorithm, *generational replacement*. The IIC's tag array is organized as a hash table and each tag entry contains a pointer to the data block, which makes the cache fully associative because the pointer can legally point to any block in the cache. Hannor and Reinhardt group cache blocks into a small number of prioritized pools. The software-managed generational replacement promotes and demotes cache blocks on a miss depending on their recent reference history. The software replacement algorithm achieves miss rate reductions from 8 to 85% compared to a 4-way LRU. Hannor and Reinhardt use software management to reduce the complexity of the hardware implementation of their design. Their replacement algorithm totally relies on run-time history and does not use any compiler-time analysis.

McKee et al. [77] use a stream buffer for stream-like data to bypass the cache. They rely on the compiler to detect stream array accesses and generate a special instruction to start a stream at run time. We mark stream data as *evict-me*, but our technique works on cache replacement directly and does not require an extra buffer.

The Intel IA-64 provides instructions to control caching [38]. Non-temporal loads/stores bypass the cache to avoid cache pollution due to streaming data. The IA-64 supports locality hints used by prefetch, load, and store instructions to control placements of cache lines in either a *temporal structure* or a *non-temporal structure*. The hints do not direct cache replacement, but our compiler analysis could specify the non-temporal instructions and locality hints. We do not explore that application in this dissertation.

2.2.5 Page/Cache Coloring and Data Remapping

Coloring is an approach to classify pages and cache lines used to assist varies run-time decisions. On-line page coloring and other mechanisms decrease paging, but are too expensive for higher levels of the memory hierarchy. For example, Early Eviction LRU (EELRU) [96] dynamically chooses to evict the LRU page or the e^{th} most recently used page. The reference history determines *e*, the *early eviction point*, but is too expensive to store and use for caches. This approach eliminates capacity page misses in a fully associative memory, whereas our technique removes conflict misses for caches, using static compiler control.

Some work uses cache coloring or data remapping to improve cache effectiveness and reduce conflict misses due to poor mapping [12, 18, 29, 91]. Calder et al. [18] use profiling to build a *Temporal Relationship Graph* (TRG), which shows a metric of cache interference among stack (local variables), global variables, heap objects, and constants. Using TRG, they propose an algorithm to decide the placement of each object in order to reduce cache interferences.

Sherwood et al. propose a hardware and a software approach to reduce cache misses by reordering pages in cache [91]. The software approach provides a color mapping at compile time for code and data pages, which can then be used by the operating system to guide its allocation of physical pages. The hardware approach works by adding a page remap field to the TLB, which is used to allow a page to be remapped to a different color in the physically indexed cache while keeping the same physical page in memory. Bugnion et al. implement a similar software page coloring approach for multi-processor systems [12]. They use compiler analysis of access patterns to direct the operating system to allocate physical pages.

Chilimbi and Larus [29] use generational garbage collection to implement cache-conscious data placement. They reorganize objects at garbage collection time to improve data local-

ity and thus cache performance. Chillimbi et al. [27, 28] later on provide a more detailed analysis and an implementation using an extension of memory allocation functions.

Compared to cache coloring, our cache replacement algorithms try to reduce cache misses at a finer level, i.e., cache sets. Even in colored caches, our algorithms can most likely still help reduce cache misses. A program with improved locality will leave a smaller space for our cache replacement to improve. However, the static compiler analysis will still help as long as the data movements for locality improvements do not destroy those static properties.

2.2.6 Improving Cache Locality—Program Transformations

Researchers have also proposed loop and data transformations to improve data locality by moving temporal reuses closer together in time and by introducing spatial locality [1, 55, 78, 111]. These algorithms do not directly improve replacement decisions and thus are complementary to our work. Region prefetching, which we describe in Chapter 4, sometimes benefits from improved spatial locality introduced by loop transformations. Region prefetching itself exploits spatial locality. It will show better prefetching accuracy and improved performance when combined with loop transformations. We leave this combination as future work.

2.2.7 Out-of-order Execution and Lock-up Free Caches

An out-of-order execution processor core is able to hide some cache latencies through a combination of dynamic scheduling and a lock-up free cache [63, 14]. The processor executes instructions when the operands become available rather than in the order that the program specifies. Out-of-order processors exploit instruction level parallelism by allowing other instructions to execute when an instruction stalls in the processor waiting for a resource. Out-of-order processors use a fixed-size instruction window from which instructions may be executed. In order to preserve program semantics, the processor typically retires or commits the instructions in order. The amount of latency that an out-of-order

processor is able to tolerate depends upon the amount of instruction level parallelism (ILP) and the size of the instruction window. Most high performance commercial processors support out-of-order execution, including the Alpha 21264 [58, 59], MIPS R10000 [115], PowerPC, and Intel Pentium [49].

2.3 Prefetching Techniques

In this section, we focus on the most pertinent aspects of the large body of literature on software and hardware data prefetching, along with the small number of previously proposed hybrid schemes. Typically we refer to prefetching techniques using compiler-generated explicit prefetching instructions as *software prefetching*. Certainly the prefetching instructions need to be implemented in hardware. *Hardware prefetching*, on the other hand, does not require special compiler support. In contrast, it generates prefetching requests at run time based on run-time state. Hardware/software cooperative prefetching uses both run-time state and static compile-time knowledge to direct prefetching.

2.3.1 Software Prefetching

In this section we first address some general issues associated with software prefetching, its strengths and weaknesses. Then we survey the major work in this area.

Software prefetching relies on non-blocking prefetch instructions that bring the indicated block of memory into the cache, much like a load instruction. Conceptually, the latency of a given load instruction is hidden by inserting a prefetch with the same effective address into the instruction stream sufficiently far in advance of the load. Because the compiler inserts prefetches only for loads guaranteed to occur (or very likely), software prefetch *accuracy* is typically high. In practice, the compiler faces two key challenges in data prefetching: *selection* and *scheduling*.

Because prefetch instructions occupy instruction cache space, pipeline slots, and data cache ports, the compiler must *select* a subset of the loads for which to generate prefetches.

Accurate compile-time identification of the loads that will cause cache misses at run time is complex, requiring both knowledge of hardware parameters (cache block size, capacity, and associativity) and sophisticated code analysis (e.g., to determine the volume of other data accessed between references to a particular block) [19, 39, 82, 113].

The compiler also faces the difficult challenge of scheduling the prefetches sufficiently early to hide the memory latency, but not so early that useful data are needlessly evicted. To find that point, the compiler must estimate cache miss latencies and run-time instruction execution rates [62]. The compiler is further constrained in that it cannot schedule a prefetch until it can compute the effective address. While this constraint is not significant for arrays [16, 82], it limits compiler-based greedy pointer prefetching [17, 72, 88]. Jump pointers bypass this limitation by identifying records several links ahead in the structure, but require much more sophisticated analysis, dynamic updates, and the addition of a jump pointer to each object [17, 72, 88]. Other approaches prefetch pointer arguments at call sites [71], and decouple prefetches from the main program using a separate thread context [31, 61, 74].

Despite these challenges to software prefetching, the compiler analyses themselves are usually sufficient to detect where prefetching opportunities exist. The difficulty largely arises from the lack of run-time information so that prefetching can be issued on time. Our guided region prefetching technique also depends on compiler analysis and most of our algorithms are similar to or derive from past work in software prefetching. However, our technique distinguishes itself by exploiting run-time information as well.

2.3.1.1 Software Array Prefetching

Software array prefetching typically focuses on array references in loops of scientific applications. It generates software prefetching instructions for likely future array references.

Callahan, Kennedy, and Porterfield [19] present a simple array prefetching algorithm that first detects loop induction variables. The algorithm inserts a nonblocking prefetching instruction for an array reference by incrementing the induction variable in the array indices by s. It denotes a reference s iterations away if the coefficient of the loop induction variable is 1. Based on the observation that one loop iteration usually consumes sufficient computing time, they then just prefetch one iteration ahead ($s = 1$), which significantly improves hit rates. This strategy has been revisited as memory latency continues to increase: prefetching one iteration ahead is often too late.

Klaiber and Levy suggest prefetching data into a separate buffer, called a *fetchbuffer*, to avoid polluting the cache [62]. They analyze the impact of prefetching distance and manually insert prefetching instructions into selected Livermore loops using their analysis. They use the average memory access time per access as a performance measurement and show a significant speedup for numeric applications. Two non-numeric applications, quicksort and binary search, present performance improvements not as dramatic and are dependent on relatively larger data set sizes.

Mowry, Lam, and Gupta [82] present a thorough evaluation and implementation of software prefetching. They also show simulated execution time. Their algorithm inserts a *prolog* loop to prefetch for the first several iterations. It also generates an *epilog* loop to avoid prefetching non-existing array references. They propose the simple formula $\lceil l/s \rceil$ to calculate prefetching distance, where l is the prefetching latency and s is the shortest path through the loop body. Their results show a speedup of up to a factor of two. They conclude that prefetching into cache directly performs impressively well, without the disadvantage of sacrificing cache size to the fetch buffer that Klaiber and Levy propose [62].

In his dissertation [75], McIntosh introduces cross-loop reuse analysis to reduce useless prefetches. Given two adjacent loop nests, some data sections accessed in the second nest are probably accessed by the first nest and stay in the cache. McIntosh implements compiler analysis to detect these reuses and disable useless prefetches in the second nest.

Cahoon and McKinley [16] present a unified compile-time analysis for software prefetching both arrays and linked structures. Their data flow analysis detects loop induction variables in array accesses and schedule prefetches for them. Across a series of array-based Java benchmarks, their technique reduces execution time by 23% on average.

Two reports evaluate software prefetching on commercial processors using the HP PA-8000 and the PowerPC. We also notice that the Alpha compiler has software prefetching implemented. Santhanam, Gornish, and Hsu [90] evaluate software data prefetching on the HP PA-8000, a 4-way superscalar processor. They report the interaction of data prefetching with loop unrolling and array padding. They also provide a detailed analysis of prefetching distance with consideration of memory latency and the number of outstanding misses the processor allows. Santhanam et al. present results showing a 26% speedup on the SPECfp95 benchmark suite. Bernstein et al. [11] describe a compiler implementation for data prefetching on the PowerPC architecture. They follow Mowry's approach but the only transformation they apply is loop unrolling [82]. They provide actual execution times for the SPECfp92 benchmarks and Nasa7 kernels. Improvements occur on only three of the fourteen SPECfp92 programs and six of the seven Nasa7 kernels.

2.3.1.2 Other Software Prefetching Techniques

In this section, we survey related prefetching techniques for linked data structures and those suitable for general data correlations rather than only regular array accesses.

Early pointer prefetching work proposed by Lipasti et al. [71] uses a compile-time heuristic, called *SPAID* (speculatively prefetching anticipated inter-procedural dereferences), for prefetching pointer arguments at call sites. Their experiments use a trace-driven statistical model. They report the best performance when prefetching one argument at a call. Since the work is restricted to prefetching arguments, the overall performance effect is limited.

To address latencies in general pointer-based applications, Luk and Mowry [72, 73] design three prefetching schemes, *greedy* prefetching, *history pointer* prefetching, and *data-linearization* prefetching. They use type declarations and control flow analysis to detect recursive traversal. In particular they check pointer updates in loops and recursive calls. We use the same technique to mark recursive pointer references in our guided pointer prefetching. The greedy prefetching scheme prefetches all remaining pointed-to nodes except the immediately following one. History-based prefetching tries to build jump pointers on-the-fly during the first traversal. Data-linearization prefetching maps linked structures into sequential memory locations so the prefetcher can enjoy an array-like prefetching context. Experimenting on the Olden benchmarks, they show up to 45% speedup with greedy prefetching, which outperforms SPAID in all but one case.

Cahoon and McKinley [17] describe a data flow analysis to generate prefetches for Java code. They investigate greedy prefetching and jump pointer prefetching and find inter-procedural analysis is critical for detecting recursive structures in Java.

Luk [74] uses a pre-executing thread to prefetch data for the main thread for simultaneous multi-threading (SMT) processors. He presents a compiler analysis, though it is not implemented, to generate pre-execution code and add instructions in the main code to initiate and stop the pre-execution. His technique shows a 24% speedup over non-prefetching and a 19% improvement over state-of-the-art software prefetching.

Another approach uses profiling to detect which data to prefetch. The profiling-based technique has the advantage of detecting those irregular data access patterns that are hard for the compiler to find. However, like all profiling-based schemes, it suffers from training cost and generality.

Chilimbi and Hirzel [30] present a dynamic prefetching technique relying on run-time sampling. Their technique is divided into three phases: profiling, analysis and prefetching, and hibernation. The profiling phase gathers a temporal reference profile. The second phase analyzes the profile and dynamically injects prefetching code. The program en-

ters the hibernation phase when there is no profiling or analysis in action. Since all the processes depend on software instrumentation and the prefetching requires no more than regular prefetching instructions, we still classify this technique as software prefetching.

Wu [113] uses profiling to detect stride patterns in irregular code. His analysis is efficient at collecting both frequency and stride in the same profiling pass. The compiler uses the stride profiling to generate prefetching instructions. Wu observes up to 59% speedup for SPEC CPU2000 benchmarks.

2.3.2 Hardware Prefetching

The converse approach to software prefetching is hardware-only prefetching, in which the hardware predicts prefetch addresses by observing a program's run-time behavior. Since prefetches do not incur overhead in the processor itself, the hardware need not be as selective about issuing prefetch operations. Recent work shows that simple dynamic prioritization techniques eliminate most memory bandwidth contention and cache pollution problems [69]. However, unlike the compiler, the hardware has no direct knowledge of future memory references; the key challenge in hardware-based prefetching is determining a reasonable set of predicted addresses to use as prefetch targets. Hardware prefetching thus suffers relative to software prefetching in both accuracy (because the predictions may be wrong) and coverage (because some addresses may require the compiler's scope to predict).

Many hardware prefetchers exploit only spatial locality, prefetching one or more subsequent blocks on a cache miss [35, 54, 98]. More sophisticated schemes detect non-unit stride access patterns, such as Chen and Baer's reference prediction table (RPT) [23] and Palacharla and Kessler's stride stream buffers [84]. Other approaches exploit pointer-based access sequences, as with correlation-based and Markov prefetching [4, 21, 53], or a broader class of patterns, using dead block information [66]. Another approach involves decoupling data structure traversal from the computation, using specialized pointer-traversal

hardware [89] or dedicated pre-execution hardware [5]. Researchers have also proposed memory-side prefetching to reduce latencies between prefetches [50, 99, 114].

Most pertinent to this work are two previous papers. First, predictor-directed stream buffering, proposed by Sherwood et al. [92], unifies stride stream buffers and Markov prefetching into a single, consistent hardware prefetching framework. In Section 4.5, we compare the GRP scheme to the stride stream buffers scheme only, since the Markov predictor consumes too much state to be practical. Second, Cooksey et al. [32] propose a stateless approach to pointer prefetching, foregoing explicit identification of pointer traversal patterns and simply prefetching any referenced memory value that could be reasonably interpreted as a memory address. Our hardware schemes are also stateless. We find that for our benchmarks, GRP with spatial hints usually performs better or the same as pointer prediction with or without pointer hints.

In the end, all hardware schemes are forced to trade coverage for accuracy (or vice versa), and focus either only on structured access patterns, which can be predicted with high accuracy (forgoing coverage of less structured access patterns), or consume significant bandwidth with incorrect prefetches in an attempt to cover less-structured references.

The relative strengths and weaknesses of hardware and software prefetching are complementary and thus suggest a combined hardware/software approach. An ideal scheme would exploit the compiler's knowledge of future reference patterns, and use a low-overhead channel to convey this information to a hardware prefetching engine, which could then generate and schedule appropriate prefetches based on dynamic information regarding cache miss events and resource availability.

2.3.2.1 Scheduled Region Prefetching

GRP, discussed in Chapter 4, uses and compares with Lin et al.'s hardware-only region prefetching technique [69]. We discuss in this section the major contributions of their work and leave the experimental comparison to Chapter 4.

Scheduled Region Prefetching exploits spatial locality of a program by prefetching a region on an L2 demand miss. Lin et al. find that a 4K byte region size delivers good performance. Prefetching such a big region exerts pressure on memory buses. This work uses several sophisticated scheduling and prioritization approaches to counteract this pressure. A prefetched cache line is put into the LRU slot to reduce cache pollution. A prefetching request is dequeued from the prefetching queue following the last-in-first-out (LIFO) policy. This prioritization guarantees that the most recent prefetch request is served first and also maximizes DRAM page locality. A prefetching request yields to a regular L2 miss and is issued only when a memory channel is free. This ensures that the regular memory request is served first. These scheduling techniques reduce the negative effects of poor accuracy but do not entirely eliminate prefetches that fetch useless data. GRP significantly improves the accuracy using compiler hints.

2.3.2.2 Predictor-directed Stream Buffer

Sherwood, Sair, and Calder [92] combine a stride and a Markov predictor to prefetch for *Predictor-Directed Stream Buffers* (PDSB). The stride predictor is able to exploit spatial locality in stride-intensive code, whereas the Markov predictor is targeted to pointer-intensive code. The basic architecture is as shown in Figure 2.2.

We choose this prefetcher to compare with region prefetching in Chapter 4 because it accommodates a set of hardware enhancements to improve prefetching accuracy and avoid stream buffer thrashing. Following Jouppi's original design [54], each stream buffer contains a FIFO buffer queue. A hit to a stream buffer will free the head entry of the queue of the buffer and the buffer will then allow a new prefetch. To control accuracy, Sherwood et al. further add a priority counter to each stream buffer. Their prefetcher increments the counter when there is a hit and decrements it when there is a miss. The prefetcher uses the counter to decide which stream buffer performs the next prediction or prefetch. The prefetcher relies on a PC-based history table to generate a stride for each stream buffer. It

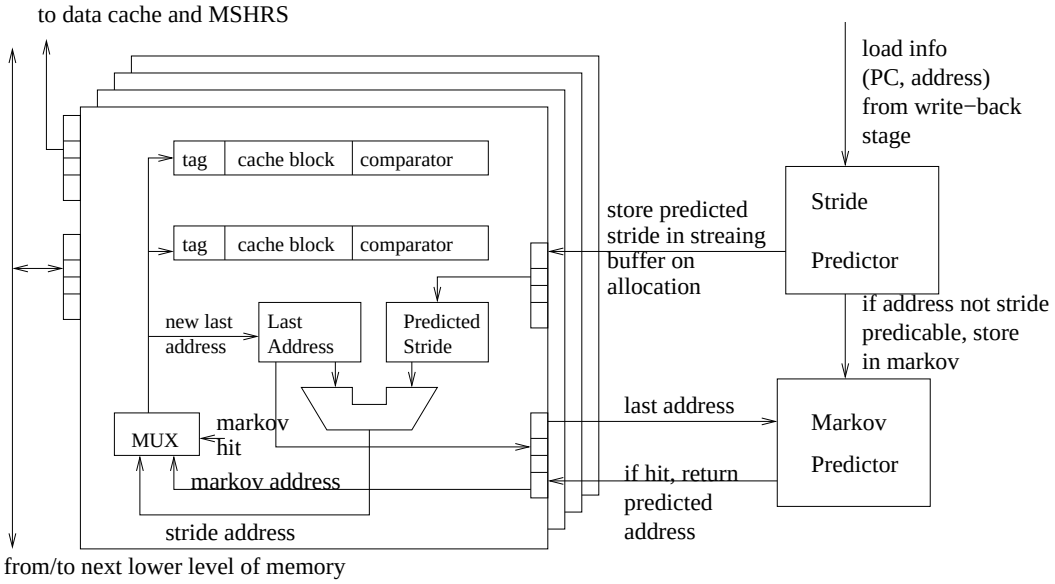

Figure 2.2. Predictor-directed stream buffer architecture

generates a new stride when there are two misses in a row exhibiting the same stride. Each table entry stores an *accuracy counter*, which is incremented every time the load's update address matches the prediction. In *confidence allocation* mode, the prefetcher allocates a new stream buffer only when the value of an accuracy counter is greater than that of it corresponding priority counter. Then the stream buffer with the lowest priority yields to the new stream built upon the stride table entry.

Compared to PDSB, GRP prefetches data into the Level 2 cache directly. It avoids the hardware cost of accuracy control by using compiler hints. Both the stride prefetcher in PDSB and GRP exploit spatial locality. It would be interesting to see if the compiler hints in GRP can drive PDSB and eliminate part of PDSB's hardware accuracy control.

2.3.2.3 Hardware Array and Spatial Prefetching

Hardware prefetching schemes add extra hardware in order to prefetch and do not require additional software support. This section discusses related work exploiting spatial locally, particularly in arrays. Hardware prefetching helps to boost application performance without re-compiling. Most hardware prefetching targets array-based scientific code to exploit spatial reuses. It could also exploit spatial reuses of other data structures since the

technique is dynamic and automatic. We discuss hardware pointer prefetching in Section 2.3.2.4.

Gintele [40] suggests a simple one block lookahead prefetching scheme: if line i is referenced, only line $i+1$ is considered. This scheme successfully reduces misses on mainframe computers. The simplicity of the scheme helps attain the high hit cycle times of caches. Smith [97, 98] points out that it is not feasible to apply other choices that require more complicated hardware implementations. However, as cache latencies increase, more aggressive hardware prefetchers become practical and necessary. Smith concludes that treating the prefetched lines differently in the LRU scheme has little effect, although Lin et al. [69] observe significant performance impact in an aggressive prefetch for modern architectures.

To control the number of useless prefetches, Gintele [40] proposes a variant of one block lookahead prefetching, called *tagged prefetching*. A tag bit associated with each cache line is set to one whenever the line is accessed. Any line brought to the cache by a prefetch operation retains its tag as zero. When a tag changes from 0 to 1 (i.e., when the line is referenced for the first time after prefetching or demand fetched), a prefetch is initiated for the next sequential line. After initiating prefetching, the tag bit is reset.

To avoid cache pollution by prefetched lines, Jouppi [54] proposes prefetching into a separate buffer, called the *stream buffer*. A stream buffer consists of a series of entries, each consisting of a tag, an available bit, and a data line. A prefetch is initiated by a cache miss. The lines following the missed line are then fetched into the stream buffer. A cache lookup will also check the stream buffer and move the hit entry into the cache. In a single-stream design, a miss to the stream buffer will reset the buffer. In multi-way stream buffers, a miss to all buffers causes a reset to the least recently hit (LRU) buffer. Multi-way stream buffers are useful for data references that contain interleaved accesses to several large data structures, nearly doubling the performance of a single stream buffer and removing 43% of the overall misses.

Palacharla and Kessler [84] enhance Jouppi's stream buffers [54] with two techniques: a *filtering* scheme, used to reduce memory bandwidth requirements, and a scheme that enables non-unit stride prefetching. The filter allocates a new stream buffer only when a miss to cache block i is followed by a miss to cache block $i+1$. Then the stream buffer will prefetch cache block $i+2$, $i+3$, and so on. Physical address space is dynamically partitioned and a stride is detected in each partition if two strides among three consecutive misses in a partition are the same.

Chen and Baer [9, 23, 26, 24] use a PC-based reference prediction table (RPT) to detect strides. Each entry consists of four fields: a *tag*, a *prev-addr*, a *stride*. and a two-bit *state*. The *tag* field corresponds to the PC value (the address of the load/store instruction). The *prev-addr* records the memory address that the load/store instruction previously accessed. The *stride* is the difference between the last two generated addresses. Finally, the *state* bits control when to initiate a prefetch and when to disable the prediction. Chen and Baer propose three schemes to generate a prefetching address based on the RPT. The basic scheme uses the current PC to look up in the RPT and predict an address. The so-called *lookahead reference prediction* technique uses a pseudo-PC called a *Look-Ahead Program Counter* (LA-PC) that remains δ cycles ahead of the regular PC. It predicts an access several iterations away. The *correlated reference prediction* scheme further helps to handle stride change across loop levels. They conclude that all three schemes are very effective in miss reduction but that the lookahead prediction scheme is best in terms of overall cost and performance.

Sherwood and Calder [92] implement a similar RPT table in combination with a Markov predictor. They also introduce more hardware control to improve prefetching accuracy. We discuss Sherwood and Calder's work separately in Section 2.3.2.2 and we compare it with region prefetching in Chapter 4.

Johnson et al. [52] propose a run-time spatial locality detection mechanism. They use a hardware table to keep track of spatial locality dynamically. The fetch size can be varied

depending on the spatial locality of fetched data. Their work helps reduce cache pollution caused by fetching large blocks unnecessarily.

Following Johnson et al, Kumar and Wilkerson [65] use a *Spatial Footprint Predictor* (SFP) to predict which portions of a cache block will get reused before getting evicted. SFP predicts the neighboring words that should be prefetched on a cache miss. Their evaluations show an average 18% miss reduction and a significant reduction in the bandwidth requirement.

2.3.2.4 Hardware Pointer and Correlation Prefetching

Array prefetching takes advantage of spatial co-location of array elements. The prefetching address can be determined from the previous address using a constant increment. For applications with linked structures, and irregular access patterns such as indirect array references, this simple prediction does not work. In this section, we first describe correlation-based prefetching where the prediction is based on general reference correlations. We then discuss the hardware-only techniques that focus on linked structures.

Charney and Reeves [21] were the first to publish the results of correlation prefetching. They use a *pair cache* to store ancestor-descendant reference pairs instead of parent-child pairs. They study different combinations of bits from the instruction and the data addresses of L1 miss references, which serve as a lookup index. They also find that the combination of a stride prefetcher and a correlation-based prefetcher provides a significant improvement in prefetch coverage over using either approach alone.

Alexander and Kedem [4] use a *distributed* prediction table to predict the next prefetch address. The table is indexed by the current miss address, and each entry consists of multiple predicting addresses. It predicts bit-line accesses in an Enhanced DRAM and prefetches individual bit lines from the DRAM to the SRAM array. This mechanism should work for irregular data accesses such as linked structures although the authors run experiments only

on a set of scientific applications. Their results show a more than 40% performance improvement on 2 of 8 benchmarks while the effect on the others is modest.

Joseph and Grunwald [53] use a Markov prefetching table to prefetch data into on-chip prefetch buffers in parallel with the Level 1 caches. The Markov table records the transition probabilities for one miss address to a set of possible subsequent missing addresses. The follow-up entries are organized using an LRU policy to avoid storing the real probabilities. To predict the next Level 1 miss, the size of the Markov table is of the same magnitude as the Level 2 cache, which makes the scheme impractical for Level 2 prefetching. Joseph and Grunwald evaluate the effectiveness of their Markov predictor using commercial workloads that contain mostly unstructured references. Compared to stream buffers and stride prefetchers, the Markov prefetcher shows better coverage but using more bandwidth and sacrificing accuracy.

Roth and Sohi [88] combine jump-pointer prefetching and chained prefetching, which use the pointers in the original unmodified program. They propose four schemes: *queue jumping*, *full jumping*, *chain jumping*, and *root jumping*. They describe three implementations for each scheme: software, hardware, and software/hardware cooperation. Queue jumping is jump-pointer prefetching applied only to the "backbone" structures that contain nodes of just one type. Full jumping prefetches both the "backbones", and the "ribs", which are the nodes pointed to from the backbone nodes. Full jumping relies on having jump pointers to both the "backbone" and the "rib" nodes. Chain jumping eliminates the jump pointer to "ribs" by applying jump-pointer prefetching to "backbones" and chained prefetching to "ribs". Root jumping starts with chained prefetching from a root without requiring any jump pointers. It particularly targets small, highly dynamic pointer structures. On a suite of pointer intensive programs, Roth and Sohi's jump pointer prefetching reduces memory stall time by 72% for a software implementation, 83% for cooperative, and 55% for hardware, producing speedups of 15%, 20%, and 22% respectively. Our pointer prefetching scheme, discussed in Chapter 4, does not generate and use jump pointers. The

performance improvement of our scheme is thus less significant than that of Roth and Sohi's.

Cooksey et al. [32] present a *content-aware* pointer prefetching scheme, which is essentially the same as the one we designed and implemented concurrently as described in Chapter 4. A fetched cache line is scanned word by word to detect if the value falls into the heap range and thus looks like a pointer. The prefetching engine then uses this value as an address to prefetch. The *pointer chasing* scheme continues prefetching by chasing pointers in prefetched cache lines. Cooksey et al. show a speedup of 10% on a subset of real-world applications, while we find that this technique is mostly subsumed by region prefetching on selected SPEC2000 benchmarks.

2.3.3 Hardware/Software Cooperative Prefetching

The limited previous work in this area has either exploited prefetching for restricted classes of access patterns, or provided an interface that is overly general and complex. Gornish and Veidenbaum [42] let software select the number of contiguous blocks to prefetch upon a miss, whereas Chen and Baer [24, 25] use the compiler to supply address and stride information to augment a reference prediction table. Skeppstedt and Dubois use a trap handler to trigger prefetching using similar information [95]. Karlsson et al. [56] use *prefetch arrays* to enable a hardware engine to perform a generalized variant of greedy and jump-pointer prefetching. Zhang and Torrellas [117] use the compiler to mark blocks in memory as belonging to contiguous spatially local regions or containing indirection pointers. Their scheme requires additional bits in main memory and significant support in the memory controller. Roth and Sohi's [88] cooperative jump pointer prefetching uses the hardware to build data dependences among linked data structure loads and relies on the software to trigger chained prefetching. Finally, fully programmable prefetch engines provide flexibility but require significant memory system support and have not yet demonstrated that the required compiler support is realistic [99, 109, 114].

Extending Smith's one block look-ahead (OBL) technique [98], Gornish and Veidenbaum [42] use compiler support to select a *prefetching degree* (PD), i.e., how many lines ahead to prefetch. A software prefetch instruction is extended to specify its prefetching degree and an additional field in each cache line is reserved to store the degree. When cache line l is accessed, line $l + degree(l)$ is prefetched. Gornish and Veidenbaum's original work aims to shared-memory multiprocessors, although their techniques can be applied to uniprocessor systems as well. They show performance improvement over OBL with their technique on three kernels. It will be interesting to see if the technique applies to more general applications.

Zhang and Torrellas [117] rely on compiler support, programmer feedback, or program directives to direct a prefetching scheme, called *memory binding and group prefetching*. A system call is inserted in the source code to specify a group. At run time, the system call marks the group in memory using two additional bits per memory line, an *N* bit and a *B* bit. The next memory line is in the same group when the *N* bit is set. The *B* bit is set on group boundary lines. When a line is accessed in the main memory, the remaining lines in the same group will be prefetched. Zhang and Torrellas also propose an additional bit, called the *P* bit, to support pointer prefetching. A *PrefetchLink* system call is used to set the *P* bit and build a link in a *group translation table*. An entry of the group translation table consists of the virtual address of the link and its physical address. An L2 miss in a group will trigger group prefetching. If the *P* bit of a line in the group is set, a prefetch on the linked group is started using the physical address retrieved from the group translation table. Using their prefetching, some of the irregular Splash-class applications [94] run 25-40% faster.

As an extension to the reference prediction table (RPT) [23], Chen later on [25] proposes a programmable prefetching engine. Chen's prefetching engine differs from the RPT in that the tag (PC), address, and stride information are supplied by the compiler rather than being dynamically detected. Before entering into a loop, entries are filled into the prefetch-

ing engine using a run-ahead instruction. The prefetching engine functions are much like the RPT once programmed.

Skeppstedt and Dubois [95] use the compiler to generate a trap handler to start the prefetch engine. The trap is triggered by an L2 miss and starts prefetching using the compiler-supplied information such as stride and count. This technique takes advantage of compiler support with less instruction overhead than software prefetching.

VanderWiel and Lilja [109] add a *Data Prefetch Controller* (DPC), an external general processor, to prefetch data from the Level 2 cache. DPC executes its own program, which is generated by the compiler through extracting reference streams from the original program. A producer-consumer relation is built between DPC and the main processor: DPC prefetches a new block into the Level 1 cache only after a previously prefetched block has been accessed by the processor. VanderWiel and Lilja show that this software/hardware cooperative technique outperforms pure software prefetching and pure hardware prefetching using a reference prediction table.

Karlson, Dahlgren, and Stenstrom [56] use a *prefetching array* to tolerate latencies in short linked data structures. They insert a number of jump pointers in each node to enable prefetching all possible nodes at the number of iterations ahead equal to the prefetch distance. The jump pointers are stored together as an array. The software approach, which needs one instruction for each array element, yields high instruction overhead. The hardware approach requires a new instruction specifying the base address and the length of the prefetching array. The instruction, when executed, will trigger the prefetching engine to prefetch on each address in the array.

Solihin, Lee, and Torrellas [99] use a User-Level Memory Thread (ULMT) running on a general processor in main memory, either in the memory controller chip or a DRAM chip. The thread performs correlation prefetching in a style similar to the Markov prefetcher proposed by Joseph and Grunwald [53]. The correlation table is just a simple memory structure. UMLT can be customized by the programmer or system on a per-application ba-

sis. Their approach achieves an average speedup of 1.32 for nine selected applications and a speedup of 1.46 when combined with a conventional processor-side sequential prefetcher.

Yang and Lebeck [114] add a programmable processor, a prefetching engine (PFE), at each level of the memory hierarchy. They use software to detect linked structures and generate code to feed the prefetch engines. The prefetch engines dereference the pointers and push the data to the upper level of the memory hierarchy.

Compared to other cooperative schemes, GRP combines the advantages of both software and hardware prefetching in a scheme that is simple yet effective. It conveys sophisticated compiler analysis results by associating a range of hints with loads, which an aggressive, simple, and general hardware prefetcher uses only when necessary. Thus, the pertinent compiler analysis is communicated to the hardware without requiring extensive static lookahead, software guarantees, or high instruction overhead.

2.3.4 Cache Replacement and Prefetching

There is limited research on combining cache replacement with prefetching. Lai et al. [66] propose a hardware-only technique to predict when a cache line is dead so new data can be prefetched into the line. We rely on software to do the prediction.

Evict-me takes an opposite approach as compared to hardware and software data prefetching, which tolerate latency [9, 54, 69, 82, 84]. Data prefetching tries to fetch data which will be used in the near future in order to reduce miss penalties. Evict-me tags instead predict which data will and will not be used in the near future, and keep the data in the cache that will be used. Evict-me tags do not bring new data into the cache and thus do not have the bandwidth and other overhead of prefetching. Prefetching pollutes caches when it brings in useless data. Evict-me tags can help alleviate the negative effects of hardware prefetching. In Chapter 5, we show that the combination of evict-me tags and hardware prefetching can further improve performance.

2.4 Other Cooperative Work

As we have discussed in this chapter, many researchers have investigated improving cache performance. The limited amount of hardware/software cooperative work is typically restricted by the hardware/software interface. A common method used in previous research is to add a few special instructions or a small code section to initiate some hardware actions or to pass static compiler knowledge [25, 42, 56, 117]. An extreme in this direction is software-managed cache as proposed by Moritz et al. [80, 81]. Some others use an independent program or thread to control prefetching hardware [99, 109, 114]. These mechanisms do not need hint bits in every memory instruction, but typically do not have the ability to adapt to the run-time states of memory accesses.

Through ISA extensions, we encode hints in every memory instruction and provide a method to make hardware/software collaboration systematic and effective. The compiler hints we use are mostly *lazy*, i.e., their actions are adaptable and dependent on run-time hardware states. We emphasize interactions between software and hardware. To achieve this, the hints need to show both *directivity* and *adaptability*. By directivity, we mean that the hints tell the hardware what to do. For example, a load marked as *evict-me* denotes a preferred eviction of the cache line that the load accesses. By adaptability, the hints sometimes need to be subordinate to other run-time actions. For example, a load marked as *spatial* triggers a region prefetching in GRP only when the load causes an L2 miss.

Only a few researchers take approaches very similar to ours. IA-64 uses compiler hints to direct loads to temporal or spatial cache structures [38]. To our knowledge, we haven't seen any effective compiler and hardware to support this design. To save energy, Unsal et al. [108] mark scalar loads and drive them to a separate small cache. They later on add more compiler control by encoding a *hot line register* index into memory instruction [105, 106]. A hit to the hot line registers will enable extraction of a way index to address directly a cache line in a set-associative cache and thus save energy by avoiding tag array lookup. This work targets single-level cache in embedded systems. Witchel et al. [110] describe a

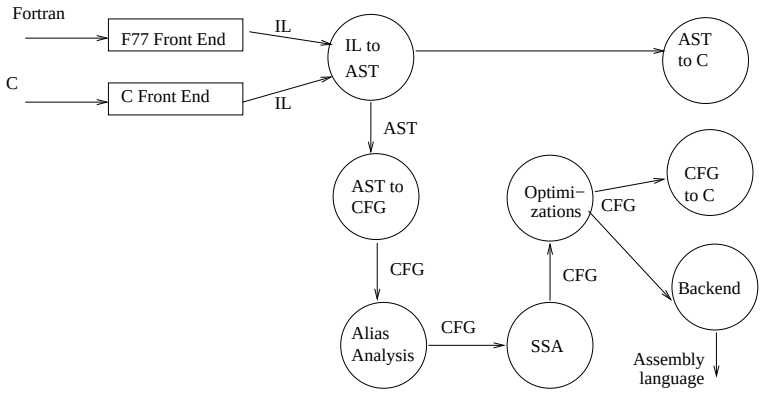

Figure 2.3. Scale data flow diagram

similar technique. Ashok et al. [7] extend Unsal et al.'s technique [105] to support complex program structures, different levels of speculation, and multi-level memory systems. Unsal et al. [107] rely on the compiler to predict IPC (instructions per cycle) and mark the instructions at which the IPC estimation is low. They observe energy savings by throttling the issue logic on marked instructions. Similar to our approach, all these work uses the compiler to generate hints and need a small amount of hint bits in memory instructions. The results of these work, including ours, show the importance and effectiveness of software/hardware cooperation.

2.5 Scale Compiler Infrastructure

Now we introduce *Scale*, the compiler infrastructure in which we implement all our compiler algorithms. Scale is developed by the Architecture and Language Implementation Laboratory of the Department of Computer Science of the University of Massachusetts, Amherst, and the University of Texas at Austin. Figure 2.3 illustrates data flows in Scale, where the nodes show actions or compilation phases and the edges show inputs and outputs of the nodes.

A source program in C or Fortran is first transformed to an intermediate language using the EDG Front End developed by the Edison Design Group. The front end completes syntax and semantic checking, including complete error checking. Diagnostics always

display the source line with a caret indicating the exact position of the error. The front end translates source programs into a high-level, tree-structured, in-memory intermediate language. The intermediate language preserves a great deal of source information (e.g., line numbers, column numbers, original types, original names), which is helpful in generating symbolic debugging information. Implicit operations in the source are made explicit in the intermediate language, but constructs are not otherwise added, removed, or reordered. The intermediate language is not machine dependent (e.g., it does not specify registers or dictate the layout of stack frames).

The in-memory intermediate language is transformed into a Clef (Common Language Encoding Form) abstract syntax tree (AST). The Clef AST consists of nodes that are *types* such as "integer" and "real", *declarations* such as variables and procedures, *expressions* such as arithmetic operations and allocations, and *statements* such as if-then-else and loop. High-level optimizations and annotations, such as loop unrolling and inlining, can be applied to the Clef AST.

Along with control-flow analysis and lowering, Scale transforms the Clef AST to a high-level and low-level mixed intermediate representation, called *Scribble*. This transformation lowers most arithmetic operations to a set of binary or unary operations. At the same time, it keeps some high level representation to facilitate further compiler analyses and optimizations. For instance, an array reference is represented in its high-level form, where all its shape and subscripts are still maintained, as well as its low-level form, which consists of a set of statements to calculate the reference address. The high-level form is critical for dependence analysis, which serves as a basis for many other transformations and analyses.

Scale performs an alias analysis on Scribble based on Steensgard's pointer analysis technique [101]. It then transforms the Scribble form with alias annotations into the *Single Static Assignment* (SSA) form, where each use of a variable is reached from only one definition.

We apply dependence testing and a set of optimizations on Scribble. Our dependence testing is based on the Omega library [86, 104]; we use its algorithms and interfaces. Scale currently supports sparse conditional constant propagation, copy propagation, partial redundancy elimination, global value numbering, scalar replacement, loop permutation, loop unrolling, and loop inlining.

Scale has a back end that targets the SPARC V8, Alpha, and Trips [6] ISAs. The back end includes register allocation, code generation, and some assembly level optimizations such as peep hole optimization.

All of the compiler algorithms discussed in this thesis are implemented in Scale, mainly in one optimization phase and the back-end code generator. Typically, compiler analysis on Scribble form generates hints annotated via Scribble nodes. The back end interprets the annotations and encodes the hints into assembly instructions. We rely on the native assembler and linker to generate executable binaries. We then use simulators to interpret the compiler hints.

CHAPTER 3

COMPILER-GUIDED CACHE REPLACEMENT

This chapter describes how the architecture and the compiler can work together to improve cache replacement decisions. We use compiler hints to direct cache replacements and follow the hardware-only policy when hints are not available.

Typically, cache replacement policies rely on access history to decide which cache line to evict on a cache miss. These policies sometimes perform well because programs often exhibit good locality. However, it is unavoidable for such schemes to make poor choices frequently. Figure 3.1 gives an example. Notice that array B is accessed in nest 1 but not in nest 2. Whenever there is a cache miss in the first nest, we prefer to evict an element of array B because it will not be reused. However, LRU ranks items from least to most recently used, i.e., A, B, C. Assuming that the cache size is a little bigger than 2*N, LRU will evict most of A even in a fully associative cache. A better replacement algorithm keeps both A and C and reuses them in nest 2.

Without additional assistance, it is infeasible for the hardware to foresee future accesses and make optimal choices [10, 103]. In this chapter, we discuss a compiler-guided

```
SUBROUTINE  TEST(N)
INTEGER A(N), B(N), C(N)

DO 1 I = 1,N
  C(I) = A(I) + B(I)
ENDDO

DO 2 I = 1:N
  A(I) = C(I) * 5
ENDDO
END
```

Figure 3.1. A simple example

cache replacement policy. Our new compiler mechanism guides cache replacements by selectively predicting when data will or will not be reused. We encode the compile-time prediction into memory instructions. We develop a comparative model of locality that uses dependence and array section analysis to determine static locality patterns in a program. This locality information, which we formulate as *reuse levels*, is conveyed to the run time to direct cache replacement. We prove that the cache replacement algorithm using reuse levels will at least match LRU in hit rate. We then describe a 16-bit encoding of reuse levels, and a practical one-bit encoding called *evict-me*. We implement our compiler analysis in Scale and simulate our proposed architecture in URSIM and SimpleScalar. By applying the evict-me bit to both Level 1 and Level 2 caches, we observe up to 21% simulated performance improvements for current technology on a selection of scientific benchmarks, and 34% for a technology prediction for 5 years from now [3]. On average, we reduce simulated execution time by around 5% to 16% depending on the cache configuration. Our results show that our technique works together well with a victim cache although neither technique subsumes the other.

3.1 Problem Formulation

In this section, we briefly review cache replacement policies, cache organizations that exploit them, and ideal replacement algorithms. We introduce our reuse notation and then present a new compiler algorithm that predicts locality within a loop nest (*intra-nest*) and between loop nests (*inter-nest*).

3.1.1 Cache Replacement Policies

As we have discussed in Section 2.2.4, it is generally preferred to have a low associativity cache to attain both high hit rate and low hit cycle latency. For a low associativity cache, a good cache replacement policy is critical. Modern architectures typically rely on

one of two replacement policies: random and LRU. The random scheme evicts a random line from a cache set on a replacement.

LRU and its approximations are the most widely used replacement policy. On a cache miss, the least recently used line is evicted. LRU tries to keep the recently referenced data in cache and expects those data will be referenced again soon. In a real cache design, LRU can be implemented with a set of bits encoding the uses of the cache line in a set. Abstractly, we can treat a cache set as a stack whose bottom is the candidate for eviction. On a cache hit, the hit line is moved the top of the stack and the relative position of the rest lines are unchanged. On a cache miss, the bottom is evicted and a new line is push to the top of the stack.

3.1.2 Perfect Locality Information: Trace-based Replacement

To use locality to direct cache replacement, we need a quantitative representation. Consider the following quantitative definition of temporal locality [85]. The *temporal locality* of a data reference at time T is $TL = 1/(T_{next} - T)$, where T_{next} is the time of the next access to that particular address. We can similarly define spatial locality as follows. The *spatial locality* of a data reference at time T is $SL = 1/(T_{next} - T)$, where T_{next} is the time of the next access to the same cache block.

In this work, we assume the minimum unit of communication between main memory and the cache is a block: whenever any part of a block causes a miss, the architecture loads the entire block. Thus, in our model, temporal locality is a special case of spatial locality. If we know the temporal and spatial locality of each data reference in a program trace, then the optimal replacement algorithm replaces the data that has reuse furthest in the future, i.e., the data with the smallest value for SL [10]. Of course, computing TL and SL requires a complete trace, which is not available at run time and is impossible to know exactly via static program analysis. To control cache replacement explicitly, we need a new method to describe locality. In the following section, we introduce the notion of *reuse level*, which is

a measure that is comparative rather than absolute. We then show how to compute reuse levels using dependences.

3.1.3 Reuse Levels

Assume that we have a complete *trace* of a program: the series of memory references in the program in execution order, i.e., $b_{f(1)}$, $b_{f(2)}$,..., $b_{f(n)}$. The subscripts are the block addresses which determine the references. The block addresses of the references need not be distinct, of course. The *reuse level* is used to approximate the locality of each reference. Rather than describe a specific distance from the current reference to the next reference to the same block, reuse levels describe a range of time in which the next reference will occur. Formally, the locality of reference $b_{f(i)}, 1 \leq i \leq n$, is a set $s \in \mathcal{S}_n$, where
$\mathcal{S}_n = \{[n+1,+\infty]\} \cup \{[j,k] \mid 1 \leq j \leq k \leq n\}$ *and* $[j,k] = \{j, j+1, .., k\}$. Here n is the total number of memory references in the trace. Apparently, $[n+1,+\infty]$ is the range out of the bound so no reference is in this range.

1. If s is $[n+1,+\infty]$, then block $b_{f(i)}$ will not be referenced again after the i_{th} reference; i.e., $f(i) \neq f(l)$ for all l, $i < l \leq n$.

2. If s is $[j,k]$ for some j,k, $i < j \leq k \leq n$, then $\exists t$, $j \leq t \leq k$, such that the next reference to block $b_{f(i)}$ is the t_{th} reference in the trace, i.e., $f(i) = f(t)$, and $f(t) \neq f(l)$ for all l, $i < l < t$.

   Then we call the set s the *reuse level* of $b_{f(i)}$. Note that a reference can have multiple valid reuse levels as long as the conditions listed above are satisfied. To compare reuse levels for references, we define three relations on $\mathcal{S}_n : \prec, \sim$, and $\succ$.

$$
\begin{array}{llll}
[i,j] & \prec & [n+1,+\infty] & \text{for all} \quad 1 \leq i \leq j \leq n \\
[i,j] & \prec & [k,l] & \text{if} \quad j < k \\
[i,j] & \sim & [k,l] & \text{if} \quad [i,j] \cap [k,l] \neq \phi \\
[i,j] & \succ & [k,l] & \text{if} \quad [k,l] \prec [i,j]
\end{array}
$$

Theorem 1. Only one relation holds for any two elements in $\mathcal{S}_n$.

Proof. By definition. □

Theorem 1 shows that reuse levels are comparable. Intuitively, if two blocks conflict, we want to replace the block whose reuse level is $\succ$ than that of the other block. When two reuse levels are $\sim$ to each other, we use access history to break ties (as does LRU).

3.1.4 Using Dependences as Reuse Levels

This section explains how to combine dependences with the loop iteration space to produce reuse levels. We briefly introduce some basic concepts of data dependences and dependence testing. We then describe bounded regular sections and the locality graph construction using dependence testing and regular sections. We finally extract reuse levels from the locality graph.

Data Dependence and Dependence Testing

The theory of *data dependence* was originally developed for automatic vectorizers. It is applicable to a wide range of optimization problems such as parallelization and loop transformations. We say that a data dependence exists between two references, *R1* and *R2*, if they access the same location in memory. There are four types of data dependences [64]:

1. *True dependence* (read after write, RAW) occurs when *R1* writes a memory location that *R2* later reads.

2. *Anti dependence* (write after read, WAR) occurs when *R1* reads a memory location that *R2* later writes.

3. *Output dependence* (write after write, WAW) occurs when *R1* writes a memory location that *R2* later writes.

4. *Input dependence* (read after read, RAR) occurs when *R1* reads a memory location that *R2* later reads.

All the four types of dependences denote that a *reuse* of *R1* occurs at *R2*. Data dependences thus imply temporal locality where the two references of a dependence access the same location. It is an obvious extension to detect spatial locality by relaxing the condition from "same location" to "same cache line" or "adjacent location".

We are particularly interested in array references in loop nests, which account for most references in scientific applications. *Dependence testing* is the method used to detect whether dependences exist between two array references in a loop nest. Typically, we say that a *dependence* exists between two array references if they access the same array and the same element. The latter condition suggests the indices of the two references be equal to each other at each dimension. Formally, let α and β be vectors of n integer indices that correspond to n valid loop index values of an n-level loop nest. There is a dependence between two m-dimension array references $A(f_1(i_1,...,i_n),...,f_m(i_1,...,i_n))$ and $A(g_1(i_1,...,i_n),...,g_m(f_1,...,i_n))$ if $f_j(\alpha) = g_j(\beta)$ for all $1 \leq j \leq m$ and α is lexicographically less than or equal to β. The goal of dependence testing is to solve those equations to determine if two references are dependent or independent. A *distance* or a *direction vector* can be used to characterize a dependence. Given a dependence from iteration α to β, the distance vector is $\beta - \alpha$, and the direction vector $D = (d_1,...,d_n)$ is defined by the following equation:

$$d_j = \begin{cases} < & if \quad \alpha_j < \beta_j \\ = & if \quad \alpha_j = \beta_j \\ > & if \quad \alpha_j > \beta_j \end{cases}$$

Much research has been conducted on solving dependence equations and detecting dependences [41, 86]. In our compiler, we use the Omega test [86]. We use data dependences to construct the locality graph described later in this section. In the graph, a dependence vector serves as a *reuse vector*, which denotes the reuse of a reference. The graph is then used to generate reuse levels to direct cache replacement.

Bounded Regular Sections

We use the dependence testing as discussed above to detect reuses within a loop nest. To detect reuses between distinct loop nests, we use bounded regular sections [44] to describe the access range of a reference in a loop nest. The descriptors for bounded regular sections (BRSD) are vectors of elements, each of which is a triplet. A triplet describes an access range in a dimension, consisting of a lower bound, an upper bound, and a step (stride). The bounded regular section for A(I,J) in both loop nests in Figure 3.2 is $[2 : M-1 : 1, 2 : N-1 : 1]$. The descriptors support union and intersection operations. There is a reuse between two references in distinct nests if the intersection set of their BRSDs is not empty.

Locality Graphs

We build a *locality graph* based on reuse. The graph describes temporal and spatial locality within each loop nest and across loop nests. An edge connecting two references in the same loop nest has as its label the *reuse vector*, which is the dependence vector of the dependence between the two references. An edge connecting two references in distinct loops has as its label the intersection of the two BRSDs and the reuse vector. Figure 3.3 shows the locality graph for the sample program in Figure 3.2, where for simplicity we omit B(I+1,J) and B(I,J+1). In Figure 3.3, the first element of a reuse vector denotes the inter-nest reuse direction. If it is '$=$', the reuse is in the same nest. If it is '$<$'. then the dependence is inter-nest. Now the vector $(=, <, >)$ from B(I-1,J) to B(I,J-1) denotes an intra-nest input dependence and a temporal reuse across the J loop.

Reuse Levels

We can rely on reuse vectors as predictors of access patterns. We can use those vectors as reuse levels if we also add information that describes the relative position between independent references. We can either track the loop iterations at run time or dynamically keep the reuse levels up to date as different instances of a reference execute. Now a reuse level is a set of loop iteration points, which consist of run-time memory references. For example, we can use the direction vectors shown in Figure 3.3 as reuse levels with the following

```
PROGRAM  SimplifiedJacobi
PARAMETER (N=1000, M=1000)
REAL A(N, M), B(N, M)

DO  J = 2 , N-1
  DO  I = 2, M-1
    A(I, J) = (B(I-1, J)+B(I+1, J)+B(I, J-1)+B(I,J+1))/4
  ENDDO
ENDDO

DO J = 2, N-1
  DO I = 2, M-1
    B(I, J) = A(I, J)
  ENDDO
ENDDO
END
```

Figure 3.2. Another sample program

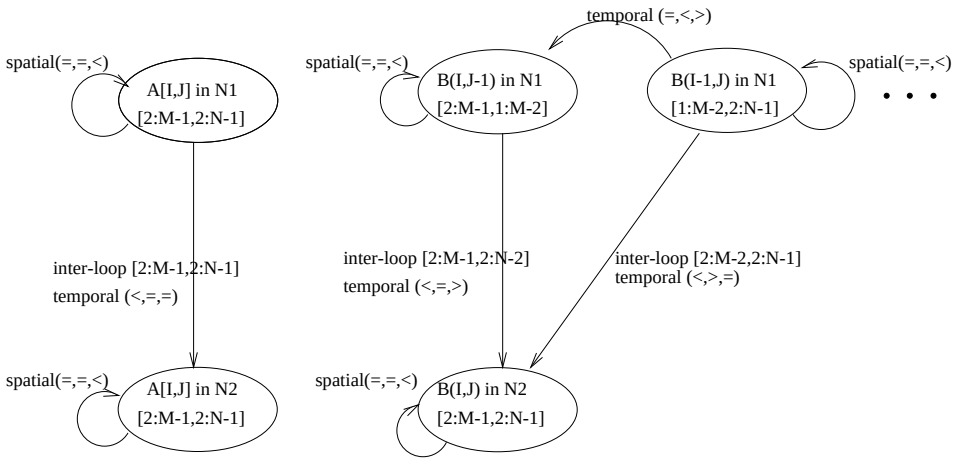

Figure 3.3. Locality graph

semantics. The B(I,J-1) has a self spatial reuse with direction vector $(=,=,<)$. We say a reference has a *self spatial* reuse if the same reference accesses the adjacent memory location in the future. The direction vector by itself means there is a spatial reuse due to a later reference to B(I,J-1) itself in the same nest, the same J iteration, but a later I iteration. As a reuse level, the direction vector means the iteration points from the next I iteration through I=M-1. Specifically, given the loop iteration at I=5 and J=4, the run-time instance of reference B(I,J-1) is B(5,3), whose reuse level $(=,=,<)$ means it has a reuse between iteration I=6 and iteration I=M-1 under J=4. To illustrate our idea, let's ignore the spatial reuse

vector of B(I-1,J). Now reference B(I-1,J) has only a temporal reuse with vector $(=,<,>)$ which means the reuse is in later J iterations. Note that $(=,=,<) \prec (=,<,>)$ because $(=,=,<)$ suggests a reuse in the same J iteration. So when B(5,3) and B(4,4) conflict, our cache replacement policy will choose the cache line of B(4,4) to evict. We now describe cache replacement algorithms that use reuse levels.

3.2 Cache Replacement Algorithms

In this section, we show how to improve cache replacement decisions in an ideal case and then within the context of realistic cache organizations. First, we develop a general framework that is guaranteed to match or improve hit rates over LRU given sufficient hardware support. We then present a simple but practical one-bit encoding, called the *evict-me* bit, that indicates when a cache block is a good choice for replacement.

3.2.1 Improving LRU Cache Replacement

Our first cache replacement algorithm, the Prediction algorithm, uses the access order of a reference and its reuse level to direct replacement. Consider a program trace $b_{f(1)}^{<s_1,1>}$, $b_{f(2)}^{<s_2,2>}$,..., $b_{f(n)}^{<s_n,n>}$, where $b_{f(i)}$ is the i_{th} block accessed by address $f(i)$, and $< s_i, i >$ are its reuse level and access order respectively. We define a relation $\lhd$ on the set $Q_n = \{< s_i, i > , s_i \in \mathcal{S}_n, 1 \leq i \leq n\}$, as follows:

$< s_i, i > \lhd < s_j, j >$ if $(s_i \prec s_j)$ or $(s_i \sim s_j$ and $i > j)$.

Each <*reuse level*, *order*> pair is an element of Q_n. Note that the second condition of the definition follows the LRU cache replacement policy, which evicts the line the the smallest access order..

Theorem 2. For each pair of elements in Q_n, $< s_i, i >$ and $< s_j, j >$, $i \neq j$, either $< s_i, i > \lhd < s_j, j >$ or $< s_j, j > \lhd < s_i, i >$.

Proof. By definition.□

Step/order		0	1	2	3
PREDICTION	block 1		r1 $<[3,4],1>$	r1 $<[3,4],1>$	r1 $<[3,4],1>$
	block 2			r2 $<[5,6],2>$	r3 $<[5,6],3>$
	miss/hit		miss	miss	miss
LRU	block 1		r1	r1	r3
	block 2			r2	r2
	miss/hit		miss	miss	miss

Step/order		4	5	6
PREDICTION	block 1	r1 $<[21,28],4>$	r2 $<[10,12],5>$	r2 $<[10,12],5>$
	block 2	r3 $<[5,6],3>$	r3 $<[5,6],3>$	r3 $<[10,12],6>$
	miss/hit	hit	miss	hit
LRU	block 1	r3	r2	r2
	block 2	r1	r1	r3
	miss/hit	miss	miss	miss

Table 3.1. LRU versus Prediction for a 2-way set-associative cache

The Prediction algorithm updates a reference's order and its reuse level in the cache on every access. Think of a cache set as an ordered list from smallest to largest by the $\triangleleft$ ordering of the <reuse level, order> pairs. Initially every reuse level is $[n+1,\infty]$, and on a reference, the architecture sets the reuse level if it is specified. Whenever there is a miss, the Prediction algorithm choose to replace the last line with the largest <reuse level, order> pair. When a reference changes the cache line's <reuse level, order> pair, we change its position in the list. We compare it to the other items in the list from first to last until the $\triangleleft$ ordering of the line is smaller than that of the next element, and then insert the line before this next element. Although the $\triangleleft$ ordering is not a partial order (because it is not transitive), the definition of the Prediction algorithm and the list ordering algorithm guarantees that there is a deterministic ordering of the list after each cache access; i.e., Theorem 1 and 2 are sufficient to ensure that the Prediction algorithm is totally specified.

The following example illustrates the algorithm. Assume a two-way set associative cache and a simple program trace $a_{r1}^{<[3,4],1>}$, $a_{r2}^{<[5,6],2>}$, $a_{r3}^{<[5,6],3>}$, $a_{r1}^{<[21,28],4>}$, $a_{r2}^{<[10,12],5>}$, $a_{r3}^{<[10,12],6>}$, ..., all of whose elements are mapped into a single cache set. Here r1, r2, and r3 are references to distinct blocks in main memory. The content of the cache is shown in Table 3.1. In step 3, LRU replaces r1, which leads to a miss in step 4. However, since

$< [3,4],1 > \lhd < [5,6],2 >$, the Prediction algorithm replaces r2 instead. In this example, it performs better than LRU.

Theorem 3. For the same cache configuration (same cache size, same degree of associativity, and same block size), at each reference point, if there is an LRU hit, there is also a Prediction hit.

Proof of Theorem 3. The proof is based on the trace we defined at the beginning of Section 3.2.1.

Say that we are working on a *w*-way set associative cache. Assume, for contradiction, that at reference $b_{f(i)}^{<s_i,i>}$ there is a miss for the Prediction algorithm and a hit for LRU. Let $b_{f(j)}^{<s_j,j>}$ be the nearest reference to the same block address where $j < i$. We know that $f(i) = f(j)$. In the following proof we let t_i denote the time when the i_{th} reference is accessed and assume that after time Δt, the access completes. We have $t_i + \Delta t < t_{i+1}$ for all i. We first show that there are no more than w distinct references between time t_j and time t_i. We then show that if reference $f(i)$ is not a Prediction hit at time t_i, all references in the set when $f(j)$ is evicted are accessed between time t_j and time t_i. This introduces contradiction since now there are at least $w+1$ distinct references between time t_j and time t_i considering the reference causing the eviction of reference $f(j)$.

Claim 1: There are no more than w distinct references mapped into the same set between $b_{f(j)}^{<s_j,j>}$ and $b_{f(i)}^{<s_i,i>}$ inclusive.

To simplify the discussion, assume that each block in a set is aged from 1 to w by the access order. The block with the smallest order value has age w, the one with the largest order value has age 1. With the LRU algorithm, the block with age w is evicted when there is a miss. At the time when $b_{f(j)}^{<s_j,j>}$ is brought into the cache, its age is 1. Assume for contradiction that at least w distinct references different from $b_{f(j)}$ between $b_{f(j)}^{<s_j,j>}$ and $b_{f(i)}^{<s_i,i>}$ are mapped into the same set. All those w references are older than $b_{f(j)}$, so each reference will increase the age of $b_{f(j)}$ by 1. Thus, when we access the $w-1_{th}$ reference, $b_{f(j)}$ has age w. The access to the w_{th} reference will evict $b_{f(j)}$, and no reference will bring

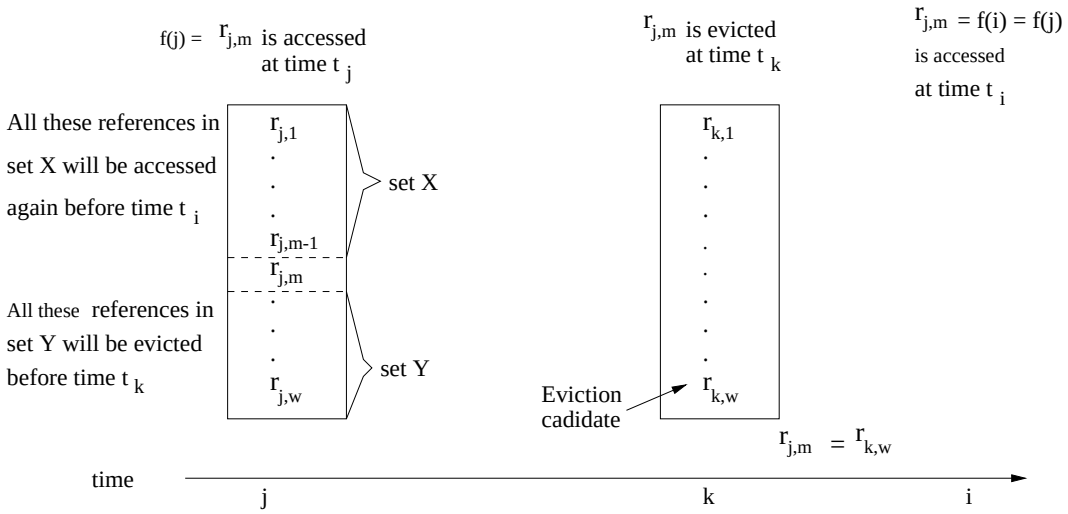

Figure 3.4. Proof of claim 2 (1. Claim 1 shows that there are no more than *w* distinct references between time t_j and time t_i. 2. Claim 2 shows that if reference $f(i)$ at time t_i is not a hit for the Prediction algorithm, then at time t_k when $f(j) = f(i)$ is evicted, all references in the cache set should be accessed at least once between time t_j and t_i. (a) If a reference in the cache set at time t_k was in set X, it must be accessed again before time t_i because it has a reuse level less than that of $f(j)$. (b) if a reference in the cache set at time t_k was not in X, then it must be recently accessed after time t_j. Otherwise, it must be in set Y and should be evicted before the eviction of $f(j)$ at time t_k. It will then not appear in the cache set at time t_k.)

$b_{f(j)}$ back because $b_{f(i)}^{<s_i,i>}$ is the most recent reference to block $b_{f(j)}$. This contradicts the LRU hit at $b_{f(i)}^{<s_i,i>}$.

Next assume that there is an age between 1 and *w* associated with each block in the list defined for the Prediction cache in Section 3.2.1. The ages of the blocks are consistent with the ordering of the list. The $< reuse\ level, order >$ pair of the block at age 1 is smaller in $\lhd$ ordering than the pair of the next block in the list, and so on.

Say now that $b_{f(j)}$ has age *m* at time $t_j + \Delta t$. Because we have a miss of $b_{f(i)}^{<s_i,i>}$ at time t_i, there exists a reference $b_{f(k)}^{<s_k,k>}$ at time t_k, for some $j < k < i$, which is also a miss and $b_{f(j)}$ has age *w* when the reference $b_{f(k-1)}^{<s_{k-1},k-1>}$ completes.

Claim 2: All addresses in the cache set at time $t_{k-1} + \Delta t$ are referenced at least once between time t_j and t_i.

We sketch the following proof in Figure 3.4 where the two boxes show the cache set content and the cache lines are ranked by their ages with the oldest as the candidate for the next eviction.

Let $r_{j,1}, r_{j,2}, ..., r_{j,m-1}, r_{j,m}, .., r_{j,w}$ be the block addresses in the cache set at time $t_j + \Delta t$ and $r_{k,1}, r_{k,2}, ..., r_{k,w}$ be those at time $t_{k-1} + \Delta t$, where the second subscript denotes the age of the corresponding address. We know that $r_{j,m} = r_{k,w} = f(j)$. Let $U = \{r_{j,1}, r_{j,2}, ..., r_{j,m-1}\} \cap \{r_{k,1}, r_{k,2}, ..., ., r_{k,w}\}$.

First, all addresses in $S = \{r_{k,1}, r_{k,2}, ..., ., r_{k,w}\} - U$ must be accessed between time t_j and $t_{k-1} + \Delta t$. Assume, for contradiction, that $r_l \in S$ is not accessed during this period. Then $r_l \in S$ must be accessed before time t_j, and must be in the cache set at time t_j since it is in the cache set at time $t_{k-1} + \Delta t$. Then since r_l is not in set $X = \{r_{j,1}, r_{j,2}, ..., r_{j,m-1}\}$ because it is in S, it has an older age than that of $b_{f(j)}$ at time $t_j + \Delta t$. No reference can change this relationship unless the two references themselves are accessed again between time $t_j + \Delta t$ and t_{k-1}. Note that $b_{f(j)}$ has age w at time $t_{k-1} + \Delta t$. r_l must be evicted before time t_{k-1} since it is older. Then it cannot be in the set at time $t_{k-1} + \Delta t$, contrary to the assumption.

Second, all references in U must be accessed between time t_j and t_i. Notice that all references in U have $< reuse\,level, order >$ pairs $\lhd$ than that of $b_{f(j)}$ at time $t_j + \Delta t$ just after $b_{f(j)}$ is brought into cache. Since the orders of these references are less than j, by the definition of relation $\lhd$, they must have smaller reuse levels which means they will be accessed before the next reference to $b_{f(j)}$, which occurs at time t_i.

The references in the w-block set at time $t_{k-1} + \Delta t$ are distinct. Furthermore, the reference $b_{f(k)}$ is distinct from the blocks in the set since it is a miss. The total number of distinct references mapped into the set between time t_j and t_i are at least $w+1$. Contradiction. □

Theorem 3 tells us that the Prediction algorithm is at least as good as the LRU algorithm at any reference point. So if we can find a reuse level for each reference point, we expect to

improve upon the LRU algorithm. In Section 3.1.4 we have shown that dependence vectors combined with loop iterations can predict reuse distances.

3.2.2 16-Bit Encoding

The Prediction algorithm makes replacement decisions based on reuse levels. In Section 3.1.4 we discussed how to obtain reuse levels for each array reference. To use reuse levels at run time, we assume an extended ISA that has extra bits for setting the reuse levels in each cache line on loads and stores, and a cache that supports these bits (we discuss other implementation options in Section 3.3). The Prediction algorithm sets corresponding cache tag bits when a memory instruction with tag bits is executed. The algorithm then chooses a line for eviction based on the value of the cache tag bits and LRU history bits. The 16-bit method we describe here is not practical for an implementation but lets us explore more fully the accuracy of our reuse information and encoding. We encode here the reuses in each loop nest and the reuses between two adjacent nests. For inter-nest reuse, we consider only adjacent loops, because most inter-nest misses occur between two adjacent nests [79]. We use a 16-bit annotation because the simulator we use supports at most a 16-bit annotation. This encoding gives us a loose upper bound on our technique. Looking for a better encoding is left to future work. Figure 3.6 shows our algorithm for computing these bits. Table 3.2 lists the function of each bit, where bit 15 is the most significant bit. The encoding assumes that the deepest level of a loop nest is 4, which is appropriate for most applications. We assume that for each routine there is a virtual loop enclosing the whole routine. The virtual loop is at level 0. For each level, we have a spatial bit and a temporal bit. The bit for a reference at loop L is set if the reference has reuse across iterations or reuse in the current iteration. Bit 0 is set when the compiler can determine the reuse levels of a reference. Bits 5 through 1 are reserved for reference step, which we will define later.

The temporal bit of loop level i also functions as an inter-nest temporal reuse bit for a nest whose outermost loop is at level $i+1$. Consider loop L at level l whose loop body

Bit	Function
15	temporal bit for level 4
14	spatial bit for level 4
13	temporal bit for level 3 (inter-nest bit for level 4)
12	spatial bit for level 3
11	temporal bit for level 2 (inter-nest bit for level 3)
10	spatial bit for level 2
9	temporal bit for level 1 (inter-nest bit for level 2)
8	spatial bit for level 1
7	temporal bit for level 0 (inter-nest bit for level 1)
6	spatial bit for loop level 0
5	sign of reference step (1:negative, 0: positive)
4-1	reference step
0	reuse level tag

Table 3.2. Encoding for 16-bit reuse level

```
L:  DO i = l:u:s
N1:   DO i1 = l1:u1:s1
        ... R ...
      ENDDO

N2:  DO i2 = l2:u2:s2
        ... R ...
     ENDDO
ENDDO
```

Figure 3.5. A sample loop nest

consists of two nests, $N1$ and $N2$, as shown in Figure 3.5. Reference R in $N1$ has an inter-nest reuse in $N2$. Its inter-nest bit for level $l+1$ is set because the outermost loop of nest $N1$ is at level $l+1$. The same bit also serves as the temporal bit for level l, which means a reference has reuse in the the current or future L iterations. These semantics are the same as the semantics of the inter-nest bit, which means the reuse is in the current iteration.

Figure 3.7 shows the program in Figure 3.2 with 16-bit reuse levels, which are listed in hexadecimal form. We use reference B(I+1, J)$_{<0X0e83>}$ as an example to explain our algorithm. B(I+1,J) has both spatial and temporal reuse at loop level 2 (the I loop), so the 10_{th} and 11_{th} bits of its reuse level are set to 1. B(I+1,J) has temporal reuse at loop level 1 (the J loop), so the 9_{th} bit is set to 1. It also has inter-nest reuse and the outermost loop of

```
reuseLevelGeneration()
{
  for each perfect nest whose outermost loop is at level j {
    for each array reference r in the nest {
      if (r is not in the locality graph)
        continue;

      reuseLevel = 0;
      for each loop at level i enclosing the reference r {
        if r has temporal reuse across the loop iterations
          /* including the current iteration */
          bit (6+2*i) of reuseLevel = 1;

        if r has spatial reuse across the loop iterations
          /* including the current iteration */
          bit (5+2*i) of reuseLevel = 1;

        if r has inter-nest reuse then
          bit (4+2*i) of reuseLevel = 1;
      }
      bits 5-1 of reuseLevel = referenceStep(r);
      bit 0 of reuseLevel = 1;
      reuse level of r = reuseLevel;
    }
  }
}
```

Figure 3.6. 16-bit reuse-level generation

the nest is at level 1, so the 7_{th} bit is set to 1. Bit 0 is set because compiler knows all reuses of the reference.

Note that in our encoding, we try to put all reuse levels of a reference together. A static reference usually has different reuse levels for different loops or nests. For instance, in Figure 3.2, A(I,J) in nest 1 has spatial reuse across the I loop. It also has inter-nest temporal reuse. We need two reuse levels for the reference. At run time, we should always first use the smallest reuse level in the $\prec$ ordering. In fact, this mechanism is implicitly shown in our encoding where we assign more significant bits to deeper loops.

Now the Prediction algorithm can make cache replacement decisions based simply on the values of reuse levels associated with each cache line: it always evicts the cache line with the smallest reuse level. A special case is that when the reuse level of a cache line is 0, the eviction is based on its access order as in LRU. Here our implementation is more

```
PROGRAM SimplifiedJacobi
PARAMETER (N=1000, M=1000)
REAL A(M, N), B(M, N)

DO  J = 2 , N-1
  DO  I = 2, M-1
    A(I, J)<0X0483> = (B(I-1, J)<0X0683>+B(I+1, J)<0X0e83>+
                       B(I, J-1)<0X0483>+B(I,J+1)<0X0683>)/4
  ENDDO
  call update(2)
ENDDO
call update(1)
DO J = 2, N-1
  DO I = 2, M-1
    B(I, J)<0X0403> = A(I, J)<0X0403>
  ENDDO
  call update(2)
ENDDO
call update(1)
END
```

Figure 3.7. Reuse levels for the sample program

aggressive than the formal definition of the Prediction algorithm in Section 3.1. In the implementation, we keep each cache set in its LRU ordering. When there is a miss in a set, the last line of the set is replaced if its reuse level is 0, which means that compiler does not know its reuse level. Otherwise, the algorithm chooses the line with the smallest reuse level.

The reuse level of a reference usually does not span the whole sub-space of the reference, particularly for spatial reuses. In the locality graph shown in Figure 3.3, given, for example, a cache line size of two words and A(1,J) is aligned on the cache line for all J, we notice that spatial reuse of A(I,J) occurs only when I is odd. If we know the starting address of each array reference, loop unrolling can produce array references with and without spatial locality. Our implementation uses another method to resolve this problem. At run time, we know the cache block size and exactly where a memory block will be mapped into a cache line. This information and the access pattern of an array reference are usually enough to decide if there is spatial reuse. For example, for A(I,J) we just mentioned, we know the next access to array A is A(I+1,J). Hence we can be sure that in the reference A(I,J) will

```
update(int l)
{
  for all reuse levels associated with each cache block {
    remove those predicting reuses for level l;
  }
} /* end update */
```

Figure 3.8. Update function

have no spatial reuse across the I loop if it is mapped into the last word of a cache line. For a given array reference whose indices are all affine expressions, the compiler discovers the pattern of spatial reuse and encodes it into the corresponding instruction. Formally, we consider only arrays with the least significant index in the form of $a*I+b$, where I is the loop induction variable, and a and b are constants. We also assume that the loop step of the induction variable I is a constant s. Let p be the word position of the reference in the cache line, l be the cache line size, e be the element size in number of words, and $a*s$ be the *reference step*. If $a*s$ is positive, the reference has self-spatial reuse when $p < l-a*s*e$. If $p > -a*s*e$ and $a*s$ is negative, the reference also has self-spatial reuse. Similar techniques resolve group-spatial reuse. A reference has *group-spatial* reuse if there exists another reference that accesses an adjacent location later. The reference step is encoded into reuse levels using bits 5 through 1. Obviously, in Figure 3.7, B(I+1,J)'s reference step is positive and its value is 1, so bits 5 to 1 are set to value 1. In other words, bits 5 to 2 are set to zeros and bit 1 is set to 1.

In our implementation, we insert an update() function as shown in Figure 3.8 at the exit of each loop. The function expires those reuse levels that are no longer valid. The update() function in Figure 3.8 can help to reduce the side effects of misprediction for both spatial and temporal reuse, because it expires the prediction of reuse in a loop after the execution of the loop. If only a small percentage of predictions are not correct, they will expire sooner or later, and will not affect the miss rate too much. In the example code, we notice that temporal reuse of B(I-1,J) exists for all J except J=N-1. The mispredictions at J=N-1 are insignificant and the update() function between the two nests will expire them.

For our encoding, the implementation of update(l) needs to set the temporal and spatial bit at level l to 0. In Figure 3.7, note that after the I loop finishes its execution, update(2) is executed, which sets bit 10 and bit 11 of all reuse levels to 0. It expires the prediction of reuses across the iterations of loop I. Similarly, update(1) expires the prediction of reuses across the iterations of loop J in the first nest, but the prediction of inter-nest reuses is kept alive.

3.2.3 Evict-me: 1-Bit Encoding

The Prediction algorithm can be implemented by encoding reuse levels into memory instructions. In Section 3.2.2, we discussed a 16-bit encoding, which serves as a useful upper bound on compiler accuracy. However, using 16 auxiliary bits for each cache line will increase the time to determine which line to replace and may consume too much area. For an 8K Level 1 cache with 32-byte cache lines, 16 extra bits would contribute about 5% to the cache area.

There are two ways to address these problems. One is to implement the policy in lower-level caches, where the cost of extra reuse level bits and the comparison latency are relatively low. For example, a 256K Level 2 cache with 128-byte cache lines only needs to devote 1.5% additional area to annotations. The other way is to simplify the model. A 16-bit encoding implies up to 2^{16} reuse levels. The evict-me tag denotes two reuse levels, s_1 (no reuse) and s_0 (reuse). We combine it with LRU bits as we discussed in Section 3.2.1. The one-bit Prediction algorithm acts as follows. If the evict-me bit of a block is set, the replacement algorithm will choose that block to replace on a miss. Otherwise, it follows the LRU policy. The compiler generates special-purpose instructions to set evict-me bits and thus explicitly to control cache replacement.

This one-bit encoding suggests that we classify reuse distances into two levels, such that a distance vector in one level is always less than one in the other level. A simple and very conservative algorithm tags these array references that have no locality in a loop nest and

are not reused in any following nest. Assume the total number of run-time memory accesses in a routine of a nest is n. In the nest, the algorithm uses two reuse levels, $s_0 = [1, n]$ for references with reuses in this nest or subsequent nests, and $s_1 = [n+1, +\infty]$ for references with no reuses in the same subroutine. Following the definition in Section 3.1, we have $[1, n] \prec [n+1, +\infty]$. A more aggressive algorithm follows Theorem 4.

Theorem 4. In a w-way set-associative cache, if the number of distinct references mapped into the same set between a reference and its reuse is greater than w, then evicting the first reference in the next replacement will not degrade the overall LRU hit rate.

Proof of Theorem 4. Let's say we are working on a w-way set associative cache and a program trace $\mathcal{P}$. We focus on a specific cache set $\mathcal{C}$. Assume that sub-trace $b^{s_1}_{f(1)}, b^{s_2}_{f(2)}, \ldots, b^{s_n}_{f(n)}$ is the largest subset of $\mathcal{P}$ mapped into set $\mathcal{C}$ in its original order. $b_{f(i)}$ is the i_{th} block mapped into $\mathcal{C}$, and its block address is $f(i)$. s_i is the evict-me tag going with the access. In particular, $s_i = 0$ means it is a regular access; $s_i = 1$ means the block's evict-me tag gets set after this access. The evict-me tag of a block is set only when its next reuse is more than w distinct references away, inclusive.

Now we prove that for any access $b^{s_j}_{f(j)}$ in the sub-trace, if LRU results in a hit, then there is a hit for evict-me at this access. Assume that we get an LRU hit at $b^{s_j}_{f(j)}$. Let access $b^{s_i}_{f(i)}$ be the closest reference to block $b_{f(j)}$ where $i < j$. Since it is an LRU hit, we know there are no more than w distinct references in the sub-trace between $b^{s_i}_{f(i)}$ and $b^{s_j}_{f(j)}$ (we showed this in the proof of Theorem 3). Now $s_i = 0$ follows from the evict-me tag assignment condition. Since $s_i = 0$, the evict-me algorithm can at most increase the age of $b_{f(i)}$ by 1 at each following reference. So at access $b^{s_{j-1}}_{f(j-1)}$, the age of $b_{f(i)}$ should be less than w. The evict-me algorithm thus leads to a hit at $b^{s_j}_{f(j)}$. □

We can design an algorithm which sets the evict-me tag when accessing a reference without reuse, or with reuse if it is sufficiently far away. Accurately counting the number of distinct references mapped to a specific cache set is impossible at compile time when the iteration counts and sizes of arrays are unknown. Now, if we can determine the total

data volume between a reference and its reuse across loop nests, and it is greater than twice the cache size, then we predict it will not be reused; i.e., that the number of distinct references mapped into a set between the two references will be greater than the degree of associativity. This intuition implies that Theorem 4 holds.

We estimate these sizes at compile time. If the loop bounds of the nest are all constants and available at compile time, we combine them with the BRSDs to compute the exact data volume. Figure 3.9 shows our algorithm. The pseudo-code computes data volume in a loop nest. The data volume of a nest is the total size of the distinct array elements in the nest. The algorithm first unions all regular sections of each array and then sums the volume of the union.

When the loop bounds of a nest are unknown, we use a simple heuristic that assumes that the data volume of a nest is greater than two times the Level 1 cache size if it contains more than one level of loop nesting.

Following Theorem 4, we can use the cache size as a bound for reuse levels. With the single evict-me bit, we have two reuse levels, [1, cache size] and [cache size + 1, ∞]. We'd like to evict the line whose reuse distance is greater than the cache size because even a fully associative cache cannot exploit the reuse. Our heuristic of using data volume of twice the cache size is derived from this intuition. We use twice the cache size with consideration that the volume of the evict-me lines is also included. A better algorithm would calculate only the total volume of non-evict-me references.

Figure 3.10 presents a more aggressive algorithm for singling out references without temporal or spatial reuse in a nest. It never marks references with temporal intra-nest reuse. It sets the evict-me bit for those references whose reuse spans more than two times the cache size, or when the data volume is unknown, whose nesting depth is 2 or more, or if the reference has no temporal reuse with the adjacent nest. If the reference has spatial locality at any loop level, the compiler still marks it as evict-me, such that the architecture will exploit it before marking it for eviction. In the program in Figure 3.2, we set the

```
int computeVolume(Loop l)
{
  int volume = 0;
  /* first compute regular sections for each reference */
  for (each array reference A in the loop) {
      compute A's bounded regular section;
  }

  /* estimate total volume */
  for (each array A accessed in the loop) {
     U = union of all regular sections of the references to the array;
     volume += volumeOfRegularSection(A, U) * elementSize(A);
  }

  return volume;
}

int volumeOfRegularSection(Array A, RegularSection r)
{
  /* let r = (l[1]:u[1]:s[1], ..., l[n]:u[n]:s[n]) */
  /* dimensions of array A are (d[1], d[2], .., d[n])
     and array A is in row-major order */

  int volume = 1;
  for (i=1; i<=n; i++) {
    volume *= (u[i]-l[i]+1)/s[i];
  }

  return volume;
}

RegularSection Union(RegularSection r1, RegularSection r2)
{
  /* let r1 = (l1[1]:u1[1]:s1[1], ..., l1[n]:u1[n]:s1[n]) */
  /* let r2 = (l2[1]:u2[1]:s2[1], ..., l2[n]:u2[n]:s2[n]) */
  RegularSection r =  (l[1]:u[1]:s[1], ..., l[n]:u[n]:s[n]);

  for (i=1; i<=n; i++) {
    l[i] = min(l1[i], l2[i]);
    u[i] = max(l1[i], l2[i]);
    s[i] = gcd(s1[i], s2[i]); /* gcd is the greatest common divisor */
  }

  return r;
}
```

Figure 3.9. Algorithms for computing data volume in a nest

evict-me tags of A(I,J) in both nests, the tag of B(I,J-1) in nest 1, and that of B(I,J) in nest 2, because these four references have no temporal reuses and the total data volume

```
setEvictMeTag()
{
  for each loop nest {
    compute nest volume;
    for each array reference r in the nest {
      if (r has no temporal reuse in this nest) {
        if (nest volume > 2 * cache size)
          mark r evict-me;
        else if (volume unknown && nest level >= 2)
          mark r evict-me;
        else if (r has no temporal reuse with the next nest)
          mark r evict-me;
        if (r has spatial reuse)
          set reference step;
      }
    }
  }
}
```

Figure 3.10. Algorithm for setting evict-me tag

of each nest (near $8 * 10^6$) is greater than twice the cache size. We are able to mark very aggressively because the evict-me bit is only examined on a miss, when the architecture needs to replace something.

In our implementation, we encode the spatial locality information of a reference into the memory instruction and let the hardware detect it at run time. The encoding method is described in Section 3.2.2. We use five bits to encode the reference step. For a reference with evict-me tag marked by the compiler, if it also has spatial locality, the run-time environment waits to set the bit until after the spatial reuse is complete.

3.2.4 Effectiveness of the Evict-me Algorithm

The evict-me algorithm is sensitive to both program access patterns and cache configurations. For a specific program with a specific input, evict-me bits can be very effective in one cache configuration but help little in another. Take the simple program in Figure 3.1 as an example. Assume that N is 4K, the word size is 4 bytes, and the starting address of array A is aligned to a 16K boundary. In a 32K 2-way Level 1 cache, A(I), B(I), and C(I) will all map to the same set for each I. In this case, annotating B(I) with the evict-me bit will help reduce inter-nest misses. However, in a 64K 2-way Level 1 cache, array A and B will map

to different cache areas and thus evict-me will perform exactly the same as LRU. Given a complicated application that contains many loops and different access patterns, evict-me will yield better cache replacement for some and not other inputs and cache configurations.

3.3 Hardware Implementation

A simple implementation of evict-me in the ISA is to provide a duplicate set of memory instructions that set the evict-me tags and are otherwise the same as the original set. We believe that the widening performance gap between memory and processor speeds must eventually be reflected by additional instructions in the ISA that help compensate for this gap. Hence, adding a new set of load and store instructions to the ISA is one step in this direction, and a simple step. However, our 1-bit evict-me replacement functionality can also be implemented without changes to the ISA in some architectures. For instance, on the Alpha 21264, we can first use the "prefetch and evict-next" instruction to set the evict-me bit and then perform a register load or store [58]. This implementation needs two loads or stores to set an evict-me bit and thus suffers relative inefficiency.

We use five extra bits in each memory instruction that the compiler sets to resolve run-time spatial locality (see Section 3.2.3). An alternative hardware implementation is to use a new instruction to store the 5-bit constant into a special register. The next memory operation will access the special register to detect spatial reuse. The compiler can use loop unrolling to avoid any extra instructions.

3.4 Compiler Implementation

We implemented our compiler analyses in Scale, a compiler infrastructure developed by our research group. We gave a brief description of Scale in Section 2.5. Our analyses described in Section 3.1 and Section 3.2 are performed on Scribble, the intermediate representation in Scale. For our experiments, we apply all available scalar optimizations in

Scale. We do not apply loop transformations because the cost model in Scale to determine when to apply them is still immature.

We write Scale in Java and implement each optimization as a subclass of an abstract class, *Optimization*. We treat our cache replacement analyses as a special optimization and implement them in two classes. One is used to generate evict-me annotations and the other is for the 16-bit encoding. We put them in the last phase of high-level optimizations and transmit the annotations directly to the back-end. The critical steps before cache replacement analyses involve building a loop structure for each routine, detecting loop invariants and loop induction variables for each loop, applying dependence testing on loop structures, and constructing regular sections for each loop. Loop structures are built during control flow analysis in the early stage of Scribble construction. The remaining steps are applied immediately before we conduct cache replacement analyses. Loop invariant and loop induction variable detection are two prerequisite steps for dependence testing. Although some other optimizations, such as scalar replacement, also call dependence testing, Scale has to do the testing again because the optimizations preceding the replacement analyses do not incrementally maintain the dependence graph. The structure of the dependence graph follows the design proposed by Kennedy et al. [57].

Our cache replacement analyses traverse the loop structures of each routine, check locality using the locality graph generated by the dependence testing, and estimate data volume based on regular sections. We annotate each load expression corresponding to an array reference in a loop with a 1-bit or 16-bit reuse level, depending on the encodings. When the back-end generates C code or assembly instructions for the annotated load expression, it will output the annotations as well. For C code, we output the annotations using inline assembly. For assembly code, we use unimplemented instructions to convey the hints.

3.5 Experimental Results

We use nine benchmarks. *Liv18*, *Vpenta*, *Erlebacher*, and *Jacobi* are loop kernels. *Swim*, *Tomcatv*, and *Applu* are from SPEC95. *Arc2d* is a Perfect benchmark and *Appsp* is from the NAS Benchmarks. We selected benchmarks that had high miss rates or loop nest structures with inter-nest misses, and that run through our compiler.

To study our cache replacement algorithms and their interactions with other miss reduction techniques comprehensively, we use two simulators: SimpleScalar 2.0 [15] and URSIM [116]. Scale outputs annotated SPARC assembly for URSIM and C code for SimpleScalar. Although a later version of sim-outorder (a simulator in SimpleScalar tool set) is able to simulate the performance impact of caches, we do not show cycles because the cycle count in SimpleScalar 2.0 is accurate only when the memory system is lightly utilized [8]. Early on, when we implemented our replacement algorithms, our collaborators at the University of Utah extended RSIM by adding a more detailed memory systems. Since memory system performance is our major interest, we instead use URSIM to simulate the performance effect of our algorithms. Below we first introduce the two simulators and parameters we use. We then present simulation results.

3.5.1 Simple Scalar 2.0 and Experiments Setting

The SimpleScalar tool set is a suite of computer simulation tools that provide both detailed and high-performance simulation of modern microprocessors [15]. The suite consists of a set of simulators: sim-fast, sim-safe, sim-cache, sim-cheetah, sim-profile, and sim-outorder. Sim-fast is only a functional simulator without time accounting. It assumes no cache and executes each instruction serially. Sim-safe adds checks for memory alignment and memory access permissions. These two simulators are much faster than the others in the tool set due to their simplicity. They are particularly useful for generating traces and simulating basic functions.

Sim-cache and sim-cheetah are cache simulators that do not consider the effect of cache performance on execution time. However, both simulators report accurate miss counts of a single-level cache, ignoring the interactions in the memory hierarchy. Sim-cache simulates three replacement policies: LRU, Random, and FIFO. Sim-cheetah implements LRU and MIN, the optimal cache replacement algorithm originally proposed by Belady [10]. In our experiments, we use both simulators to report miss rates. We also implement an 8-entry fully associative victim cache [54]. The victim cache is positioned between the first and second-level cache. On a Level 1 cache miss, the architecture checks the victim cache. When there is a hit to the victim cache, the hit entry is exchanged with the replaced victim in the Level 1 cache, otherwise the replaced victim is put into the victim cache. The victim cache is implemented in sim-cache, but we use the optimal miss rate from sim-cheetah.

Sim-profile is a functional simulator that can generate detailed profiles of instruction classes and addresses, text symbols, memory accesses, branches, and data segment symbols.

Sim-outorder is the most complicated and detailed simulator. It supports out-of-order issue and execution. The register update unit (RUU) scheme uses a reorder buffer to rename registers automatically and hold the results of pending instructions. The reorder buffer retires completed instructions in program order. The processor also contains a load/store queue to support speculative execution. Store values are placed in the queue if the store is speculative.

PISA (portable instruction set architecture), the instruction set architecture in the SimpleScalar Toolset 2.0, is derived from the MIPS_IV ISA. Later versions of SimpleScalar support other ISAs, for example, the Alpha ISA as we use in Chapter 4. PISA provides a 16-bit *annote* field in memory instructions. We use the field to encode reuse levels or evict-me tags. Typically, a field annotation is in the form of *lw/6:4(7) $r6, 4($r7)*, which sets bit 4 to bit 6 in the annotation field to 7.

The reuse levels or evict-me tags are annotated in Scribble through the annotation tool. The back-end translates Scribble to C with annotations of reuse levels or evict-me tags. The annotations are implemented as special inline assembly instructions. The C code with assembly inline is then compiled using the modified version of gcc in the tool set. Gcc in the tool set generates PISA binaries. We updated sim-cache and sim-cheetah to interpret the annotations.

3.5.2 URSIM and Experiments Setting

We use URSIM developed at the University of Utah to simulate the performance impact of the evict-me cache [116]. URSIM is an extension to RSIM, an execution-driven simulator for instruction level parallelism (ILP) based shared-memory multiprocessors and uniprocessors [83]. It simulates a state-of-the-art out-of-order processor, lock-up free cache, and multi-bank memory. Although URSIM models a uniprocessor or shared-memory multiprocessor, we use the uniprocessor configurations only. The key features of the processor model include superscalar execution, out-of-order scheduling, register renaming, dynamic branch prediction, non-blocking loads and stores, speculative load execution, superpage-supporting TLB, precise exceptions, and register windows. The key memory hierarchy features include two levels of cache, multiported and pipelined L1 cache, pipelined L2 cache, multiple outstanding cache requests, and memory interleaving. URSIM extends RSIM by adding a more complicated memory system. URSIM supports Synchronous DRAM (SDRAM) and Rambus DRAM where hot pages and channel contention are modeled.

The URSIM processor model is close to the MIPS R10000 CPU implementation [115]. A major difference between URSIM and the R10000 is that URSIM executes the SPARC instruction set, which uses a register window mechanism that the R10000 does not implement. URSIM models the R10000 *active list*, *register map table*, and *shadow mappers*. The active list holds the currently active instructions, corresponding to the *reorder buffer*

or *instruction window* of other processors. The register map table maps logical registers to physical registers. Shadow mappers store the register map table information to allow single-cycle state recovery on branch misprediction. The URSIM instruction pipeline contains five stages: fetch, decode, issue, execute, and complete. The fetch and decode stages process instructions in program order, but the issue, execute, and complete stages may process the instructions out-of-order. Instructions graduate in-order after passing through all five stages, which enables URSIM to implement precise exceptions.

URSIM allows many of the processor and memory features to be configurable at simulation time. We configure URSIM to fetch and graduate a maximum of four instructions per cycle. Our processor configuration contains two ALUs, two FPUs, and two address generation functional units. URSIM uses a two-bit history branch predictor that contains up to 512 counters. URSIM uses eight shadow mappers, which restricts the number of outstanding branches to eight. The L1 cache is write-through with a non-allocate policy. The L1 cache has two ports, which means that two accesses can occur concurrently. The L2 cache is write-back with a write-allocate policy. The L2 cache maintains inclusion of the L1 cache.

Scale generates SPARC assembly code with annotated load/store instructions for URSIM. The evict-me tag and reference step of a memory instruction are encoded into an unimplemented SPARC instruction. We put the marking instruction before the memory instruction. We updated the URSIM preprocessor to merge the unimplemented instruction into the memory instruction following it. We thus avoid additional instruction overhead at run time. We updated URSIM to accept the special load/store instructions and perform the corresponding replacements.

3.5.3 Experimental Results Using SimpleScalar 2.0

In Figures 3.11 through 3.18, we show the miss rates of LRU, LRU with victim cache, evict-me, evict-me with victim cache, and the 16-bit Prediction algorithm, on 2-way and

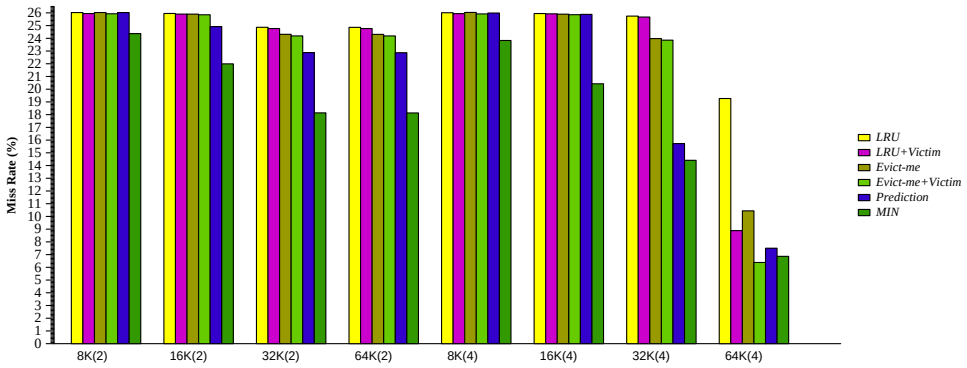

Figure 3.11. Vpenta

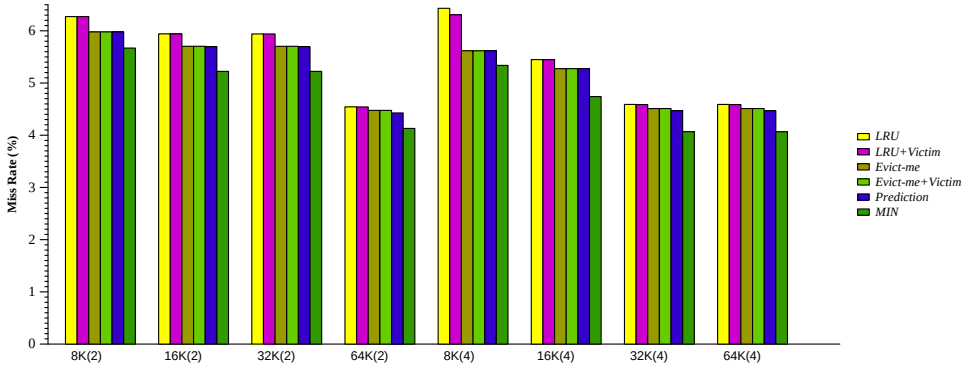

Figure 3.12. Liv18

4-way set associative caches. We also give the miss rate of the MIN algorithm, which is the optimal miss rate without considering write-backs [10]. The cache sizes range from 8K to 64K by powers of two and the cache line size is 32 bytes. We observe that evict-me caching provides up to 45% improvement in the number of misses in *Vpenta* for a 4-way 64K cache. This result is significant considering the minor architectural support we need. We improve the miss rates of *Swim*, *Tomcatv*, *Liv18*, and *Jacobi* by 10-20% in the best cases. *Appsp*, *Arc2d*, and *Erlebacher* improve by 5-6% in the best cases. Evict-me caching never degrades the miss rate, although the mispredictions we mentioned in Section 3.2 might cause a degradation.

We note that the 16-bit Prediction algorithm can further improve miss rate in some cases. For example, for *Vpenta* at 64K and *Tomcatv* at 16K, it achieves more than a 20% improvement even when compared to a 4-way evict-me cache. This result means that there

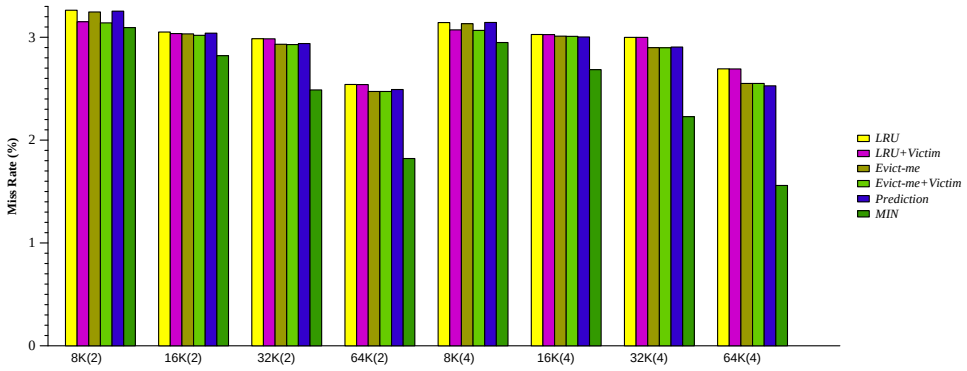

Figure 3.13. Appsp

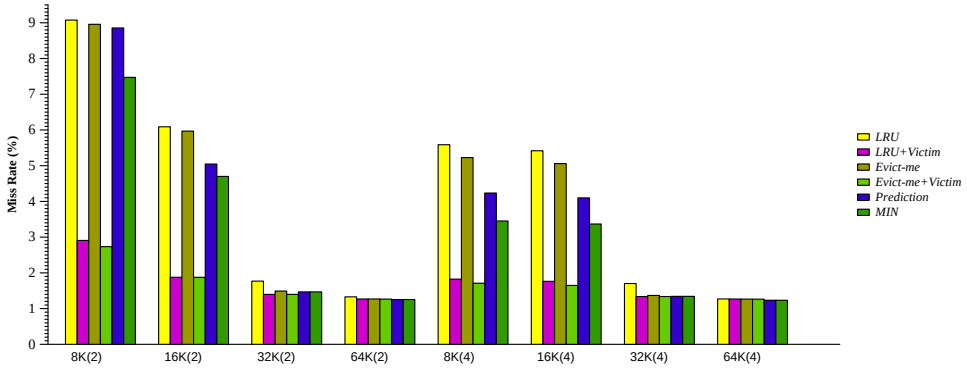

Figure 3.14. Tomcatv

is still some room for the compiler to improve, although we observe that the one evict-me bit is sufficient in most cases. In some cases, such as in *Jacobi* and some configurations for *Vpenta*, the miss rates of evict-me and Prediction are pretty close to optimal (MIN). There is still a large gap in the other cases. We think a better encoding for the 16-bit algorithm can come very close to optimal.

Both the evict-me and the 16-bit Prediction algorithm present significant improvements in certain cache configurations but very minor ones in others. Generally, The evict-me and the Prediction algorithm reduce conflict misses. When the cache size is very big, there are few conflicts available for them to resolve. When the cache size is very small, the conflicts become so intense that no replacement algorithm can do well. The improvement is sensitive to the degree of associativity and cache line size, because those factors affect the distribution of conflict misses. Increasing the degree of associativity can reduce conflict

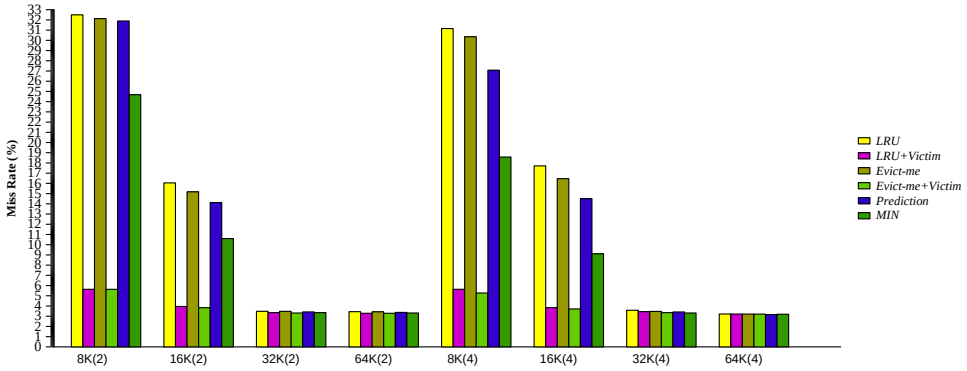

Figure 3.15. Swim

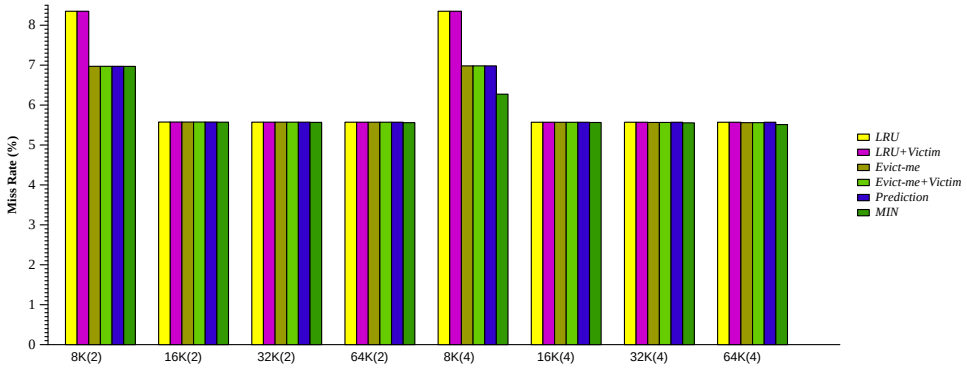

Figure 3.16. Jacobi

misses and also give the evict-me algorithm and the Prediction algorithm more flexibility. We expect the evict-me and Prediction algorithms to increase their relative performance, as compared to LRU, in proportion to the increase in the degree of associativity when associativity is small.

Victim caches eliminate many misses in *Tomcatv* and *Swim* when the cache size is small. However, in *Jacobi*, the victim cache has no effect at all, but evict-me caching shows a 16% improvement at 8K. A further observation is that the two strategies can work together. For example, in *Vpenta*, the miss rate is reduced from 8.9%, when applying victim cache only, to 6.4%, when applying both. The victim cache by itself outperforms evict-me on some programs but putting the two strategies together always dominates using only one of them.

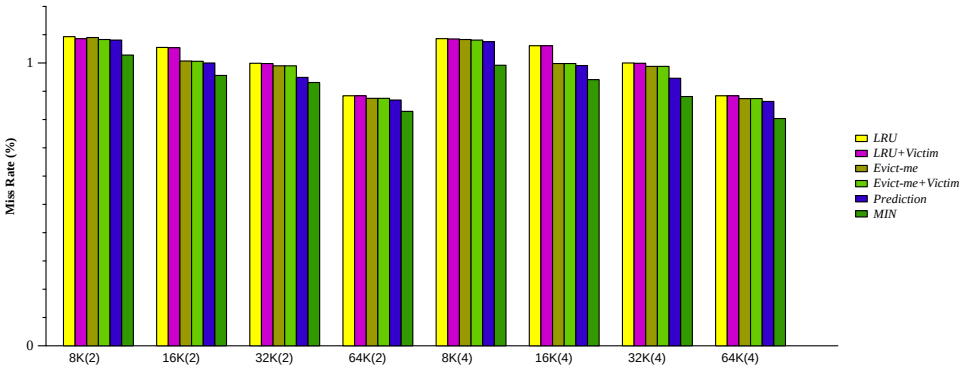

Figure 3.17. Erlebacher

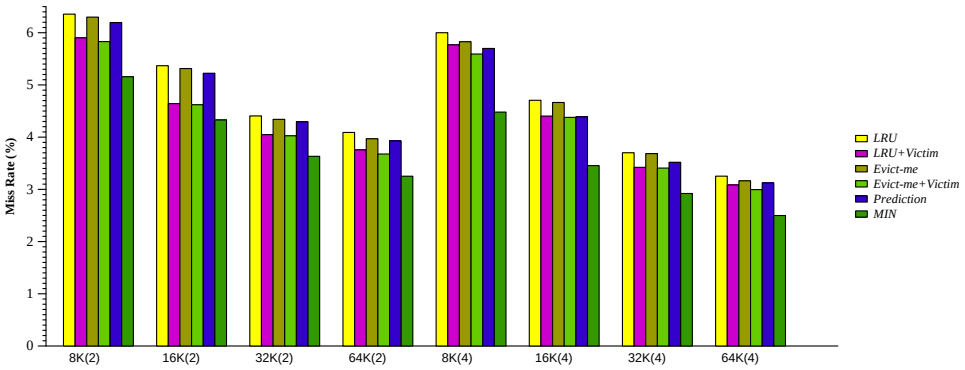

Figure 3.18. Arc2d

We further investigate the performance of the victim cache by observing the status of the victim cache when there is a hit to it. We keep track the number of lines in the victim cache that are from the same cache set in the Level 1 cache as the hit line. In particular, we are interested in those lines that are older than the hit line (evicted from the Level 1 cache earlier than the hit line). Let the *associativity extension* be the total number of such cache lines divided by the total number of accesses to the victim cache. Table 3.3 lists the statistics for *Tomcatv* and *Swim*, where the victim cache performs extremely well at 8K and 16K. Usually a larger *associativity extension* means a lower miss rate. In a coarse estimation,

	2-way				4-way			
Program	8K	16K	32K	64K	8K	16K	32K	64K
TOMCATV	1.98	1.30	0.22	0.05	2.14	2.18	0.24	0.00
SWIM	1.81	1.66	0.04	0.05	2.81	2.65	0.14	0.00

Table 3.3. Associativity extension by victim cache

	Conf. 1	Conf. 2	Conf. 3
Level 1	8K, 2-way	32K, 2-way	64K, 4-way
	32 byte cache line		
Level 2	128K, 2-way	256K, 4-way	512K, 2-way
	128 byte cache line		

Table 3.4. Three cache configurations

assuming the *associativity extension* is *e*, the miss rate of a Level 1 cache of size *s* with associativity *a* plus a victim cache is similar to that of a Level 1 cache whose associativity is $a+e$ and size is $(a+e)/a*s$. For *Tomcatv* and *Swim*, the associativity extension is much bigger at cache sizes of 8K and 16K than at 32K and 64K. This difference reflects the huge improvement in miss rate at 8K and 16K for the two benchmarks.

3.5.4 Experimental Results Using URSIM

In practice, evict-me can be turned on/off in both levels of the cache. We apply three Level 1 and Level 2 cache combinations of sizes and associativities, as shown in Table 3.4. The three configurations share the same cache line size and latencies. The Level 1 cache line size is 32 bytes and the latency 2 cycles. The Level 2 cache line size is 128 bytes and latency 8 cycles. The latency for memory access is between 48 and 200 cycles, depending on the state of the machine; this range reflects the sophistication of the accurate memory model. We also examine all of our benchmarks using a 5 year hardware projection where the Level 2 latency is increased to 20 cycles and the memory access latency is 200-500 cycles. These projections come from Agarwal et al. [3].

3.5.4.1 Miss Rates Results

Figures 3.19 through Figure 3.21 show the normalized miss rates of the three cache configurations when evict-me replacement is turned on for both Level 1 and Level 2 caches. We list the LRU miss rate at the top of each bar. For example, in Configuration 1, the miss rate of *Applu* for LRU at Level 1 is 9.43%. Evict-me reduces the miss rate by 21%. We state the miss rate of the Level 2 cache as the Level 2 misses divided by total accesses rather

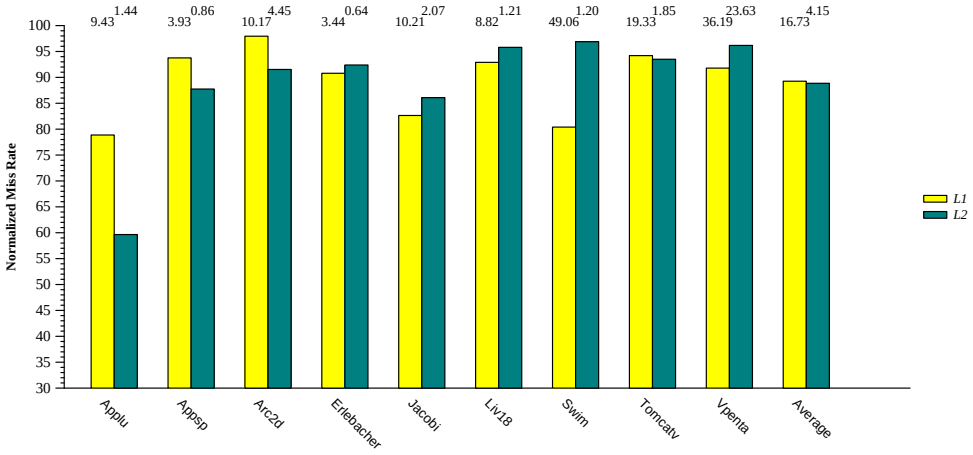

Figure 3.19. Miss reduction by evict-me (Conf. 1)

that by the misses of the Level 1 cache. We use this representation to show the combined effects of evict-me on the two levels of cache. The miss reduction in the Level 2 cache comes not only from better replacements in the Level 2 cache itself, but also from better Level 1 replacements, which reduce the traffic between the two caches.

We observe a significant miss reduction for both levels of cache. As we discussed in Section 3.2.4, evict-me can be very effective in one cache configuration but less so in others. For *Applu*, the miss rate at Level 1 is reduced by 21.13% in Configuration 1 but only 1.37% in Configuration 2. For certain cache configurations, we reduce the miss rate of *Applu*, *Swim*, and *Tomcatv* by about 50%. Overall, the miss reduction ranges on average from 10% to 20%.

3.5.4.2 Static and Dynamic Replacement Counts

Table 3.5 shows static and dynamic statistics on evict-me tags and their effect on replacements for our programs. The second column is the percent of annotated instructions among all static load and store instructions. We mark 25% of the memory instructions on average at compile time. The numbers in the remaining columns are collected under the cache configurations of a 64K L1 4-way cache and a 512K L2 2-way cache (Conf. 3), with evict-me caching on in both caches. The third and the fifth columns are the percent of

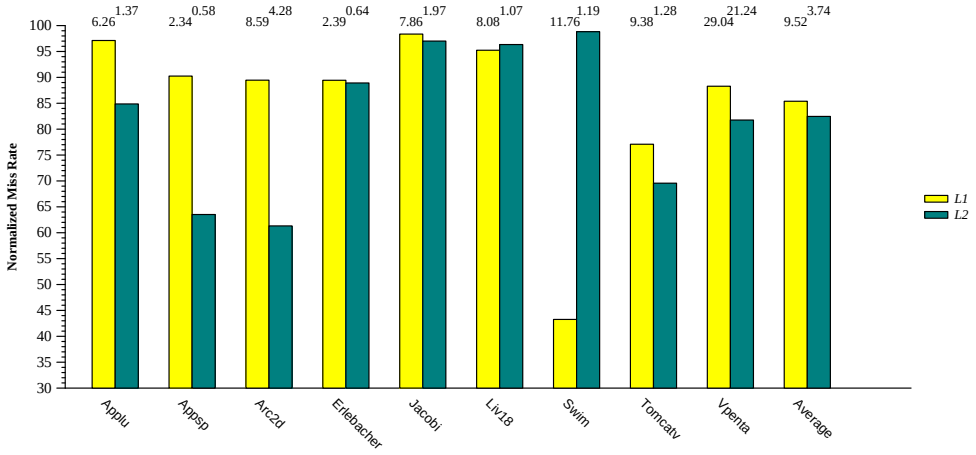

Figure 3.20. Miss reduction by evict-me (Conf. 2)

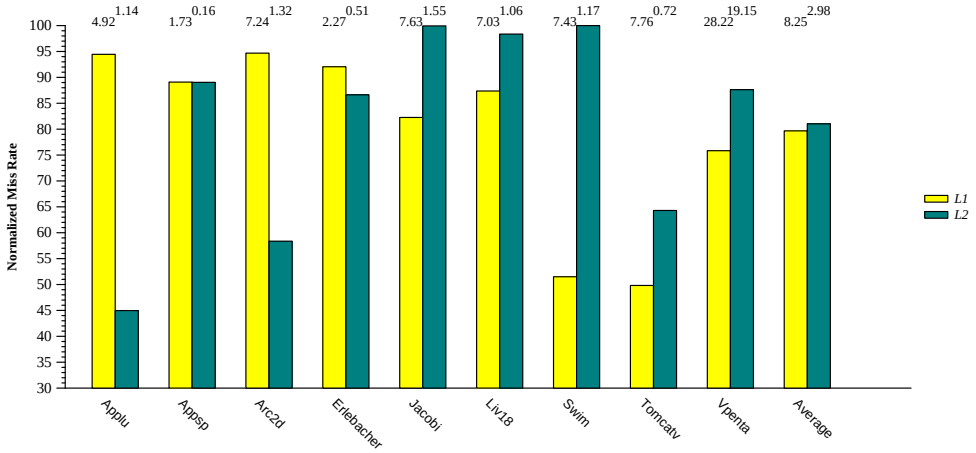

Figure 3.21. Miss reduction by evict-me (Conf. 3)

cache accesses in which we set the evict-me bit in the Level 1 and 2 caches respectively. The fourth and the sixth columns show the percent of replacements where the evict-me bit changes the replacement decision as compared to LRU's decision. It changes 4% to 24% of the decisions in the L1, and 0 to 15% in the L2. These changes do not correspond well to changes in miss rates because one change can result in several more hits, or no additional hits. For example, evict-me removes many misses for *Tomcatv* and *Swim* but alters 13% or fewer L1 replacement decisions. The changes to Level 2 replacements are on average very low which means that the L2 miss reductions also come from better L1 replacements and more L1 hits that yield less traffic between the caches.

	Static	Dynamic (Conf. 3)			
	evict-me	L1 evict-me	L1 Repl.	L2 evict-me	L2 Repl.
Applu	12.05	13.81	6.66	30.41	3.44
Appsp	8.93	9.06	15.04	24.46	9.61
Arc2d	25.96	20.61	4.29	36.30	14.86
Erlebacher	25.36	12.64	12.62	15.68	13.16
Jacobi	50.00	33.95	24.10	56.36	2.35
Liv18	43.82	30.01	21.61	53.31	1.42
Swim	20.95	21.51	8.30	53.35	0.21
Tomcatv	9.52	4.89	13.03	4.98	0.57
Vpenta	35.05	15.69	11.71	25.57	3.51
Average	25.74	18.02	13.04	33.38	5.46

Table 3.5. Static and dynamic statistics on evict-me

	8K L1, 128K L2 (Conf. 1)				32K L1, 128K L2 (Conf. 2)				64K L1, 512K L2 (Conf. 3)			
	Current			Pred.	Current			Pred.	Current			Pred.
Program	L1	L2	**L1+L2**	**L1+L2**	L1	L2	**L1+L2**	**L1+L2**	L1	L2	**L1+L2**	**L1+L2**
Applu	10.58	0.37	**11.30**	**31.91**	0.24	3.92	**4.17**	**34.21**	10.08	10.04	**11.74**	**25.86**
Appsp	0.32	2.57	**2.60**	**8.50**	0.27	5.22	**4.93**	**16.65**	0.19	0.55	**0.49**	**2.53**
Arc2d	0.00	4.89	**4.81**	**7.93**	0.00	21.97	**21.59**	**30.22**	0.00	9.87	**9.48**	**26.03**
Erle.	0.48	1.10	**1.20**	**3.27**	0.66	1.10	**1.46**	**4.67**	0.37	1.37	**1.48**	**5.92**
Jacobi	2.05	4.85	**5.17**	**11.88**	0.00	5.19	**0.79**	**1.89**	1.57	0.00	**1.40**	**2.18**
Liv18	0.32	1.17	**1.50**	**2.26**	1.35	0.64	**2.17**	**2.67**	2.12	0.48	**2.54**	**2.78**
Swim	10.57	1.48	**9.48**	**11.30**	11.46	1.72	**11.23**	**11.68**	6.59	0.00	**6.53**	**6.38**
Tomcatv	0.66	1.98	**2.45**	**7.01**	5.72	3.23	**7.30**	**13.60**	7.66	0.00	**7.62**	**16.05**
Vpenta	0.00	11.03	**5.51**	**5.83**	0.31	32.00	**21.91**	**24.98**	0.21	20.03	**21.21**	**23.07**
Average	2.78	3.27	**4.89**	**9.99**	2.22	8.33	**8.39**	**15.62**	3.20	4.70	**6.94**	**12.31**

Table 3.6. Percent performance improvement by evict-me

3.5.4.3 Simulated Performance Results

Table 3.6 shows the performance impact of evict-me. The columns titled "L1" and "L2" show performance improvements when the evict-me caching is turned on for the Level 1 cache only and for the Level 2 cache only, respectively. The columns titled "L1+L2" are the improvements when the evict-me caching is turned on for both caches. For current technology, we see reductions in execution time of 4.89%, 8.39%, and 6.94% on average for the three configurations. We see larger improvements in simulated cycle time, 9.99%, 15.62%, and 12.31%, for technology predicted for 5 years from now, when the gap between processor speed and memory speed will be larger. Usually and on average, the performance improves most when evict-me caching is turned on in both caches. We see more contribution from the Level 2 cache in most cases, because the gap between the access time of the

Level 2 cache and memory is relatively larger than the gap between the two caches. In our experience, out-of-order execution often hides L1 cache latencies, but not L2 [70].

An interesting case is *Vpenta*, which improves the most with evict-me turned on only in the Level 2 cache for configurations 1 and 2. When evict-me is on in both caches, the Level 1 evict-me cache replacements change the access pattern of the Level 2 cache, and in this case, reduce its effectiveness. In *Swim*, the opposite is true; the Level 1 cache dominates the evict-me performance improvements because the Level 1 miss rate is very high and evict-me reduces it by 19%-56%.

3.5.4.4 A Less Aggressive Compiler Marking Algorithm

We also investigate a slightly more conservative compiler algorithm for setting the evict-me bit. We use the same algorithm as in Figure 3.10, except we change the test for the nest level ≥ 2 to be ≥ 3, and thus mark fewer references as evict-me. With this algorithm, our results are unchanged for *Jacobi*, *Liv18*, and *Vpenta* because the compiler computes the data volume precisely. For *Applu*, *Appsp*, and *Arc2d*, this more conservative algorithm is slightly better, but for *Tomcatv* and *Swim*, it does not set enough bits, and the original is much better.

3.6 Chapter Summary

This chapter develops a theoretical model for static compile-time analysis to direct cache replacement algorithms and prove that it is at least as good as LRU. This work opens a new path for reducing cache misses by using compiler hints to improve replacement decisions. We present and implement a 16-bit and a 1-bit (evict-me) version of our algorithm. We demonstrate that the 1-bit evict-me algorithm is practical enough to implement in current set-associative caches, and in multiple levels of the cache hierarchy. Furthermore, our simulation results show that the evict-me algorithm consistently improves performance

through reduced miss rates when compared with LRU and is effective on multiple levels of the cache.

We conclude that both the compiler hints and the run-time history used by the LRU policy play critical parts in our unique evict-me cache replacement algorithm. We have shown that our compiler hints are predictive and therefore can help the hardware to make better run-time decisions. However, the compiler hints use heuristics and thus are neither complete nor perfect. It is not guaranteed that each cache set always contains an evict-me line when a conflict occurs. In this case, the LRU policy will take over.

A better cache replacement policy can help reduce unnecessary conflict misses. The remaining misses still hurt cache performance. The next chapter will describe a cooperative prefetching technique that effectively hides much of the memory latency of misses that cannot be avoided.

CHAPTER 4

COMPILER-GUIDED REGION PREFETCHING

This chapter introduces a new prefetching technique. This hardware/software cooperative technique uses compiler hints to enhance an aggressive hardware prefetching engine. The prefetching requests are initiated by the demand misses as well as the compiler hints.

Cooperative cache replacement reduces cache misses but does not eliminate them. To improve cache performance further, we can rely on latency tolerance techniques such as data prefetching that hides latency by prefetching data that will be used in the near future so the future access can be a hit. However, the prefetch itself still suffers miss latencies. A large number of software and hardware prefetching schemes have been investigated. Most of them rely on either pure software direction or pure hardware mechanisms but rarely on both. In this chapter we describe a new cooperative prefetching technique: guided region prefetching (GRP). GRP builds on the strengths of both hardware and software prefetching. In GRP, a sophisticated compiler analysis produces a rich set of load hints, including the presence or absence of spatial locality, pointer structures, or indirect array accesses. A run-time hardware engine, triggered by L2 cache misses, generates prefetches based on the compiler's hints. GRP thus benefits from compiler analysis of application reference patterns, but—unlike traditional software prefetching—the compiler is not required to generate or schedule individual prefetch addresses. Because the hardware generates the prefetches, it can run far ahead of the missing references. Because the compiler guides it, the hardware need not struggle to deduce future references with complex pattern matching on prior accesses stored in large tables.

Using previously proposed techniques [69], the GRP hardware prefetching engine keeps uniprocessor bus contention low by prefetching only when the memory bus is otherwise idle, and keeps cache pollution low by loading prefetches into the LRU set of the L2 cache. Without compiler support, this prefetching hardware is effective at improving performance, but consumes copious bandwidth. Through GRP, the compiler informs the hardware of application reference patterns, enabling the hardware to prefetch only when it is likely to be effective. We evaluate compiler hints that mark loads with the following hints: *spatial*–prefetch the spatial region around a load; *size*–how many lines to fetch on a spatial reference; *pointer*–prefetch by following the pointers in the load's cache line; *recursive*–prefetch this pointer data structure recursively. For *size* hints, the compiler can encode a *variable-size region* that specifies how much to prefetch based on enclosing loop bounds, instead of using a fixed value. The compiler also generates indirect prefetching instructions which trigger prefetching a set of references using an indirection array.

We also propose a pointer prefetching technique that aggressively prefetches blocks pointed to by a pointer-like word in a fetched cache line. This work was also implemented concurrently by Cooksey et al. [32] who refer to this scheme as *content-aware prefetching*. We use this notion for our further discussion. We find region prefetching generally performs better than pointer prefetching and putting them together degrades performance due to excessive memory traffic.

This cooperative GRP hardware/software interface improves the high performance of the previously proposed scheduled region prefetching (SRP) [69] by over 10% on two of the SPEC2000 benchmarks, and matches the performance of SRP on the rest. Table 4.1 shows a summary of the GRP results using the geometric mean. We show GRP both with (GRP/Var) and without (GRP/Fix) variable-size region prefetching. Without prefetching, the mean performance across the benchmark suite is 33.7% lower than a perfect Level 2 cache. Stride prefetching (using the Sherwood et al. design [92]) provides a 15% speedup over a system with no prefetching. SRP, which uses no compiler analysis, outperforms

	Speedup	traffic increase	Performance gap from perfect L2
No prefetching	1	1	33.72
Stride prefetching	1.147	1.09	23.99
SRP	1.226	2.80	18.75
GRP/Fix	1.216	1.62	19.42
GRP/Var	1.212	1.23	19.69

Table 4.1. Summary of prefetching performance and traffic

stride prefetching by 7%, but consumes excessive memory bandwidth, a 180% increase over a system with no prefetching. GRP provides near-equivalent performance to SRP but with substantially less traffic, an increase of only 23% over not prefetching. This reduction in traffic saves power and is more amenable to multiprocessor systems, where additional traffic more often directly affects performance. Both SRP and GRP still incur a 19% gap versus a perfect L2.

In Section 4.1, we describe the hardware implementation of the region prefetcher and content-aware pointer prefetcher. In Section 4.2, we discuss our hint encoding methodology. In Section 4.3, we present a set of compiler analyses that generate hints to direct the two prefetchers. We then discuss our compiler implementation in Section 4.4 and experimental results in Section 4.5

4.1 Hardware Prefetching Engine

The GRP hardware prefetching engine builds on the scheduled region prefetching designed by Lin et al. [69]. We extend the original design with two capabilities. First, we add support for aggressive prefetching of pointer-based data structures. Second, we add the ability to prefetch indirect array references under software control.

4.1.1 Scheduled Region Prefetching

Scheduled region prefetching (SRP) aggressively exploits spatial locality by attempting to prefetch large (4 KB) memory regions on each L2 cache miss [69]. The two negative effects of aggressive prefetching—memory bus contention and cache pollution—are ad-

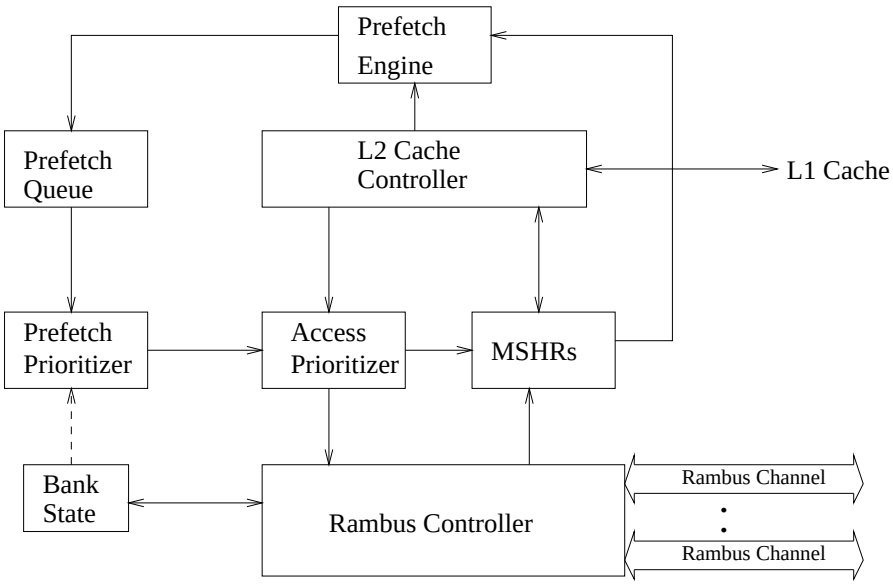

Figure 4.1. Prefetch engine organization

dressed directly by reducing the priority of prefetches in memory bus request scheduling and in replacement decisions, respectively. Unlike most prefetching schemes, which must maintain high prefetch accuracy to avoid degrading performance, SRP can identify and access prefetch candidates liberally without degrading uniprocessor performance.

Figure 4.1 shows the memory system with the SRP engine that forms our experimental baseline. The access prioritizer is the central component of the SRP prefetching engine. It forwards requests to the memory controller whenever the controller indicates that a memory channel is idle. The prioritizer forwards prefetch requests only when there are no outstanding demand misses from the L2 cache. Demand misses thus encounter contention only from prefetches that the memory controller has already issued, and not from prefetch candidates buffered in the prefetch queue. The miss status handling registers (MSHRs) track all outstanding accesses, regardless of type.

On an L2 cache miss, the prefetching engine allocates a new entry in the prefetch queue representing the aligned memory region containing the accessed block. Each prefetch queue entry contains the base address of the region, a bit vector indicating the prefetch candidate blocks in the region, and an index field, which identifies the next block within

the region to prefetch. On the first miss to a region, the engine initializes the bit vector to identify the blocks not already present in the L2 cache, and sets the index field to indicate the next prefetch candidate block after the miss block. It adds these new entries to the head of the queue, giving them priority over older, and thus typically less relevant, entries. The queue is a fixed size (32 in these experiments), and old entries fall off the bottom. On a miss to a region already in the queue, it clears the bit corresponding to the miss block, sets the index field to the next prefetch candidate block after the new miss block, and moves the prefetch bit vector entry to the head of the queue. In this work, we use a base region size of 4 KB and a cache block size of 64 bytes, resulting in a 64-bit vector and a 6-bit index field. Once the controller prefetches all the candidates, it deallocates the entry.

Although the access prioritizer practically eliminates performance loss from useless prefetches due to bandwidth contention, prefetching can still pollute the cache by over-prefetching. We address this issue by placing prefetched data in the lowest priority position of the replacement scheme. The controller puts prefetched data in the LRU position of the pertinent cache set, and moves a block to the MRU position only if it is referenced explicitly by the CPU. As a result, useless prefetches in an *n*-way associative cache can displace at most one *n*th of the useful data in the cache. (We use a 4-way set associative cache in our experiments.) The drawback is that the controller occasionally replaces potentially useful prefetched data before they are referenced; however, previous work [69] shows this effect to be insignificant. As a final optimization, the queue issues prefetches first to those DRAM banks that already have the needed page open.

Scheduled region prefetching is highly effective at exploiting spatial locality to improve performance [69]. However, it has two shortcomings addressed by GRP. First, SRP does not provide any direct support for non-spatial reference patterns. We add a pure hardware pointer prefetching mechanism to address this issue (see Section 4.1.2). We also add an indirect array scheme that requires compiler support (see Section 4.1.3). However, for the SPEC benchmarks, we find that spatial prefetching works as well as pointer schemes—

even for pointer-intensive benchmarks—because of the regular layout programmers use and memory allocation patterns for pointer data structures. Second, SRP can produce copious amounts of excess memory traffic. Although this useless traffic does not reduce uniprocessor performance due to SRP's prioritization techniques, it consumes energy, can cause contention with useful prefetches, and may reduce performance in a multiprocessor environment. We thus use compiler hints for spatial and pointer accesses to gain both lower bandwidth and higher accuracy. We describe the GRP hardware modifications and hints below in Section 4.1.3, and the compiler analysis itself in Section 4.3.

4.1.2 Hardware Prefetching of Pointer-Based Structures

As discussed in Section 2.3, hardware prefetching for pointer-based structures is challenging. Instead of using complex hardware to recognize pointer traversal patterns or store pointer correlations, the base pointer prefetching scheme greedily generates a prefetch for any fetched value that falls within the ranges of legitimate heap memory addresses. The implementation performs a simple base-and-bounds check using the start and end addresses of the heap. In the Alpha ISA, pointers are aligned 8-byte entities; thus the engine must check only eight values out of each 64-byte cache block.[1]

Once the controller identifies a datum as a possible pointer value, it translates the virtual address to a physical address and forwards the address to the SRP prefetch queue, which allocates a region-style entry for the prefetch. We generalize this mechanism to chase recursive pointers by scanning prefetched lines for addresses and generating additional prefetches.

Because these pointer dereferences frequently do not exhibit spatial locality, the prefetching engine sets only two bits in the entry's prefetch bit vector, indicating the block containing the prefetch address and its immediate successor (which prefetches data struc-

[1]Cooksey et al. [32] describe a similar but more efficient pointer test using bit masks, and apply it to prefetching in the more challenging IA32 environment.

Benchmark	# of Cache Blocks		
	1	2	> 2
164.gzip	100.0%		
175.vpr	99.8%	0.2%	
177.mesa	80.1%	9.7%	10.2%
179.art	100.0%		
181.mcf	25.3%	74.7%	
183.equake	100.0%		
186.crafty	100.0%		
188.ammp	41.4%		58.6%
197.parser	98.9%	1.1%	
254.gap	100.0%		
256.bzip2	100.0%		
300.twolf	100.0%		
sphinx	94.0%	5.6%	0.4%

Table 4.2. Size distribution of pointed-to structures

tures that span two cache blocks). The statistics shown in Table 4.2 list the static size distribution of pointed-to structures in 13 C benchmarks we are using. The numbers are collected for the compiler marked pointer/recursive loads as we will discuss in Section 4.1.3. The sizes are measured by the number of cache blocks of size 64. Except *ammp*, the pointed-to structures in all other benchmarks are dominantly one or two blocks in size. This suggests that it is sufficient to prefetch only two blocks for pointer prefetching if we do not consider implicit spatial locality.

4.1.3 GRP: Incorporating Compiler Prefetch Hints

This section describes the compiler hints used by GRP to improve the precision of L2 spatial and pointer prefetching. The GRP compiler annotates load instructions with hints predicting whether spatial or pointer-based prefetches will be useful. In this study, the compiler conveys the hints with a duplicate set of memory instructions from unused Alpha opcodes as discusses in Section 4.2. The memory system propagates the load's hint bits through the memory hierarchy with any resulting request. Table 4.3 presents the five hints and shows typical representative code snippets for each. We summarize the changes to the hardware for each hint below, and then describe the pointers, recursive pointers, and the indirection hardware in more detail.

code in loop	spatial	indirect	pointer	recursive pointer	size
a[i]	√				√
a[b[i]]	√	√			
∗p; p+=c	√				
p→f			√		
p = p→next				√	

Table 4.3. Compiler hints for representative references in loops

- A *spatial* hint indicates that a reference is likely to exhibit spatial locality. GRP initiates a spatial prefetch only when the L2 miss is marked spatial.

- A *size* hint combined with a loop upper bound indicates how many cache lines to prefetch.

- An *indirect* hint indicates that the program is using an array to index a second array. On an indirect L2 miss, GRP generates sets of prefetches based on the base address and the index values.

- A *pointer* hint indicates that the reference is to a structure that contains one or more other pointers that the program is likely to follow. If the reference is an L2 miss, GRP scans the returned block for pointer values and generates prefetches only for those values.

- A *recursive pointer* hint indicates not only that the reference is to a structure that contains other pointers, but that the program recursively follows these pointers. On a recursive pointer L2 miss, GRP scans the returned data for pointer values, generates prefetches for these addresses, and continues generating prefetches on the subsequent n levels into the recursive data structure. (We use $n = 6$ in our experiments.)

4.1.3.1 GRP for Spatial Region Prefetching

SRP prefetches a region on any demand L2 miss. GRP filters useless prefetches by indicating which load/store is a candidate for region prefetching. It initiates a region pre-

fetching request only when the L2 miss is marked *spatial*. Contrary to a spatial hint is a *non-spatial* hint, which causes the prefetching engine not to prefetch on a miss. The negative non-spatial hint is helpful in places that the compiler analysis cannot reach, such as library calls. We find that using a positive spatial hint is sufficient for the selected SPEC benchmarks.

4.1.3.2 GRP for Variable-Size Region Prefetching

GRP by default prefetches the same fixed region size as SRP. If the spatial reuse of a reference does not span the default region size, prefetching wastes bandwidth. We enhanced GRP to allow the compiler to control region sizes for references in singly nested loops. The compiler computes the loop upper bound for the primary induction variable and conveys the bound to the hardware using a special instruction. The compiler encodes a coefficient for each spatial reference in the loop. On a miss, the prefetch engine uses this bound and the coefficient to calculate the region size as *loop bound* $\ll$ *coefficient value*. The region size is rounded up to the nearest power of 2 and used to set the bit vector of the prefetching request.

4.1.3.3 GRP for Indirect Array References

Two of the benchmarks from the SPEC2000 suite (*vpr* and *bzip2*) incur a significant number of misses due to indirect array references of the form *a[b[i]]*. References to *a* are not amenable to spatial prefetching unless the *b[i]* values are clustered, which cannot be determined statically. Pointer prefetching for these references is ineffective since the desired addresses are computed, not contained in memory as pointers. A specialized extension to GRP targets these patterns. A single *indirect prefetch* instruction conveys both a base address (*&a[0]*), an element size (*sizeof(a[0])*), and an index array address (*&b[i]*) to the prefetching engine. The prefetch engine reads the cache block containing *b[i]* and, for each word in the block, generates a prefetch address by adding the scaled value to *&a[0]*. GRP then forwards these addresses to the prefetch queue, as in the pointer prefetching scheme.

Currently, we assume the index array element size (*sizeof(b[0])*) is 4, which is typical on most systems, although the element size could be included in the instruction if necessary.

The indirect prefetching instruction can take the general form of a memory instruction as *indirectpref $a, d($b)*. We implement it with the store instruction, *stl_c*, which is used only in parallel applications. Note that our compiler generates assembly code. We rely on the native compiler and linker to generate the executable. It is necessary to treat an indirect instruction as a store, since otherwise the peephole optimization in the native assembler may remove a useful memory instruction because the indirect instruction could introduce false redundancy.

The compiler can process a more general form of indirect reference such as *a[c*b[i]+f]* where *c* and *f* are constants known at compile time. We discuss our compiler algorithm in Section 4.3.3. The only extra work is computing the displacement in the indirect prefetching instruction, which will be *c*sizeof(a[0])*. We can ignore the impact of *f* if it is small, i.e., *a[c*b[i]]* will most likely sit in the same cache block as *a[c*b[i]+f]*. When *f* is large, the compiler can compute the base address as *&a[0] + f * sizeof(a[0])*. This calculation is loop invariant and thus can be moved out of loop. We do not implement this option in our compiler.

The indirect prefetching scheme is distinct from the other mechanisms proposed in this chapter because the information is encoded as a separate instruction, not as a hint on an existing load. Although the introduction of an explicit prefetch instruction adds overhead, the number of such instructions is small, and each one generates up to 16 prefetches (one for each index within a cache block of the indirection array). An alternate implementation could use a single instruction prior to a loop nest to set the base address, and an additional hint bit on the *b[i]* loads to trigger the indirect prefetches. This approach would reduce execution overhead at the cost of limiting an application to prefetching one single indirection array concurrently per base address/indirect hint pair.

4.1.3.4 GRP for Pointer and Recursive Pointer References

GRP uses the same mechanism as SRP for pointer and recursive pointer hints. However, GRP applies the mechanism only to a pointer hint miss, and GRP applies it repeatedly to the resulting prefetched lines for recursive pointer hints.

We implement GRP for pointer and recursive pointer hints by adding a three-bit depth counter both to the L2 MSHRs and to the prefetch queue entries to control pointer and recursive pointer prefetching uniformly. GRP initializes the counter on an L2 miss: for *pointers*, it sets the value to one, and for *recursive pointers*, it sets the value to six. Thus the only difference between pointer prefetching and recursive pointer prefetching is their initial counter value.

When GRP fetches a pointer-hinted missing line, it starts the pointer prefetching engine on the returned line. The engine checks the counter. If it is zero, it stops queuing prefetches. Otherwise, it decrements the counter, and queues prefetches for pointers in the returned line. The engine thus terminates after one level for pointers and six levels for recursive prefetching. We prefetch two cache blocks for each pointer based on our statistics that the typical structure size in the SPEC benchmarks is less than 64 bytes (one L2 cache block in our configuration) as we have discussed in Section 4.1.2. Two blocks are sufficient to cover structure alignment.

4.2 Encoding Compiler Hints

This section describes encoding of our compiler hints and its performance impact. We duplicate a whole set of Alpha memory instructions using unused opcodes. This simulates one additional bit in the opcode of each memory instruction which denotes if this instruction is a regular memory instruction or a compiler-marked one. To encode compiler hints, we use the four most significant bits from the 16-bit displacement field. We use three bits to denote if a load is marked as *spatial*, *pointer*, or *recursive*, one bit each. We use this encoding to compare pointer prefetching, SRP, and GRP. For variable-size region prefetching

Benchmark	All Insts		Marked Insts	
	low	up	low	up
164.gzip	0	111	0	111
168.wupwise	0	8	0	8
171.swim	0	112	0	0
172.mgrid	0	144	0	0
173.applu	0	144	0	8
175.vpr	0	232	0	76
177.mesa	0	10148	0	10145
179.art	0	56	0	56
181.mcf	0	600	0	576
183.equake	0	84	0	0
186.crafty	0	262	0	262
188.ammp	0	2200	0	2200
197.parser	0	104	0	104
254.gap	0	336	0	336
256.bzip2	0	16	0	16
300.twolf	0	88	0	88
301.apsi	0	120	0	8
sphinx	-8	548	0	548

Table 4.4. Bounds of memory instruction displacement fields

Benchmark	#mem insts	#out-of-range insts	#marked out-of-range insts	code size increase (%)	performance impact (%)
177.mesa	26777	1181	202	0.7	0 23%
sphinx	6335	21	0	0	-0.03%

Table 4.5. Performance impact of using 12-bit displacement field

(GRP/Var), we use three bits to encode region size and reserve one bit for evict-me, which makes a total of 4 hint bits.

Taking 4 bits away from the displacement field reduces the effective range of the displacement field. Table 4.4 lists the upper and lower bounds of the displacement fields for all memory instructions and for compiler-marked instructions only. We observe that a 12-bit displacement field is sufficient for all but 2 benchmarks, *sphinx* and *mesa*, if we use the 12 bits to represent an unsigned displacement, which gives us a range from 0 to 4095. As shown in Table 4.5, less than one half percent of total memory instructions use displacements out of this range. For *sphinx*, none of them are marked and only a very small number are marked in *mesa*.

We measure the worst case performance impact of using 12-bit displacement field by eliminating the displacement fields from all *out-of-range* memory instructions in *sphinx* and *mcf*. Given a memory instruction, *ld $r1, d($r2)*, where *d* is greater than 4095 or less than 0, we insert *lda $r2, d($r2)* in front and *lda $r2, −d($r2)* behind, and remove *d* from the memory instruction. Note that *lda $r2, d($r2)* is an arithmetic operation that allows a 16-bit immediate field and changes the value of *r2* to *r2+d*. The new set of instructions is semantically equivalent to the original memory instruction. The 12-bit version bloats the code size slightly for *mesa* by 0.7% and shows no change for *sphinx*. We run the original executable and the new executable using 12-bit displacement fields five times each on an Alpha 21264 machine and choose the fastest execution time from the five runs. The new code is negligibly slower for *sphinx* and slightly faster for *mcf*. The variation is within the measurement noise level. We conclude that using 12-bit displacements has little impact for the 18 benchmarks.

4.3 Compiler Analysis Framework

This section describes the analyses for the five classes of hints (*spatial*, *size*, *indirect*, *pointer*, *recursive pointer*) that guide the L2 prefetching engine. We implement these analyses in the Scale compiler and use them to generate these hints automatically for both C and Fortran programs.

4.3.1 Spatial Locality Analysis for Arrays

In GRP, the compiler predicts which misses truly have spatial locality, examining arrays in Fortran and C, and spatial pointer accesses to structures in C. The compiler uses locality analysis to mark references with the spatial hint annotation, and the compiler back-end augments the special load instruction with a spatial hint. The prefetch engine then prefetches only misses with marked spatial references and does not prefetch misses without spatial marks. We describe our array analysis and then our spatial pointer analysis.

```
integer a[N][M], B[N]
do j=1, m
 do i=1, n
   ...a(i,j)...
   ...c(b(i),j))...
```

Figure 4.2. Fortran array

```
T ** buf;
...
buf = malloc(...);
...
buf[i] = malloc(...);
...
for (i=0; i<m; i++)
 for (j=0; j<n; j++)
     ... buf[i][j] ...
```

Figure 4.3. C heap array

```
T *p, *s;
...
for (; p < s; p += c) {
  /* if T is a primitive type */
  ...*p...;
  /* if T is a structure */
  ...p->f...;
}
```

Figure 4.4. C induction pointer

```
struct t {
  T      f;
  struct t * next;
}
struct t *a;

while(...) {
  ...a ->f...;
  a = a->next;
  ...
}
```

Figure 4.5. C recursive pointer

We augment prior work that statically detects spatial locality by extending dependence testing [78, 111]. Dependence testing first finds induction variables and then detects when the spatial dimension (the row in C, the column in Fortran) is accessed as a function of the index variable, and whether it is the inner or outer nesting level. The dependence testing detects locality only for *affine* subscription expressions, i.e., linear functions of loop induction variables. Our approach marks references with either inner or outer loop spatial locality. The typical array reference with spatial locality is accessed in its spatial dimension in an innermost loop. For example, we mark *a(i,j)* in Figure 4.2, assuming column-major Fortran storage. The compiler also marks arrays with spatial locality that cross larger distances within a deep nest or between two nests (*inter-nest reuse*). We use the Level 2 cache size as our upper bound on the distance of the spatial reuse we mark, assuming that the Level 2 cache has sufficient set associativity to avoid conflict misses and exploit the reuse.

```
generate_spatial_hints()
{
  /* recognize induction variables including pointers*/
  induction_variable_recognition();
  /* perform dependence testing */
  dependence_testing();

  for (each loop) {
    /* generate basic spatial hints */
    for (each memory reference r in the loop) {
      if (r is an array reference) {
        if (r has spatial reuse in the enclosing innermost loop)
          mark r spatial;
        else {
          compute r's reuse distance when applicable;
          if (reuse distance < the Level 2 cache size)
            mark r spatial;
        }
      }
      if (r is a loop induction pointer)
        mark r spatial;
    }

    /* propagate spatial hints for loop induction pointers */
    do {
      for (each memory reference r) {
        if (r is a loop induction pointer)
          mark *r as spatial;
        else if (r is a->f && a is marked as spatial) {
          mark a->f as spatial;
      }
    } while (no new hints generated);
  }
}
```

Figure 4.6. Algorithm for generating spatial hints

If the compiler determines the loop bounds and step sizes, it can compute the reuse distances accurately at compile time. For arrays with spatial intra- and inter-nest locality, it computes the reuse distances. It marks as spatial all array references with spatial locality with a known distance less than the Level 2 cache size. When the compiler does not know the reuse distances statically due to symbolic loop bounds and uncertain executions paths, it estimates the reuse distance based on the nesting level of the loop. The compiler is conservative when reuse distance is unknown: we mark a reference as spatial only if its spatial reuse is in the innermost enclosing loop.

The above analysis works well for Fortran arrays and heap arrays in C if the array elements are referenced as subscript expressions. We handle heap arrays in C using the same analysis. In Figure 4.3, *buf* is a heap array with type *T***. In addition to detecting the obvious spatial reuse of $buf[i][j]$ when j is an loop induction variable, the compiler is able to find the spatial reuse of $buf[i][a*j+b]$ when a and b are constants.

4.3.2 Spatial Locality Analysis for Pointer Dereferences

To prefetch pointer references that show spatial locality, as illustrated in Figure 4.4, the compiler performs loop induction variable recognition on pointers that are repeatedly incremented by a constant. The type T in Figure 4.3 and Figure 4.4 does not have to be a primitive type. We treat pointer p as a special integer, and insert spatial hints for $*p$ or $p \rightarrow f$, if constant c is small. Our analysis of L2 cache misses shows that almost all spatial reuses in C code are covered by regular spatially local array references along with the cases in Figure 4.3 and Figure 4.4.

Figure 4.6 summarizes the algorithm used for generating spatial hints for both arrays and spatial pointer accesses. The first part of the algorithm inserts the spatial hints for arrays and loop induction pointers, and the second part propagates spatial hints to the uses of loop induction pointers. This algorithm is intra-procedural and flow insensitive, and it marks only references enclosed in loops.

4.3.3 Indirect Array Access Analysis

The compiler also detects and marks indirect array accesses, such as $c(b(i), j)$ in Figure 4.2. In particular, it looks for access patterns in the form of $a(s*b(i)+e)$ where s and e are constants, and i is a loop induction variable. Dependence testing detects the spatial reuse of $b(i)$ in the standard way. We add a simple analysis that detects when a sequentially accessed array is used as an index into another array (a in this example), and generates an indirect prefetch instruction using the address of $b(i)$ and the base address of array a, as described in Section 4.1.3.3.

4.3.4 Variable-Size Region Analysis

The compiler detects and marks array references within singly nested loops for variable-size region prefetching. For an array access with a pattern of $a(b*i+c)$ and an array element size of e, the compiler encodes $b*e$ into a three-bit value x such that $x < 7$ and 2^x is closest to $b*e$. We reserve the encoding value 7 for fixed-size region prefetching. The compiler marks the upper bound of the loop induction variable i. The two hints are used to control the region size as described in Section 4.1.3.2.

4.3.5 Pointer and Recursive Pointer Analysis

As with spatial locality, the compiler can improve the accuracy of hardware-based pointer prefetching by restricting it to misses on a load to a field from a structure that contains a pointer or recursive field. We mark a field reference as *pointer* if a pointer field from the same structure is accessed in the same loop. We mark a pointer update to be *recursive* if it updates itself in a loop with an object of the same data type. For example, in Figure 4.5, *a* is updated with its *next* field, which points to a structure of the same type, *struct t*. This idiom analysis simply identifies pointer updates in a loop that use a field with the same type and marks them as recursive pointer updates.

We mark pointer accesses with the spatial hint for references to arrays of pointers. For example, Figure 4.3 shows an array reference $buf[i]$, whose access pattern results in a spatial hint from the compiler. Furthermore, each $buf[i]$ points to a heap array, so the compiler marks it with the pointer hint as well. GRP will then use the address to prefetch the pointed-to array.

The algorithm to generate pointer and recursive pointer reference hints is shown in Figure 4.7. It is complementary to the spatial marking algorithm for pointers shown in Figure 4.6.

```
generate_pointer_hints()
{
  for (each field access) {
    if (a pointer field from the same structure
        is accessed in the same loop)
      mark the field access as pointer;
    if (the field access updates a recurrent pointer)
      mark the field access as recursive pointer;
  }

  for (each array reference marked as spatial} {
    if (the reference points to a heap array)
      mark the reference as pointer;
  }
}
```

Figure 4.7. Algorithm generating pointer and recursive pointer hints

4.4 Compiler Implementation

The Scale compiler infrastructure inserts the prefetch hints [6]. In the previous chapter, we discussed compiler implementation of our cache replacement algorithms in Scale. The implementation strategy for compiler hints for prefetching is similar. By default, we turn on all scalar optimizations available in Scale. After these conventional optimizations, we apply our prefetching hint analysis and cache replacement analysis just before the back-end code generation phases. The hints are attached to the memory instructions as comments.

We post-process the annotated assembly code to generate assembly files containing compiler-hinted instructions. We then use the native compiler and linker to generate the executable.

4.5 Experimental Evaluation

In this section, we compare the performance benefits of SRP, GRP, and unified stride prefetching for the SPEC CPU2000 benchmarks, and one additional benchmark, *sphinx*. We demonstrate that GRP provides a compelling balance between higher performance and increased memory traffic among the three prefetching techniques. We demonstrate the effectiveness of the compiler generated size information, and the sensitivity of our results to the compiler's heuristic for computing the useful distance of spatial locality. We conclude

Processor Core	1.6 GHZ, 4-way, 64-entry RUU
L1 cache	split 64K, 64-byte block, 2-way, 8 MSHRs, 3 cycles
L2 cache	1M, 64-byte block, 4-way, 8 MSHRs and 8 prefetching MSHRs, 12 cycles
Memory	Rambus DRAM, 4 channels, 800M HZ

Table 4.6. System parameters

with case studies and a discussion of the characteristics of the remaining benchmarks for which GRP does not eliminate main memory accesses as a significant source of performance loss.

4.5.1 Experimental Methodology

We simulate program binaries on a version of sim-outorder [15] with scheduled region prefetching (SRP) [69] added to the simulator. We added the hardware pointer prefetching mechanisms, and modified the simulator to accept compiler hints and schedule prefetches accordingly if the binaries contain the hints. The system configuration is shown in Table 4.6. We use the Alpha-ISA and configure the simulator as a 1.6 GHz, 4-way issue, 64-entry RUU (reorder buffer), out-of-order core with 64K 2-way split Level 1 caches and a unified 4-way 1MB Level 2 cache. This cache hierarchy is combined with an effective 800-MHz, 4-channel Rambus memory system. The L1 and L2 latencies are 3 and 12 cycles, respectively. Each cache contains 8 MSHRs. For SRP, the prefetching queue size is 32 and uses LIFO scheduling. The stride predictor [92] uses a 4-way history table with 1K entries. There are 8 entries in each of 8 streaming buffers sharing the history table. Finally, we use the SimPoint [93] tool set to select a representative starting point beyond the program's initialization phase. We simulate for 200M instructions from that point. Previous work shows this simulation method catches statistical significance of program execution [69, 93].

We use the 17 SPEC CPU2000 C and Fortran benchmarks that the Scale infrastructure is able to compile correctly, plus *sphinx*, a speech recognition application [68]. Table 4.7 lists these benchmarks, along with statistics on memory instructions and the number and type of compiler hints generated. The second column contains the total number of

Benchmark	mem insts	spatial	pointer	recursive	ratio(%)	indirect
164.gzip	1873	433	268	0	37.1	9
168.wupwise	507	152	0	0	30.0	0
171.swim	250	115	0	0	46.0	0
172.mgrid	314	232	0	0	73.9	3
173.applu	1491	858	0	0	57.5	0
175.vpr	4230	1001	682	74	33.8	84
177.mesa	26777	4532	4419	76	32.8	9
179.art	1016	732	278	0	77.6	0
181.mcf	845	168	287	201	60.8	0
183.equake	1679	597	473	0	51.3	7
186.crafty	11702	1994	736	0	21.6	5
188.ammp	6271	1043	1158	0	33.2	5
197.parser	4090	915	932	1263	70.2	2
254.gap	29781	5102	11243	0	52.6	36
256.bzip2	698	279	59	0	48.3	14
300.twolf	12397	2080	2577	1398	45.1	38
301.apsi	3225	1001	0	0	31.0	0
sphinx	6335	2211	1129	364	46.8	106

Table 4.7. Number of compiler hints for each benchmark

static memory reference instructions. Columns 3 to 5 show the number of instructions the compiler marks as *spatial*, *pointer*, and *recursive*. Note that the compiler can mark an instruction both *spatial* and *pointer*. Column 6 lists the percent of static memory operations with hints, and Column 7 shows the static number of indirect prefetch instructions. We do not present further results for *crafty* in subsequent results because its L2 miss rate is negligible (0.4%).

4.5.2 Comparison of Region Prefetch and Pointer prefetching

In this section, we present the effects of both hardware pointer and recursive pointer prefetching. We show that explicit pointer prefetching is generally subsumed by aggressive spatial prefetching (SRP or GRP). We then discuss the effect of compiler pointer hints.

We apply pointer prefetching alone to all benchmarks, which unsurprisingly has little effect on the Fortran benchmarks. Eight C benchmarks show a significant performance improvement, notably a 47.8% boost for *equake*, a 15.2% increase for *mcf*, and an 16.8% improvement for *sphinx*, as shown in Figure 4.8. Pointer prefetching outperforms SRP only for *twolf* and *sphinx*, by 2%. In all other cases, SRP performs much better than pointer or

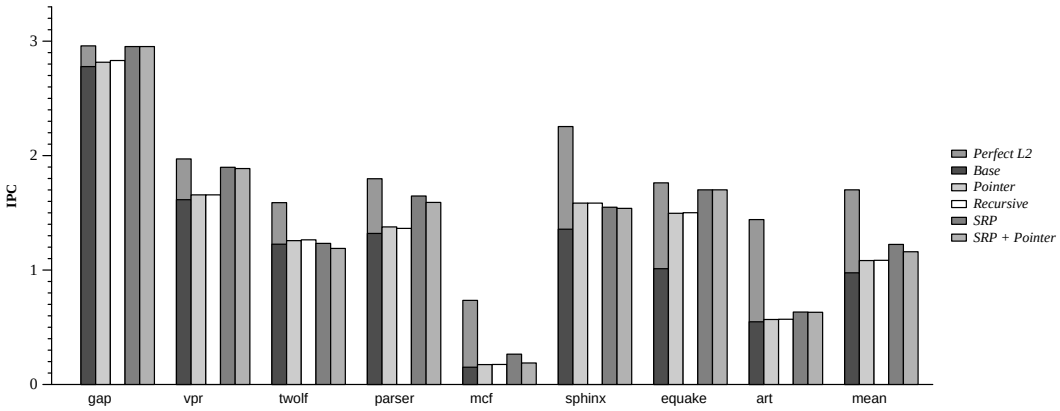

Figure 4.8. Performance gains from pointer prefetching

```
for (i = 0; i < ARCHnodes; i++)
  for (j = 0; j < 3; j++)
    disp[disptplus][i][j] = disp[disptplus][i][j] /
                (M[i][j] + Exc.dt / 2.0 * C[i][j]);
```

Figure 4.9. Code segment in 183.equake (quake.c)

recursive prefetching. Applying SRP and pointer prefetching together gives little benefit and sometimes degrades performance due to much higher bandwidth consumption, which can result in fewer successful prefetches.

For *equake*, the performance gain is not from pointer structure traversal as expected. It stems instead from prefetching arrays of pointers from the heap arrays. Figure 4.9 is a typical loop in *equake*. The heap array, *disp*, is declared as *double ***disp*. A miss to *disp[disttplus][i]* will trigger pointer prefetching from the surrounding elements in the same cache line, which point to heap arrays such as *disp[disttplus][i+1][]*. Region prefetching hides the latency of this kind of access very well and subsumes pointer prefetching in this case.

In *mcf*, the performance gain comes from a loop that sequentially resets a field in each object in a heap array as shown in Figure 4.10. Pointer prefetching happens to prefetch the objects accessed later. Region prefetching also hides the latency of this kind of access which is essentially sequential.

```
for( node = root, stop = net->stop_nodes;
     node < (node_t*)stop;
     node++ )
        node->mark = 0;
```

Figure 4.10. Code segment in 181.mcf (mcfutil.c)

GRP with pointer and recursive hints shows performance gains similar to SRP for seven of the eight benchmarks, but with lower average memory traffic as shown in Figure 4.11. We show the IPC in each set of bars for each benchmark and list the traffic normalized to the base above each set. On average, the marked pointer scheme performs 2% worse than the hardware-only pointer scheme. The gap between them mostly comes from *mcf*, where SRP/Pointer gains 15% over the base while GRP/Pointer shows little improvement. Note that SRP/Pointer gains in the loop as shown in 4.10. Our compiler pointer analysis does not mark *node→mark* as *pointer* since the loop does not contain any pointer field references. For *equake*, GRP/Pointer performs as well as SRP/Pointer. Our compiler algorithm marks *disp[disptplus][i]* as *pointer*.

GRP/Pointer causes less traffic increase, 20% compared to 35% for SRP/Pointer on average. The traffic increase caused by pointer prefetching in SRP is not as dramatic as that caused by region prefetching, as we shall discuss in Section 4.5.4. This is because pointer prefetching prefetches only two cache blocks for each potential pointer. Thus, a false pointer in this pointer prefetching scheme does not have much penalty.

4.5.3 Comparison of Stride Prefetching, SRP, and GRP

In this section, we compare stride prefetching with SRP and GRP. GRP uses all the compiler analyses, including variable region sizes. The end of this section compares variable and fixed region sizes, and finds that variable size region pefetching decreases bandwidth requirements for 3 programs.

Figures 4.12 and 4.13 show the performance of SRP, GRP, and stride prefetching for integer and floating point benchmarks, respectively. In most cases and on average, SRP

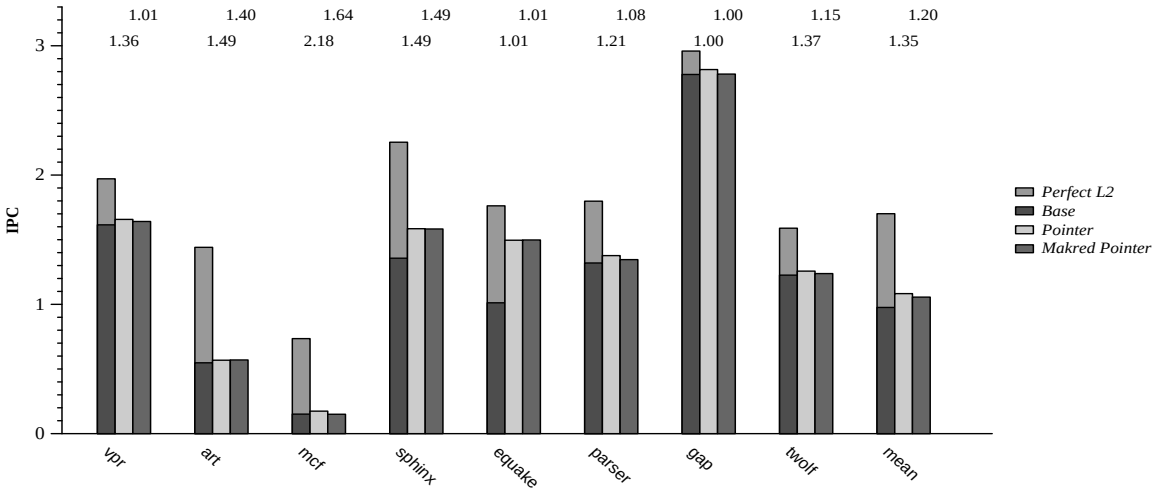

Figure 4.11. Pointer vs. marked pointer prefetching

and GRP both perform better than stride prefetching. For 10 benchmarks, SRP improves performance to within 10% of a perfect L2 cache. For *swim*, GRP performs over 10% better than SRP due to its lower traffic. It also outperforms SRP for *art* and *ammp*. Due to indirect prefetching, GRP is 4% faster than SRP for *bzip2*. For *gzip*, *mcf*, *parser*, and *gap*, the IPC of GRP is at least 2% less than that of SRP. A typical reason is that the compiler misses opportunities to exploit locality outside of loops.

Although we detect indirect references in 11 benchmarks, indirect prefetching shows significant speedups for only *vpr* and *bzip2*. For *vpr*, the indirect references show high spatial locality. SRP thus performs as well as GRP, but with 50% additional traffic. *Bzip2* is one of the benchmarks where SRP does not perform well. With indirect prefetching, the gap from a perfect L2 cache is reduced to 12.5% from 15.9%, with only 15% of the memory traffic of SRP.

In terms of both performance and memory traffic, GRP using a variable region size (GRP/Var) and a fixed region size (GRP/Fix) differ in only three benchmarks, *mesa*, *bzip2*, and *sphinx*. Table 4.8 shows that for *mesa* and *bzip2*, both strategies deliver roughly the same performance while GRP/Var results in much less traffic than GRP/Fix, as we discuss in Section 4.5.4. For *sphinx*, GRP/Var has 5.8% lower performance than GRP/Fix, but

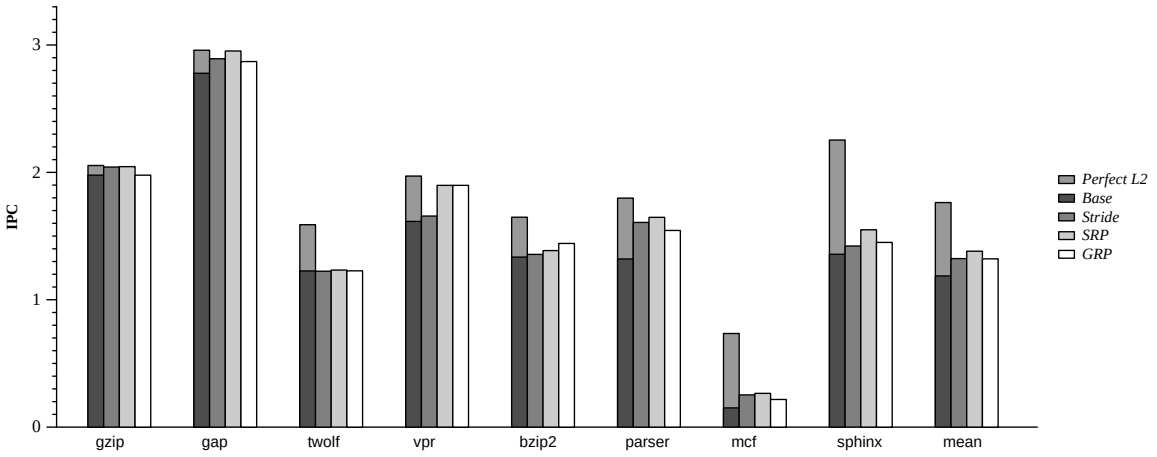

Figure 4.12. Performance gains from region prefetching and stride prefetching for integer benchmarks

	GRP Traffic		Region Size Distribution (%)			
	Var	Fix	2	4	8	64
mesa	1.11	6.55	90.3	9.5	0.1	0.1
bzip2	1.47	4.97	76.8	22.4	0.0	0.8
sphinx	2.09	11.66	82.9	1.0	16.1	0.0

Table 4.8. GRP/Var versus GRP/Fix

benefits from an 82% traffic reduction. The compiler cannot guarantee that there is spatial locality, so it chooses small prefetch regions, and misses some opportunities.

4.5.4 Prefetching Accuracy, Coverage, and Memory Traffic

Although SRP and GRP provide comparable performance, SRP consumes much more bandwidth than does GRP. Figure 4.14 shows the normalized memory traffic for the three prefetch schemes. SRP increases memory traffic from 2% to a factor of 25.5 times over no prefetching. GRP generates a mean of only 23.0% additional traffic compared to no prefetching, versus a SRP's mean increase of 180%. GRP eliminates over 20% of the total memory traffic for ten of the seventeen benchmarks as compared to SRP, and over 50% for six benchmarks. The traffic for stride prefetching is 11% less than GRP, but stride prefetching achieves only 69% of the performance improvement that GRP does.

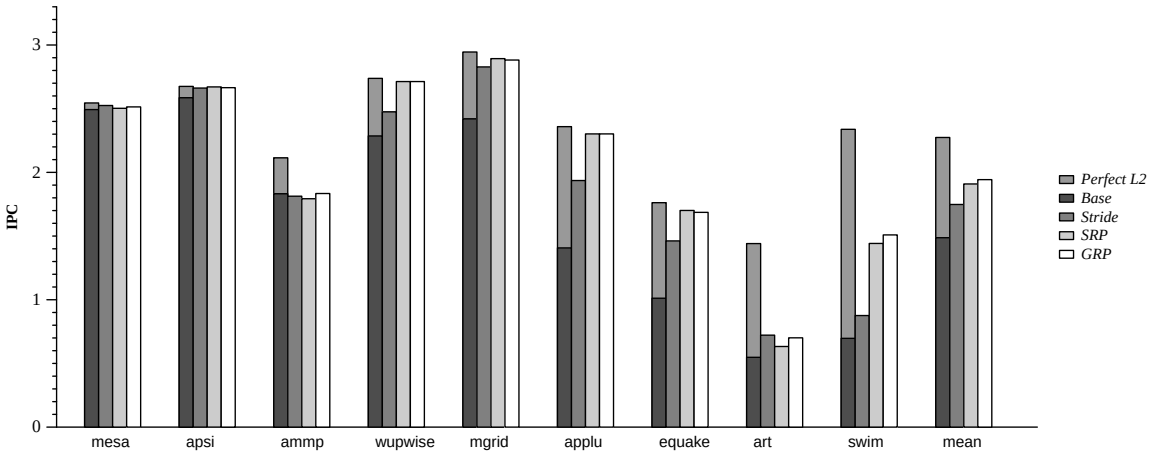

Figure 4.13. Performance gains from region prefetching and stride prefetching for floating-point benchmarks

Compared to GRP/Fix, GRP (GRP/Var) cuts memory traffic significantly for three benchmarks while showing the same traffic for the others. Table 4.8 lists the three benchmarks and their traffic increase compared to no prefetching in columns 1 through 3. The subsequent four columns show the distribution of prefetching requests by region size (no requests with regions of 16 or 32 blocks). We observe that GRP/Var prefetches only one additional block (region size = 2) in most cases, due to the poor spatial locality of these references.

Table 4.9 shows both prefetching accuracies and coverage for the three prefetching techniques that we implemented. We use the percentage reduction in L2 misses as a metric for coverage. On average, SRP provides the best coverage and the worst accuracy. Stride prefetching trades the lowest coverage with the highest accuracy. GRP obtains the best of both worlds: an accuracy that is closer to stride prefetching, but coverage closer to that of SRP.

Since the normalized traffic in Figure 4.14 does not reflect the absolute bandwidth consumption of each benchmark, we also list the actual memory traffic, in bytes, of each benchmark in Table 4.9. On average, SRP consumes 99.8% more memory bandwidth over

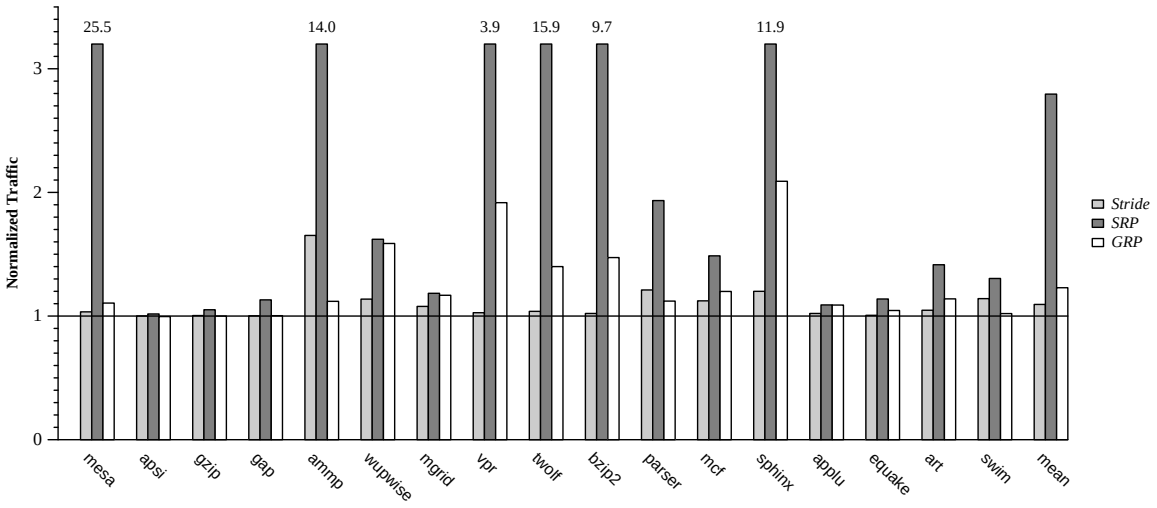

Figure 4.14. Normalized traffic

the no-prefetching system. GRP and stride prefetching produce an 18.3% and a 10.1% increase in memory requests, respectively.

4.5.5 Compiler Sensitivity

We explored the sensitivity of our results to the compiler policy by implementing both more and less aggressive variants of the scheme described in Section 4.3. The more aggressive policy marks a reference as *spatial* even if its reuse distance is greater than the L2 cache size. The more conservative scheme marks a reference as *spatial* only when its reuse sits in the innermost loop. Compared to our default GRP policy, the aggressive policy degrades performance by 2% overall and increases traffic by an additional 5%. It degrades *swim* and *art* by 8% and 4.4% each compared to the default. The conservative scheme shows little effect on memory traffic compared with GRP, but causes performance losses of an average of 5% across the benchmark suite. Compared to the default scheme, it degrades *applu* and *equake* by 14% and 34%, respectively.

Benchmark	Base		Stride			SRP			GRP		
	Miss Rate	Traffic	Cov.	Accu.	Traffic	Cov.	Accu.	Traffic	Cov.	Accu.	Traffic
mesa	9.3	51k	60.9	93.2	53K	29.3	0.8	1305K	43.5	70.1	56K
apsi	25.0	85K	79.2	99.8	85K	96.4	95.8	86K	88.8	97.6	84K
gzip	25.3	182K	65.2	99.8	183K	76.3	94.4	192K	0.0	91.2	182K
gap	46.8	179K	66.7	99.6	179K	97.6	86.3	202K	52.8	99.3	179K
ammp	15.3	594K	-7.8	23.1	982K	-7.8	0.9	8340K	0.7	27.5	665K
wupwise	73.1	486K	42.5	75.4	553K	96.3	60.2	788K	96.2	61.6	772K
mgrid	43.9	504K	77.9	89.9	544K	87.5	80.7	597K	85.6	81.7	589K
vpr	40.2	730K	15.9	85.5	749K	86.3	27.6	2820K	76.4	49.4	1399K
twolf	12.6	1125K	0.0	27.3	1167K	15.9	4.2	17878K	3.2	28.7	1575K
bzip2	22.4	1163K	8.4	85.7	1186K	27.2	5.3	11255K	37.1	51.6	1713K
parser	33.4	1450K	67.4	75.0	1756K	77.5	44.7	2804K	56.0	82.5	1625K
mcf	61.6	43901K	51.0	80.5	49284K	24.7	53.9	65263K	5.4	51.1	52656K
sphinx	65.9	1208K	12.6	27.3	1449K	42.8	4.7	14429K	21.7	20	2521K
applu	58.0	2578K	62.6	95.7	2631K	96.9	89.0	2810K	96.9	89.2	2806K
equake	59.8	3628K	75.6	99.2	3649K	96.3	86.9	4127K	95.2	95.3	3790K
art	44.4	20229K	17.3	99.7	21189K	8.6	40.6	28632K	20.9	78.0	23031K
swim	57.8	7861K	34.6	70.8	8966K	67.3	65.2	10249K	68.2	96.5	8021K
average	40.9	5057K	42.9	78.1	5565K	59.9	49.5	10105K	49.9	68.9	5981K

Table 4.9. Prefetching accuracy, coverage, and memory traffic

4.5.6 Performance Improvement and Miss Reduction

An X percent total miss reduction does not necessarily result in the same performance improvement in modern architectures. However, we observe that performance improvement is still a function of miss reduction by adding two variables: the base performance and the improvement space. Using guided region prefetching as an example, we can predict the IPC of GRP given the base IPC without prefetching and the IPC under a perfect L2 cache. Given an application, let its base IPC be B and the IPC with a perfect L2 cache be L. Assume that GRP causes an X percent miss reduction. Then we predict the IPC of GRP, P, to be $B + (L - B) * X/100$, i.e., the IPC increment is proportional to the miss reduction. Figure 4.15 verifies our model. Let the model *error* be $|(P - G)/G|$ where G is the IPC of GRP. Two bars for each benchmark show the IPC with GRP and the predicted IPC using our model. We list the error, as a percentage, on the top of each set. On average, this model yields only 1% error. For only three benchmarks, *mcf*, *sphinx*, and *swim*, is the model error greater than 5%. This result suggests that we can predict performance gain by using just miss reduction if we know first how much performance loss is due to these misses.

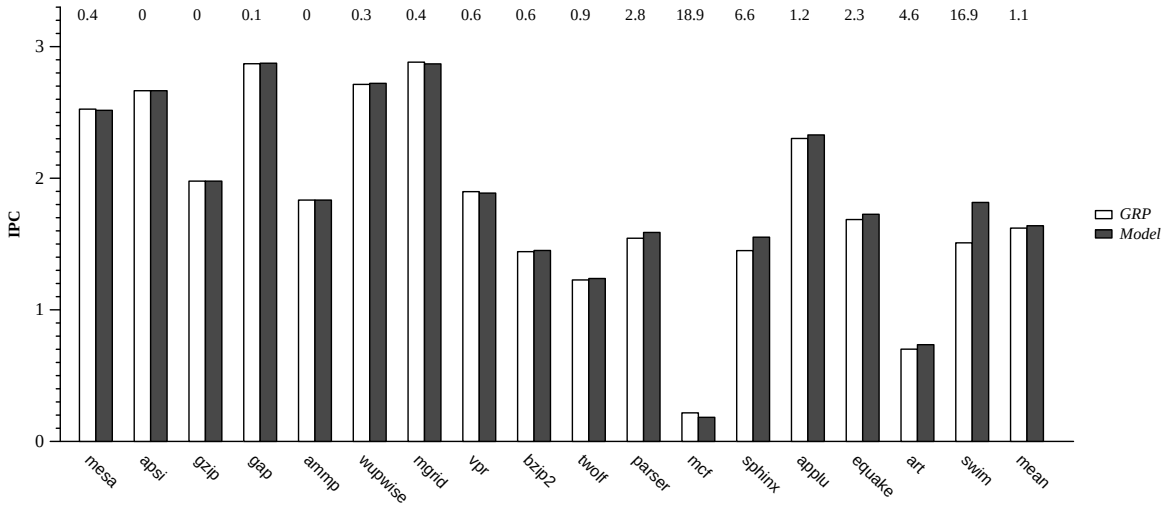

Figure 4.15. Miss reduction versus performance improvement

4.5.7 Case Studies

In this section, we discuss the remaining benchmarks where there is still a significant L2 gap after region prefetching. We study each specific benchmark on where region prefetching gains or loses performance and where it delivers high or low prefetching accuracy.

4.5.7.1 Remaining L2 Cache Misses

Seven of the benchmarks show a gap of greater than 15% between SRP and a perfect L2 cache. We list them in Table 4.10, with a description of the key causes of the misses, obtained by analyzing the source.

With its more accurate prefetching, coupled with indirect accesses and pointer prefetching, GRP is able to bring *bzip2* and *ammp* under 15% but the gap remains large. *Swim* has a low IPC due to pathological array conflicts. We can prevent that benchmark from being memory-bound by manually applying loop distribution and loop permutation [20]. We observe that *art* is bandwidth bound. While GRP reduces traffic and increases performance over SRP by 10.7%, the performance gap is still large. Larger caches and wider channels improve *art* appreciably. For *sphinx*, the hash table lookup usually touches only a small number of adjacent hash slots in a short loop. Prefetches occur simply too late to tolerate

Benchmark	GRP Performance Gap (%)	L2 Miss Causes	Ratio (%)
171.swim	38.32	transpose array access	92.08
179.art	56.07	bandwidth	24.26
		transpose heap array access	35.92
181.mcf	63.94	tree traversal	60.70
188.ammp	15.18	linked list traversal	88.64
256.bzip2	15.89	indirect array reference	49.68
300.twolf	22.40	linked list and random pointers	35.37
sphinx	31.28	hash table lookup	28.79

Table 4.10. Level 2 miss characteristics

the latencies. Finally, *mcf* and *twolf* contain heavy traversals of short linked lists and tree data structures, making them poor matches for the GRP pointer prefetching or spatially-based schemes.

4.5.7.2 Discussion of Prefetching Accuracy

Even for GRP, the accuracy is low for several benchmarks. In this section, we discuss case by case where SRP or GRP gains and loses prefetching accuracy.

In the simulator, we track down the PC values (code locations) where the prefetch engine issue prefetches. We then categorize the number of issued prefetches and useful prefetches based on PC values. By mapping those values back to the source code, we are able to figure out which memory references cause high or low prefetching accuracies.

References in library calls or outside of loops

Our compiler algorithm does not analyze library calls for spatial locality. It also skips references not enclosed by loops. Two benchmarks, *gzip* and *parser*, are affected. SRP improves the IPC of *gzip* from 2.01 to 2.09. The prefetches are predominantly from library calls to *memcpy()*. Our compiler does not touch the libraries. This is the only benchmark where prefetching in a library call dominates the IPC gain. A slightly different scenario is seen in *parser*. SRP beats GRP by 15.9%. The useful prefetches that GRP misses reside in one function call, which contains no loops. The compiler thus does not generate any hints for the references in the function. However, the function is frequently called *within*

```
DO 190, L = 1, K
   IF( B( J, L ).NE.ZERO )THEN
       TEMP = ALPHA*DCONJG( B( J, L ) )
       DO 180, I = 1, M
         C( I, J ) = C( I, J ) + TEMP*A( I, L )
   180  CONTINU
   END IF
190  CONTINUE
```

Figure 4.16. Code segment in 168.wupwise

loops. Detecting spatial reuse in these functions requires expensive and complicated inter-procedural analysis.

Arrays with linear indices

For arrays whose indices are linear expressions of loop induction variables, compiler spatial locality analysis can predict the reuse pattern very well. SRP usually generates high prefetching accuracy too. For *mgrid*, *applu*, and *apsi*, SRP enjoys more than 89% prefetching accuracy due to the substantial spatial locality in these benchmarks. The two prefetching schemes provide almost the same IPCs and memory traffic.

An exception occurs in *swim* and *art*. GRP improves the prefetching accuracy from 65% to about 96% for *swim* and from 40% to 78% for *art*. The unused prefetches originate from accesses to transposed two-dimensional arrays. GRP avoids this by predicting that the reuse distance is too large to be exploited by region prefetching

The 32% of useless prefetches in *wupwise* come mostly from the array reference A(I,L) in the code segment shown in Figure 4.16. The compiler marks A(I,L) with a spatial hint. We suspect the low accuracy is due to self-interference conflict misses among those references to $A[1:M,L]$ and the prefetches triggered by them.

Linked structures

Figure 4.17 shows a code segment from mcfutil.c in *mcf*. The references *node*→*basic_arc* and *node*→*basic_arc*→*cost* exhibit significant spatial locality based on our statistics. SRP for the two references exhibits more than 50% prefetching accuracy, and the useful prefetches at the two points account for a significant portion of overall useful prefetches. This

```
while( node != root )
{
    while( node )
    {
      if( node->orientation == UP )
          node->potential = node->basic_arc->cost
                          + node->pred->potential;
       else /* == DOWN */
        ....
       tmp = node;
       node = node->child;
    }
     ...
}
```

Figure 4.17. Code segment in 181.mcf

```
for( netptr = dimptr->netptr; netptr; netptr = netptr->nterm) {
  oldx = netptr->xpos;
  if( netptr->flag == 1 ) {
    newx = netptr->newx;
    netptr->flag = 0;
  } else {
    newx = oldx;
  }
  *costptr += ABS(newx - new_mean) - ABS(oldx - old_mean);
}
```

Figure 4.18. Code segment in 300.twolf

is because *node* and *node→basic_arc* both point to sequential heap arrays, and the linked list structure is built contiguously in memory. Our compiler detects the link structure here but of course does not mark the references as *spatial*. An opposite case is in *twolf*. Figure 4.18 shows a typical code section. The loop traverses a linked list and the loop body is short. Unlike *mcf*, the list does not show much locality and SRP suffers low prefetching accuracy with little performance improvement.

Another interesting linked structure is shown in Figure 4.19. It is a code fragment from rectmm.c in *ammp*. The reference *(*atomlist)[i].who)→serial* accounts for a large portion of overall misses and the prefetching accuracy of SRP here is under 1%. Due to the list indirection of *i = (*atomlist)[i].next* and the pointer reference to *serial*, the reference has little spatial locality. The excessive prefetches generated by SRP at this point hurt perfor-

```
for( j=1; j< (*nodelist)[inode].innode -1 ; j++)
{
 i = (*atomlist)[i].next;
 if( ((*atomlist)[i].who)->serial > a1->serial)
 {  (*atomall)[imax++] = (*atomlist)[i].who;}
}
```

Figure 4.19. Code segment in 188.ammp

```
for (from_edge=0;from_edge<from_num_edges;from_edge++) {
    to_node = rr_node[from_node].edges[from_edge];
    to_rr_type = rr_node[to_node].type;
    ....
}
```

Figure 4.20. Code segment in 175.vpr

mance. GRP does not mark this reference as *spatial* and gains a little performance over SRP. The indirection makes it difficult to predict *who* in future iterations. By instrumenting the source code, we find the probability of **(atomlist)[i].next = i+1* is 61.4%. It is possible to fetch just *(*atomlist)[i+1].who* and prefetch the structure that *who* points to. However, this prefetch helps little in the L2 cache because the loop is too short and prefetching one-iteration ahead is too late. We are interested in checking whether this is more effective in L1.

Array references through indirect arrays

Indirect array references are a typical reason for low prefetching accuracy in *vpr* and *bzip2*. Figure 4.20 shows a code fragment from check_rr_graph.c in *vpr*. The prefetching accuracy of reference $rr\_node[to\_node].type$ is less than 20%. Our compiler does not mark it as spatial. However, the absolute number of useful prefetches at this point is also significant. SRP shows an 8.8% better IPC compared to GRP without indirect prefetching turned on.

Figure 4.21 shows a different case: the indirect array reference *quadrant[a2update]* does not show much spatial locality. Indirect prefetching reduces the misses at this point by nearly 60%. Note that the compiler needs to follow use-def links to figure out that *a2update* is defined by an array reference *zptr[bbStart + j]*, which is itself marked as *spatial*.

```
for (j = 0; j < bbSize; j++) {
  Int32 a2update     = zptr[bbStart + j];
  UInt16 qVal        = (UInt16)(j >> shifts);
  quadrant[a2update] = qVal;
  if (a2update < NUM_OVERSHOOT_BYTES)
    quadrant[a2update + last + 1] = qVal;
}
```

Figure 4.21. Code segment in 256.bzip2

4.6 Chapter Summary

Purely compiler-based prefetching techniques have difficulty managing the large latencies of modern main memories. Previous work shows that aggressive hardware prefetching addresses this issue effectively for applications with spatial locality, at the cost of potentially significant increases in memory bandwidth consumed. As the number of processors per chip increases, this bandwidth will become increasingly precious.

This chapter shows that a cooperative approach between compiler-based analysis and hardware-based aggressive prefetching provides benefits comparable to aggressive hardware prefetching with much lower traffic. Compiler techniques identify accesses that clearly possess spatial locality. Rather than use this information to attempt to schedule software prefetches—with the resulting complications of providing timely prefetches while minimizing instruction overhead—our system simply passes this access-pattern information to a hardware prefetching engine. The engine then generates prefetches for the L2 cache with low overhead. Compared to pure hardware prefetching, the compiler analysis saves bandwidth by avoiding useless prefetches to addresses with little locality.

We also extend the hardware prefetching engine to address pointer-based applications by aggressively prefetching any datum that appears to be a pointer. We see significant traffic benefits from having the compiler indicate pointer and recursive-pointer loads although it is not as dramatic as for region prefetching. For the SPEC2000 benchmarks, the aggressive spatial locality analysis subsumes pointer prefetches for most benchmarks, due to spatially local layouts of pointer-connected objects with respect to large regions. Even

sphinx, which we chose for its sparse irregular pointer behavior, benefits very little from pointer prefetching. It still remains to be seen whether this phenomenon will dominate the benchmarks that other researchers have used to show the importance of greedy pointer hardware prefetching [32].

With solely the spatial and indirect hints, the GRP compiler/hardware prefetch framework eliminates most L2-related stalls across the SPEC2000 suite, with comparatively modest increases in traffic. The remaining three benchmarks that are limited by L2 memory system performance are either bandwidth bound (*art*) or contain many irregular linked-lists and/or tree traversals (*mcf*, *twolf*), where memory-side prefetching may help. For the rest of the SPEC2000 suite, however, the GRP approach eliminates physical memory accesses as a performance bottleneck while making significantly more efficient use of the system bandwidth than similarly aggressive prefetch engines.

Stride prefetching generates less traffic than GRP but with significant performance loss. It also depends on a set of hardware features to control accuracy and thus introduces more hardware complexity than SRP or GRP. GRP is thus the most cost-effective design compared to SRP or a stride prefetcher. Using compiler hints, GRP reduces the bus utilization of SRP to a practical level while retaining its high performance. Since an L1 cache performance gap remains, we will discuss a *push* scheme built on GRP in the next chapter. We also discuss the combination of evict-me, GRP, and the data push technique.

CHAPTER 5

COMBINING CACHE REPLACEMENT AND PREFETCHING

Cooperative cache replacement and region prefetching both improve cache performance. One reduces cache misses and the other tolerates miss latencies. In this chapter, we discuss how these two techniques can work together. We observe that a better cache replacement policy can reduce the side effects of useless prefetches. The combination can further improve performance although region prefetching leaves only a very small space for improvement.

Region prefetching targets the L2 cache. Using compiler hints, guided region prefetching achieves high prefetching accuracy. This accuracy means the data prefetched into the L2 cache result in hits that serve the L1 misses. It is possible that a prefetched line arrives in the L2 cache before an L1 miss. In that case, we can hide part of the L1 latency by *pushing* the prefetched line into the L1 cache. The pushing scheme brings new pressure on cache replacement to both levels of cache. The pressure on the L2 cache comes from the additional write-backs due to the additional L1 replacements caused by the pushes. Different cache placement policies and replacement policies can either alleviate or aggravate this pressure. In Section 5.1, we discuss an implementation of the push scheme and the effects of different cache placement policies. In particular, we compare pushing data into LRU or MRU slots, where LRU is the conservative choice and MRU will be effective only when the line is used quickly.

The compiler-guided evict-me cache can be combined with region prefetching and the push scheme. The evict-me cache replacement policy can help reduce pollution resulting from prefetching. It reduces cache misses and is thus orthogonal to region prefetching

techniques, which tolerate latencies. Since region prefetches are triggered by on-demand misses, reducing the L2 misses by using evict-me will also reduce total prefetches as well as total memory traffic. However, the combination does not necessarily bring additional performance gains if region prefetching has already hidden these same latencies. In Section 5.2, we examine how well the aggressive region prefetching and evict-me work together.

5.1 L1 Push Scheme

In this section, we discuss our data push scheme, which pushes prefetched L2 cache lines into the L1 cache. We first describe the hardware implementation. We then experiment with various combinations of two different cache placement polices: MRU and LRU.

5.1.1 Hardware Description

We implement the push engine in the L2 cache controller. It is a pure hardware scheme, but it is implicitly driven by the compiler if the L2 prefetcher is compiler-guided. A *pull-based* prefetcher would follow a request-service-response path. First, the prefetches sends a request to the next lower level of the memory hierarchy. The lower memory will then send the data back when it is ready. A push-based prefetcher is simpler. It needs only a response process: when a prefetched block in the lower level is ready, the prefetcher simply pushes it to the higher level. In our design, the push stream shares a common response queue with the regular responses. The L2 cache line size is usually no smaller than the L1 cache line size. When the L2 cache line is bigger, we break an L2 cache line into units of L1 cache line size and push all the units into the L1 cache.

Following the default LRU cache replacement policy, a pushed or prefetched line will evict the LRU line in a set and be loaded into the MRU slot. By manipulating the LRU bits, we can make the new line reside in the LRU slot, the MRU slot, or other positions in the set. Previous work, which does not use compiler guidance, shows that keeping L2 prefetch

lines in the LRU slots yields the best performance [69]. In Section 5.1.2, we examine the impact of the MRU and LRU placement policies.

One major concern about this push scheme is address translation. In our implementation, the L2 cache is physically indexed and the L1 cache is virtually indexed. We need to translate a physical address to a virtual address when pushing a line into the L1 cache. The translation can be done using an ITLB (inverted translation look aside buffer). In contrast to a TLB, an ITLB is a small cache indexed by physical page addresses and each entry contains a virtual page address. Since the L2 prefetching region size is aligned and no bigger than a page size, all requests of a region sit in a single page. This ensures that all pushes of a region typically yield no more than one ITLB miss.

An alternative technique is to keep track of the virtual addresses of L2 prefetches. We can extract the virtual page address from an L2 demand miss. A region prefetching request is enqueued with this address. When a prefetch is issued, the prefetching MSHR for the prefetch will keep the virtual address and the push engine can use this address when the data is ready for pushing.

5.1.2 Results of the Push Scheme

For our experiments, we use the same set of benchmarks as in Chapter 4, which involve 16 Spec CPU2000 benchmarks and *sphinx*. The system configurations, including cache and memory settings, are also the same.

5.1.2.1 Push Performance

Table 5.1 shows our results with the L1 push scheme. For each benchmark, the leftmost five columns list the benchmark name and the IPC for the base case, perfect L1 cache, perfect L2 cache, and GRP using the default LRU placement policy (GRP/LRU). The rightmost five columns are percentage performance improvements over GRP/LRU. On average, the placement policies of the prefetched or pushed lines have a very small impact on performance. The worst case, GRP/LRU plus Push/LRU, is within a half percent

	IPC				Improvement over GRP/LRU (%)				
	Base	Perf L2	Perf L1	GRP/LRU	GRP/MRU	GRP/LRU Push/LRU	GRP/MRU Push/LRU	GRP/LRU Push/MRU	GRP/MRU Push/MRU
gzip	1.98	2.05	2.09	1.98	0.00	0.00	0.00	0.00	0.00
wupwise	2.29	2.74	2.90	2.71	0.18	3.21	3.17	3.54	3.35
swim	0.70	2.34	3.04	1.51	0.80	-3.64	0.66	-1.92	2.12
mgrid	2.42	2.95	3.08	2.88	0.00	3.19	0.83	3.44	0.90
applu	1.41	2.36	2.67	2.30	0.52	8.34	8.69	11.42	11.82
vpr	1.62	1.97	2.09	1.89	-0.05	2.01	1.37	2.70	1.85
mesa	2.52	2.57	2.61	2.53	0.12	-0.16	0.08	-0.28	0.00
art	0.55	1.44	2.18	0.70	-6.70	5.85	-6.28	6.85	-6.42
mcf	0.15	0.74	2.01	0.22	12.44	0.00	12.90	-0.46	12.90
equake	1.01	1.76	1.96	1.69	0.65	8.24	8.72	9.07	9.43
ammp	1.83	2.11	2.39	1.83	-0.33	0.05	-0.33	0.05	-0.38
parser	1.32	1.80	2.13	1.54	-0.32	3.89	4.15	3.82	4.08
gap	2.78	2.96	2.98	2.87	0.00	0.24	0.24	0.24	0.24
bzip2	1.26	1.59	1.83	1.40	0.07	0.36	0.79	0.57	0.93
twolf	1.23	1.59	2.06	1.23	-0.41	0.08	-0.41	0.00	-0.41
apsi	2.59	2.68	2.70	2.67	0.00	0.23	0.23	0.23	0.23
sphinx	1.36	2.25	2.64	1.45	-1.45	1.66	-0.07	1.72	-0.07
mean	1.33	2.01	2.40	1.62	0.27	1.93	1.95	2.35	2.28

Table 5.1. Performance impact of the L1 push scheme and placement policies

of the best case, GRP/LRU plus Push/MRU. Compared to GRP/LRU, GRP/MRU shows little improvement or degrades performance for all but *mcf*, where it improves the performance by 12%. GRP/MRU performs 6.7% worse than GRP/LRU for *art*, although it still improves over the base by 18%. The gap between GRP/MRU and GRP/LRU for *art* comes from a short bandwidth-bounded loop where the prefetches have short reuse distances and placing them into the LRU slots causes less pressure on the L2 cache when the prefetches are useless.

The push scheme brings us an additional 2% performance improvement over GRP/LRU. The best combination, GRP/LRU plus Push/MRU, offers an 11% performance boost for *applu*, 9% for *equake*, and 6% for *art*. For four benchmarks, *wupwise*, *mgrid*, *applu*, and *equake*, the combination of GRP and data pushing is able to beat a perfect L2 cache.

5.1.2.2 Push Accuracy and Coverage

Push accuracy is the number of used pushed lines divided by the total number of pushes. A pushed line is used if it is hit before its eviction. Since the push scheme is built upon the region prefetcher, we use the miss reduction over GRP as a measurement of coverage. Table 5.2 lists the L1 and L2 miss rates of GRP/LRU in the leftmost two

	GRP/LRU		GRP/LRU + Push/MRU		
	Miss Rate		Coverage		Accuracy
	L1	L2	L1	L2	
gzip	1.03	0.21	0.00	0.00	71.96
wupwise	1.74	0.02	67.82	16.67	52.17
swim	11.23	2.80	48.17	-43.37	95.71
mgrid	0.91	0.07	82.42	-11.43	86.68
applu	3.58	0.10	87.71	-86.73	88.09
vpr	2.24	0.13	35.27	2.29	36.46
mesa	0.35	0.02	-11.43	-4.76	3.45
art	49.59	17.90	11.72	6.69	59.48
mcf	51.87	22.71	-1.08	-0.04	9.53
equake	8.38	0.11	67.78	-14.02	84.49
ammp	4.54	0.46	-0.44	1.09	13.56
parser	5.09	0.50	25.93	-5.58	78.63
gap	0.29	0.08	48.28	-9.33	98.6
bzip2	5.26	0.48	8.56	-6.95	42
twolf	9.03	0.89	-0.55	0.34	5.89
apsi	0.22	0.01	40.91	0.00	65.23
sphinx	3.64	1.43	15.11	1.82	16.13
average	9.35	2.82	30.95	-9.02	53.42

Table 5.2. Coverage and accuracy of the L1 push scheme, GRP/LRU plus Push/MRU

columns. The miss reduction of GRP/LRU plus Push/MRU is shown in the middle two columns and the rightmost column lists its push accuracy. Because the L1 miss reduction affects the raw L2 miss rates, the L2 miss rates shown in this table are L2 misses over *all* data accesses. The push scheme is able to reduce L1 misses by at least 40% for 7 benchmarks over GRP and up to 87% for *applu*. Although the average miss reduction is 30%, we only see a 2.3% performance improvement for two reasons. First, the L1 gap is smaller: miss reduction at L1 yields less performance gain than at L2. Second, the push scheme causes additional pressure on the L2 cache, in fact increasing the L2 misses by 9% compared to GRP. However, the 9% increase does not cause much performance loss since the L2 miss rates of GRP are typically very low. Push accuracy is 53% on average, lower than the 69% accuracy of GRP on L2. This suggests that there is higher replacement pressure on the smaller L1 cache.

Table 5.3 shows the coverage and accuracy of the other schemes. The push accuracies are very close across different schemes. The small gap on the average coverage among

	GRP/LRU Push/LRU			GRP/MRU Push/LRU			GRP/MRU Push/MRU		
	L1	L2	Accu	L1	L2	Accu	L1	L2	Accu
gzip	0.00	0.00	71.43	0.00	0.00	73.79	0.00	0.00	71.15
wupwise	58.62	11.11	44.10	56.90	22.22	44.48	64.37	22.22	51.79
swim	44.61	-42.90	85.68	45.33	-12.30	91.18	48.62	-12.44	97.58
mgrid	78.02	-10.00	82.59	45.05	-1.43	80.09	46.15	0.00	81.76
applu	67.04	-69.39	66.28	66.48	18.37	70.21	87.43	17.35	91.20
vpr	28.57	0.00	28.21	31.25	-19.08	36.65	38.84	-19.85	46.82
mesa	-5.71	-4.76	3.34	-2.86	4.76	6.59	-5.71	4.76	6.71
art	10.32	5.46	54.07	10.43	-5.77	55.30	11.53	-6.10	60.25
mcf	-0.37	-0.03	9.40	-0.27	0.49	9.77	-0.94	0.51	9.91
equake	63.13	-15.89	74.12	63.37	12.15	76.08	68.74	11.21	86.31
ammp	-0.22	1.09	12.71	-0.22	-2.19	12.30	-0.44	-2.19	13.14
parser	25.93	-5.58	78.06	26.33	-1.99	78.92	26.13	-1.79	79.71
gap	48.28	-9.33	96.26	48.28	0.00	96.23	48.28	0.00	98.61
bzip2	7.79	-6.95	36.38	7.41	1.05	34.52	8.17	0.84	40.17
twolf	-0.22	0.34	4.73	-0.11	-1.46	5.30	-0.33	-1.34	6.58
apsi	40.91	-14.29	64.45	40.91	0.00	64.50	40.91	0.00	65.28
sphinx	15.66	1.47	14.65	17.31	-3.91	14.99	17.03	-3.84	16.77
average	28.37	-9.39	48.62	26.80	0.64	50.05	29.34	0.55	54.34

Table 5.3. Coverage and accuracy of the other push schemes

the four schemes does not reflect that there is large variance on some benchmarks. For example, the miss reduction is 82% for *mgrid* with GRP/LRU plus Push/MRU compared to only 45% with GRP/MRU plus Push/LRU. In general, varied placement policies have more impact on performance and coverage for specific benchmarks than for the overall average.

The push scheme cache does not hide the latencies of those L1 misses that hit the L2 cache. Those misses are mostly L1 capacity misses that are contained by the larger L2 cache. They do not trigger L2 prefetching and thus do not bring in pushed lines, which could hide their latencies. A prefetching engine that triggers prefetches upon L1 misses will solve this problem; we leave this option to future work.

5.2 Combination of Evict-me and Hardware Prefetching

Prefetched cache blocks may pollute the cache if the blocks are useless. Several techniques seek to reduce cache pollution. For instance, hardware can detect stride accesses

and selectively prefetch blocks that are expected to be useful [84]. Compilers can help reduce the impact of cache pollution introduced by hardware prefetching in two ways. First, the cache pollution of a prefetched cache block may be harmless if it evicts a block that is marked as evict-me. In this situation, the marked line is probably useless anyway. Second, compilers can use locality analysis to decide when prefetching is necessary. We examined the second option in Chapter 4, where we generate compiler hints to guide an aggressive hardware prefetcher. In this section, we explore the first option. GRP enhances an aggressive hardware prefetcher and tolerates L2 miss latencies. In Section 5.1, we described a push scheme that hides L1 miss latencies. The evict-me cache replacement policy helps reduce cache misses and can interact with the prefetching and pushing techniques. This section provides results of combining GRP, evict-me, and data pushing.

5.2.1 Performance

For our experiments, we use the same system configurations as in Chapter 4 except that we change the L1 cache line size to 32 bytes and make it 4-way set-associative. The Level 1 cache size does not change. We change the L1 cache line size so that we can examine the performance impact when the cache line sizes of the two levels of caches differ. We merge the benchmarks used in Chapter 3 with the five Fortran benchmarks we chose in Chapter 4. We do not include C benchmarks from SPEC CPU2000 because our compiler currently does not support dependence testing in C code very well.

Figure 5.1 compares the performance of GRP, evict-me (EM), and their combination. Evict-me is turned on for both levels of cache. GRP and Push use LRU and MRU placement policies respectively. Evict-me does not offer much performance improvement over the base, although there are no degradations. It improves *mgrid* and *arc2d* by about 4% and boosts overall performance by 1.5% on average across all benchmarks. GRP, which tolerates most L2 misses for these Fortran benchmarks, improves performance by 30.5% on average. Combining GRP and Evict-me adds an additional 1.7% and the Push scheme adds

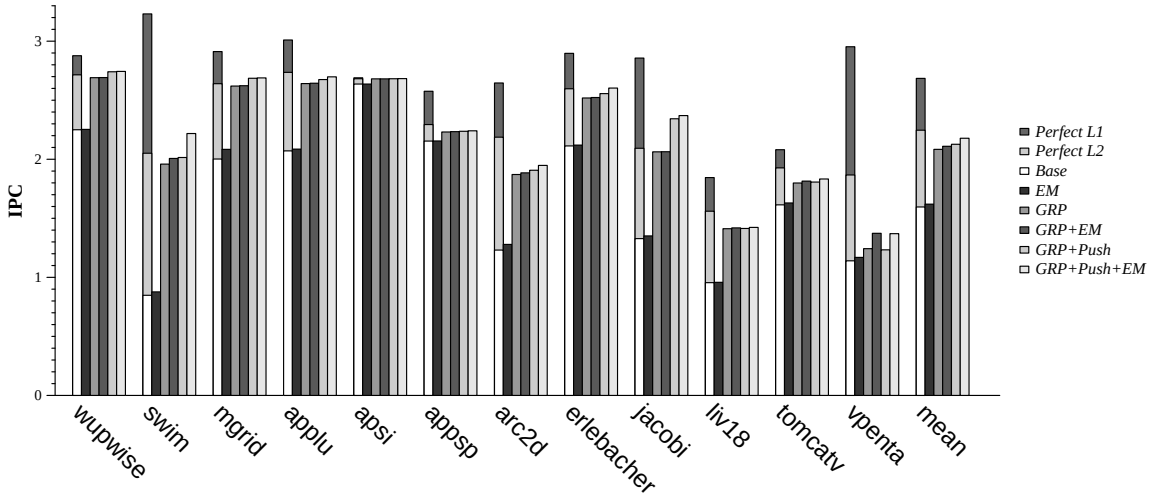

Figure 5.1. Evict-me and GRP

an extra 2.7%. The combination of GRP, Push, and evict-me boosts the base performance by 36.5% on average, adding an additional 6% over GRP. The performance gain mostly arises from *swim*, *jacobi*, and *vpenta*, with improvements over GRP by 13.2%, 14.9%, and 10.2%, respectively. For all but *vpenta*, combining the three techniques beats any single one or two combined. For *vpenta*, GRP/Push/EM is negligibly worse than GRP/EM because of the slight degradation of GRP/Push.

5.2.2 Cache Pollution

We use a pseudo direct-mapped structure to measure the pollution caused by L1 pushes. The structure uses the same line size as the L1 cache and its total number of lines equals the number of sets of the L1 data cache. When a pushed line evicts a cache line, we record the evicted line's address in the pseudo structure. On a demand L1 miss, we check if it hits the structure. If so, we consider the previously pushed line as having polluted the cache. Figure 5.2 shows the normalized L1 pollution caused by pushed lines of GRP/Push and GRP/Push/EM. Above each set of bars is the pseudo structure hit rate with GRP/Push, which we use as a metric of cache pollution. The overall pollution caused by pushed lines

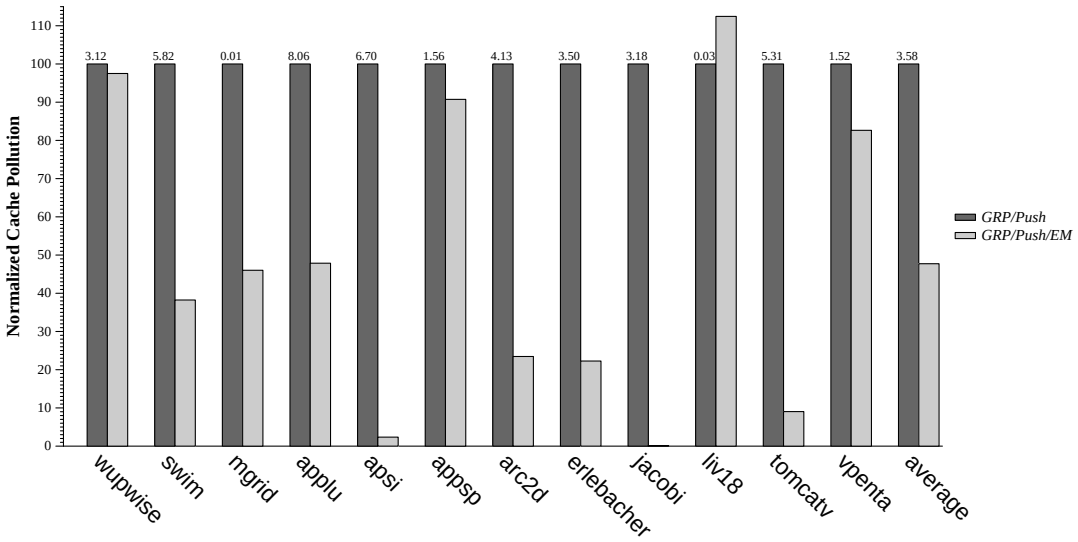

Figure 5.2. Cache pollution

is small. Evict-me reduces this pollution by over a half. For three benchmarks, *apsi*, *jacobi*, and *tomcatv*, evict-me eliminates almost all pollution.

5.2.3 Discussion

The evict-me cache shows less performance improvement here than we reported in Chapter 3. We attribute this to three reasons. First, the ISAs are different. In Chapter 3, we targeted the SPARC V8 ISA, while here we use the Alpha. Different back-end optimizations have an impact on data layout and data access patterns, which affect cache replacement. Second, we use two different simulators, URSIM and SimpleScalar. URSIM is designed to model multi-processor systems. It implements strict cache inclusion while SimpleScalar does not. This choice has a significant impact on cache replacement decisions. In an inclusion system, an L2 replacement will invalidate the corresponding L1 cache lines. On an L1 miss, an invalid line, if it exists, will be evicted before an evict-me line or an LRU line. Third, cache ports are not modeled in SimpleScalar, which makes the caches in SimpleScarlar very aggressive and assumes infinite parallelism in cache accesses. It will be interesting to see how evict-me performs in SimpleScalar when we implement a

stricter cache model. We envision the stricter model will make cache performance more critical and create more improvement space for GRP and Evict-me.

5.3 Chapter Summary

In this chapter, we evaluate the synergy among GRP, data pushing, and evict-me. We show that both the push scheme and evict-me bring an additional performance improvement over GRP. The three techniques together add an additional 6% over GRP. The evict-me cache does not perform as well in SimpleScalar as in URSIM. We attribute this to differences in the compiler, in the accuracy of the L1 cache models, and that cache inclusion is not enforced in SimpleScalar. However, it is worth further investigation to examine where exactly the gap arises. Even in the aggressive cache model of SimpleScalar, we observe that evict-me is very effective at reducing cache pollution caused by prefetched or pushed lines. It eliminates half the L1 cache pollution from the data push scheme. This result suggests our cooperative techniques have potential to interact well to improve memory system performance.

CHAPTER 6

CONCLUSION

The memory system continues to be a major bottleneck on modern architectures. This dissertation proposes a hardware/software cooperative approach to address this challenge. We show the promise of our approach by examining techniques to improve cache replacement and data prefetching and by demonstrating their effectiveness. This chapter summarizes our contributions and discusses future work.

6.1 Contributions

We propose a unique hardware/software cooperative approach that combines the strengths of both software and hardware. Compiler analyses can detect program characteristics such as data locality, data access patterns, loop structures, and the call graph. These static features provide a global view of a program, and, if communicated to the hardware, the hardware can use them to direct run-time decisions. On the other hand, the hardware at run time is able to supply a precise execution history and perfect run-time state such as variable values, cache hits or misses, and loop bounds. However, the scope of the run-time information is limited. The cooperative techniques discussed in this dissertation benefit from static compiler hints as well as run time status. Our approach relies on ISA extensions to convey compiler hints to the hardware by encoding the hints in load/store instructions. This new interface ensures that the hints can interact with each memory access at run time and help hardware make decisions when needed.

We apply the hardware/software cooperative approach to cache replacement. We enhance a primary cache replacement policy, LRU, using a one-bit (evict-me) compiler hint.

Each cache line is augmented by a single evict-me bit. Cache replacement chooses to evict a marked line if any and follows the LRU policy if no line is marked. This architecture extension is unique and practical. We introduce a new notation, reuse level, to formulate our compiler model and prove that our cache replacement strategy will at least match LRU in hits if the compiler can make correct predictions. Our volume-based compiler analysis uses the total data volume across adjacent loop nests to estimate reuse distance and reuse levels. Our simulated results validate this heuristic. By applying evict-me in the L1 and L2 caches, we observe up to 56% miss reduction, and a corresponding 34% performance improvement. Combined with data prefetching techniques, the evict-me cache reduces cache pollution by half.

Our cooperative approach leads to a new data prefetching paradigm: compiler-guided prefetching. Using software hints to direct hardware prefetchers, this new paradigm enjoys the high accuracy of software prefetchers as well the high performance of hardware prefetchers. Specifically, we propose a new prefetching technique called guided region prefetching. Guided region prefetching enhances hardware-only scheduled region prefetching by using compiler assistance to decide when to prefetch and how big the prefetching region size should be. We propose a set of compiler analyses to generate *spatial* hints and *size* hints. Guided by these two types of hints, GRP is able to deliver performance close to SRP but reduce additional bus traffic from 180% to only 23%. Both GRP and SRP perform significantly better than a state-of-the-art stride prefetcher. The stride prefetcher uses several hardware features to control accuracy, which makes the traffic even lower than GRP. However, GRP has a simpler hardware implementation and higher performance than the stride prefetcher.

We propose a pointer prefetching scheme, which is essentially the same as and developed independently of a published prefetching technique, called *content-aware prefetching* [32]. Our content-aware prefetcher shows over a 10% performance improvement in 3 of 17 selected benchmarks. It, in fact, exploits spatial locality among two of the three

benchmarks. We find that region prefetching mostly outperforms or subsumes pointer prefetching. It will be interesting to investigate more benchmarks to see if this conclusion holds more widely.

We propose a data push scheme that pushes prefetched data at the L2 cache to the L1 cache. This scheme is independent of any specific L2 prefetcher. Considering the cache pollution and cache replacement pressure introduced by the pushed lines, this scheme works better with guided region prefetching, which reduces useless L2 prefetches. Improving over GRP, the push scheme delivers performance improvement of around 10% for two benchmarks, and 2.5% on average by hiding L1 latencies. Varying the placement policy for the prefetched and pushed lines has a small overall impact across all benchmarks. However, we see larger gaps for several programs.

Combining evict-me, GRP, and data pushing brings an additional 6.5% improvement over GRP. Evict-me helps reduce by half the L1 cache pollution due to pushed lines. The push scheme is built upon the highly accurate GRP, which makes L1 pushing accuracy high and cache pollution low. Our experiments on this combination use only Fortran benchmarks, which typically show good spatial locality and yield high prefetching accuracy even using SRP. It will be interesting to see if the combination can bring us more synergy for C benchmarks.

6.2 Future Work

Our future interests include applying the cooperative approach to other areas and addressing some concerns of our current focuses, cache replacement and prefetching.

The evict-me cache uses one bit to denote preferred eviction of a cache line. Contrary to the semantics of an evict-me bit would be a *save-me* bit, which denotes a preference to retain a line until a hit to it. We can apply similar locality analyses to generate save-me bits. The combination of evict-me bits and save-me bits can probably work well together. Given a program and an input, varying the cache size could mean an evict-me hint should

be changed to a save-me hint or vice versa. Given a loop nest, when the total data volume is smaller than the cache size, we would want to save every reference with inter-nest reuses. When the data volume is much greater than the cache size, we want to evict data without temporal locality in the current nest whether it has inter-nest reuse or not. But if the cache size is not too big or too small, we probably want to save part of the data for reuse across nests and evict other data to exploit locality in the current nest. In this case, we need two bits.

Our evict-me analyses now only work for Fortran code. We plan to investigate the possibility of applying evict-me to C and Java code. Current compiler support for the evict-me cache involves data locality analysis and data volume estimation in loop nests. Locality analysis in C code is much harder due to aliases and the pointer-like internal representation of C arrays. Aliases create uncertainties about temporal reuses of arrays involving aliasing. A conservative strategy assumes temporal uses among all references in an alias group. We are interested in the impact of the accuracy of alias analysis on marking evict-me loads. Arrays in C code are frequently represented in pointer form. For example, $*(p+i)$ can be treated as an array reference when p is a pointer and i is an induction variable. We want to extend our compiler infrastructure to detect these implicit arrays and feed them to dependence testing.

Although region prefetching improves performance by more than 20% on average across the SPEC CPU2000 benchmarks, there are still 7 remaining benchmarks where the performance gap from a perfect L2 is greater than 15%. We are interested in investigating these benchmarks further. A typical problem in these benchmarks is linked data structures traversed in short loops. Unfortunately, we find that content-aware prefetching does not help these benchmarks much. Content-aware prefetching does not perform as well as region prefetching and is typically subsumed by it. We are interested in examining more benchmarks to see if content-aware pointer prefetching is always subsumed by region prefetching. Content-aware prefetching detects the pointer-like values in a fetched cache line.

This kind of prefetch is usually too late for the pointers in a short loop. It would be interesting to examine the interaction between region prefetching and software linked structure prefetching techniques. In particular, we plan to examine how jump pointer prefetching interacts with region prefetching.

The application of the hardware/software cooperative approach is not limited to cache replacement and prefetching. We can use compiler hints to guide data placement and movement in a partitioned cache or a NUCA (non-uniform cache access) cache [60]. We also see the applicability of this approach to cache coherence, memory disambiguation, and I/O controls. In a partitioned cache, the compiler can tell which two arrays in a nest tend to conflict with each other and thus should be mapped to difference partitions. In a NUCA cache, compiler analysis can determine which cache line should be promoted to a bank closer to the processor. Compiler dependence testing and alias analysis can specify when there is no dependence between a store and a load. This information can be used to speed up speculative execution.

6.3 Concluding Remarks

We have demonstrated that a hardware/software cooperative approach can bridge much of the processor-memory performance gap. We propose two unique techniques using this approach and show that they improve cache replacement and prefetching. We also show that the two techniques work well independently and together. We present practical and simple hardware designs, compiler algorithms, and compiler implementations. Our approach is thus feasible to include in future computer systems.

BIBLIOGRAPHY

[1] W. A. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[2] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 169–178, San Diego, CA, May 1993.

[3] V. Agarwal, M.S. Hrishikesh, S. W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, Vancouver, Canada, June 2000.

[4] T. Alexander and G. Kedem. Distributed predictive cache design for high performance memory system. In *Second International Symposium on High Performance Computer Architecture*, pages 254–263, February 1996.

[5] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 52–61, 2001.

[6] Architecture and Language Implementation Group, University of Massachusetts, Amherst. Scale compiler infrastructure. In *http://ali-www.cs.umass.edu/Scale*.

[7] R. Ashok, S. Chheda, and C. A. Moritz. Cool-mem: Combining statically speculative memory accessing with selective address translation for energy efficiency. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–143, San Jose, CA, October 2002.

[8] T. Austin and D. Burger. Micro-30 SimpleScalar tutorial. Technical report, http://www.cs.wisc.edu/ mscalar/ss/tutorial.html, 1997.

[9] 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, Albuquerque, NM, November 1991.

[10] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):79–101, 1966.

[11] D. Bernstein, D. Cohen, A. Freund, and D. E. Maydan. Compiler techniques for data prefetching on the PowerPC. In *The 1996 International Conference on Parallel Architectures and Compilation Techniques*, pages 19–26, Limassos, Cyprus, June 1995.

[12] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 244–257, Cambridge, MA, October 1996.

[13] D. Burger. *Hardware Techniques to Improve the Performance of the Processor/Memory Interface*. PhD thesis, Dept. of Computer Science, University of Wisconsin at Madison, 1998.

[14] D. Burger, A. Kägi, and J. R. Goodman. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 78–89, Philadelphia, PA, May 1996.

[15] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[16] B. Cahoon and K. S. McKinley. Simple and effective array prefetching for Java. In *ACM Java Grande*, pages 86–95, Seattle, WA, November 2002.

[17] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compiler Techniques*, pages 280–291, Barcelona, Spain, Sept. 2001.

[18] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–150, San Jose, CA, October 1998.

[19] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991.

[20] S. Carr, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, CA, October 1994.

[21] M. Charney and A. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE_CEG_95-1, Cornell University, February 1995.

[22] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behaviour of nested loops. In *Proceedings of the SIGPLAN 2001 Conference*

on Programming Language Design and Implementation, pages 286–297, Snowbird, Utah, June 2001.

[23] T. Chen and J. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Boston, MA, October 1992.

[24] T. Chen and J. Baer. Effective hardware based data prefetching. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[25] T-F Chen. An effective programmable prefetch engine for high-performance processors. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 237–242, Ann Arbor, Michigan, November 1995.

[26] Tien-Fu Chen. *Data Prefetching for High Performance Processors*. PhD thesis, University of Washington, Department of Computer Science and Engineering, July 1993.

[27] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 13–24, Atlanta, GA, May 1999.

[28] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 1–12, Atlanta, GA, May 1999.

[29] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *The International Symposium on Memory Management*, pages 37–48, Vancouver, BC, October 1998.

[30] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 199–209, Berlin, Germany, June 2002.

[31] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 14–25, June 2001.

[32] Robert Cooksey, Stephen Jordan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the Tenth Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–290, San Jose, CA, October 2002.

[33] Vinodh Cuppu and Bruce Jacob. Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor DRAM-system performance. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 62–73, Goteborg, Swenden, June 30 - July 4 2001.

[34] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. A performance comparison of contemporary DRAM architectures. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 222–233, Atlanta, GA, May 2 - 4 1999.

[35] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared-memory multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages 56–63, St Charles, IL, 1993.

[36] F. Dahlgren and P. Stenstrom. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *First International Symposium on High Performance Computer Architecture*, pages 68–77, Raleigh, NC, January 1995.

[37] B. Dileep. Parallelism in mainstream enterprise platforms of the future. In *Keynote, the 2002 International Conference on Parallel Architecture and Compilation Techniques*, Charlottesville, Virginia, September 2002.

[38] C. Dulong. The IA-64 architecture at work. *IEEE Computer*, 31(7):24–32, July 1998.

[39] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, October 1998.

[40] J. D. Gindele. Buffer block prefetching method. *IBm Tech. Disclosure Bull.*, 20(2):696–697, July 1977.

[41] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 15–29, Toronto, Canada, June 1991.

[42] E. H. Gornish and A. V. Veidenbaum. An integrated hardware/software scheme for shared-memory multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages 281–284, St Charles, IL, 1994.

[43] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 107–116, Vancouver, Canada, June 2000.

[44] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[45] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1995.

[46] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, Computer Science Dept., University of California, Berkeley, 1987. Available as Technical Report UCB/CSD 87/381.

[47] M. D. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, December 1988.

[48] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[49] Glenn Hinton, Dave Sager, Mike Upton, Darren Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, (Q1 2001), 2001.

[50] Christopher J. Hughes and Sarita Adve. Memory-side prefetching for linked data structures. Technical Report UIUCDCS-R-2001-2221, University of Illinios, Urbana Champagne, May 2001.

[51] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the design space of future CMPs. In *Proceedings of the 2001 International Conference on Parallel A rchitectures and Compilation Techniques*, pages 199–210, Sep 2001.

[52] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-time spatial locality detection and optimization. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 57–64, Research Triangle Park, NC, December 1997.

[53] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, 1997.

[54] 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 International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, June 1990.

[55] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *The 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 306–313, Paris, France, October 1998.

[56] M. Karlsson, F. Dahlgren, and P. Sternstrom. A prefetching technique for irregular accesses to linked data structures. In *Sixth International Symposium on High Performance Computer Architecture*, pages 206–217, Toulouse, France, January 2000.

[57] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[58] R. E. Kessler, E.J. McLellan, and D.A. Webb. The Alpha 21264 microprocessor architecture. Technical report, http://www.compaq.com/AlphaServer/download/ev6chip.pdf, November 1999.

[59] Richard E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, Mar/Apr 1999.

[60] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the Tenth Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, San Jose, CA, October 2002.

[61] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 159–170, San Jose, CA, October 2002.

[62] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–53, Toronto, Canada, May 1991.

[63] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the Eighth International Symposium on Computer Architecture*, pages 81–87, May 1981.

[64] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, pages 207–218, Williamsburg, VA, January 1981.

[65] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 357–368, Barcelona, Spain, June 1998.

[66] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, July 2001.

[67] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, pages 15–26, October 1994.

[68] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the SPHINX speech recognition system. In *IEEE Transactions on Acoustics, Speech and Signal Precessing*, volume 38(1), pages 35–44, 1990.

[69] Wei-Fen Lin, Steven K. Reinhardt, and Doug Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 301–312, Jan 2001.

[70] G. Lindenmaier, K. S. McKinley, and O. Temam. Load scheduling with profile information. In A. Bode, T. Ludwig, and R. Wismüller, editors, *Euro-Par 2000 – Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 223–233, Munich, August 2000. Springer-Verlag.

[71] 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 IEEE/ACM International Symposium on Microachitecture*, pages 231–236, November 1995.

[72] C. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, MA, October 1996.

[73] C. Luk and T. C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 182–194, Dallas, TX, December 1998.

[74] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 40 – 51, Goteborg, Swenden, June 30 - July 4 2001.

[75] N. McIntosh. *Compiler Support for Software Prefetching*. PhD thesis, Rice University, May 1998.

[76] S. A. McKee, A. Aluwihare, B. H. Clark, R. H. Klenke, T. C. Landon, C. W. Oliver, M. H. Salinas, A. E. Szymkowiak, K. L. Wright, W. A. Wulf, and J. H. Aylor. Design and evaluation of dynamic access ordering hardware. In *Proceedings of the 1996 ACM International Conference on Supercomputing*, pages 125–132, Philadelphia, PA, May 1996.

[77] S. A. McKee, R. H. Klenke, K. L. Wright, W. A. Wulf, M. H. Salinas, J. H. Aylor, and A. P. Batson. Smarter memory: Improving bandwidth for streamed references. *IEEE Computer*, 31(7):54–63, July 1998.

[78] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[79] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, November 1999.

[80] C. A. Moritz, M. Frank, and S. Amarasinghe. Flexcache: A framework for compiler generated data caching. In *Lecture Notes in Computer Science*. Springer Verlag, 2001.

[81] C. A. Moritz, M. Frank, W. Lee, and S. Amarasinghe. Hot pages: Software caching for raw microprocessors. Technical Report LCS-TM-599, Laboratory for Computer Science, MIT, August 1999.

[82] T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, October 1992.

[83] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM reference manual (version 1.0). Technical Report Technical Report 9705, Rice University, Dept. of Electrical and Computer Engineering, August 1997.

[84] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21th International Symposium on Computer Architecture*, pages 24–33, Chicago, IL, April 1994.

[85] M. Prvulovi, D. Marinov, Z. Dimitrijevic, and V. Milutinovic. The split spatial/non-spatial cache: A survey and reevaluation of performance. *IEEE TCCA Newsletters*, pages 8–17, 1999.

[86] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[87] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 449–456, Melbourne, Australia, July 1998.

[88] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 111–121, Atlanta, GA, May 1999.

[89] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Proceeding of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, 1998.

[90] Vatsa Santhanam, Edward H. Gronish, and Wi-Chung Hsu. Data prefetching on the HP PA-8000. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 264–273, Denver, CO, June 1997.

[91] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 155–164, Rhodes, Greece, June 1999.

[92] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 42–53, Monterey, California, December 2000.

[93] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Barcelona, Spain, Sept. 2001.

[94] J. P. Singh, W. Weber, and A. Gupta. Splash: Standford parallel applications for shared-memory. Technical Report TR CSL-TR-91-469, Computer Systems Laboratory, Stanford University, April 1991.

[95] J. Skeppstedt and M. Dubois. Hybrid compiler/hardware prefetching for multiprocessors using low-overhead cache miss traps. In *Proceedings of the 1997 International Conference on Parallel Processing*, pages 298–307, Bloomington, IL, August 1997.

[96] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: Simple and effective adaptive page replacement. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 122–133, Atlanta, GA, May 1999.

[97] A. J. Smith. Sequential program prefetching in memeory hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[98] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[99] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 171–182, May 2002.

[100] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. *Journal of Instruction Level Parallelism*, 1:1–24, 1999.

[101] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the Twenty-third Annual ACM Symposium on the Principles of Programming Languages*, pages 21–24, St. Petersburg, FL, January 1996.

[102] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 24–35, Santa Clara, CA, May 1993.

[103] O. Temam. An algorithm for optimally exploiting spatial and temporal locality in upper memory levels. *IEEE Transactions on Computers*, 48(2):150–158, February 1999.

[104] Univerisity of Maryland. *The Omega Library*, 1996. http://www.cs.umd.edu/projects/omega/.

[105] O. S. Unsal, R. Ashok, I. Koren, C. M. Krishna, and C. A. Moritz. Cool-cache for hot multimedia. In *Proceedings of the 34rd International Symposium on Microarchitecture*, pages 274–283, Austin, TX, December 2001.

[106] O. S. Unsal, R. Ashok, I. Koren, C. M. Krishna, and C. A. Moritz. Cool-cache: A compiler-enabled energy efficient data caching framework for embedded and multimedia systems. *ACM Transactions on Embedded Computing Systems, Special Issue on Low Power*, 2(3):373–392, August 2003.

[107] O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Moritz. Cool-fetch: Compiler-enabled power-aware fetch throttling. In *ACM Computer Architecture Letters*, pages 100–103, May 2002.

[108] O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Moritz. The minimax cache: An energy-efficient framework for media processors. In *Seventh International Symposium on High Performance Computer Architecture*, pages 131–140, Boston, MA, February 2002.

[109] S. P. VanderWiel and D. J. Lilja. A compiler-assisted data prefetch controller. In *Proceedings of International Conference on Computer Design*, pages 372–377, Austin, TX, 1999.

[110] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanovic. Energy efficient architectures: Direct addressed caches for reduced power consumption. In *Proceedings of the 34rd International Symposium on Microarchitecture*, pages 124–133, Austin, TX, December 2001.

[111] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[112] W. A. Wong and J. Baer. Modified LRU policies for improving second-level cache bahavior. In *Sixth International Symposium on High Performance Computer Architecture*, pages 49–60, Toulouse, France, January 2000.

[113] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 210–221, Berlin, Germany, June 2002.

[114] C.-L. Yang and A. R. Lebeck. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 176–186, May 2000.

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

[116] Lixin Zhang. URSIM reference manual. Technical Report UUCS-00-015, University of Utah, August 2000. http://www.cs.utah.edu/projects/impulse.

[117] Z. Zhang and T. Torrellas. Speeding up irregular applications in shared memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd*

International Symposium on Computer Architecture, pages 1–19, Santa Margherita Ligure, Italy, June 1995.