RUN-TIME ADAPTIVE CACHE MANAGEMENT

BY

TERESA LOUISE JOHNSON

B.S., University of Illinois, 1993
M.S., University of Illinois, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

RUN-TIME ADAPTIVE CACHE MANAGEMENT

Teresa Louise Johnson, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1998
Wen-mei W. Hwu, Advisor

The growing disparity between processor and memory performance has made cache misses increasingly expensive. Additionally, data and instruction caches are not always used efficiently, resulting in large numbers of cache misses. Therefore, the importance of cache performance improvements at each level of the memory hierarchy will continue to grow. For numeric programs there are several known compiler techniques for optimizing data cache performance. However, integer (non-numeric) programs often have irregular access patterns that are more difficult for the compiler to optimize. In the past, cache management techniques such as cache bypassing were implemented manually at the machine-language-programming level. As the available chip area grows, it makes sense to spend more resources to allow intelligent control over the cache management.

The objective of this dissertation is to improve cache effectiveness, taking advantage of the growing chip area, utilizing run-time adaptive cache management techniques, and optimizing both performance and cost of implementation. Specifically, the aim is to increase cache effectiveness for integer programs. This dissertation proposes a microarchitecture scheme where the hardware determines data placement within the cache hierarchy based on dynamic referencing behavior. This scheme is fully compatible with existing instruction set architectures. This dissertation also examines the theoretical upper bounds on the cache hit ratio that the proposed techniques can provide, for several integer applications. Then, detailed trace-driven simulations of several integer applications are used to show that the implementations described in this dissertation can achieve performance close to that of the upper bound.

DEDICATION

To my family.

# ACKNOWLEDGMENTS

First, I would like to thank my advisor, Professor Wen-mei W. Hwu, for his insight, guidance and support during my studies, and for the opportunities he has provided. My future career will benefit greatly from the lessons I have learned.

Next, I would like to extend my gratitude to the other members of my dissertation committee, Dr. Santosh Abraham, Professor Ravishankar Iyer, and Professor William Sanders. Their comments, questions, and suggestions improved the quality of this work immensely. Santosh Abraham in particular has provided helpful guidance and suggestions throughout my graduate tenure. I would also like to thank Professor Pen-Chung Yew, for his guidance during my first year of graduate school.

This research would not have been possible without the support of the members of the IMPACT research group, both past and present. The group members were always willing to provide assistance, including research discussions, practice talks, and software enhancements. John Gyllenhaal provided the simulation framework upon which my research has built, and spent countless hours answering my questions and providing feedback and suggestions on this research. Grant Haab spent a great deal of time introducing me to the Pcode environment, and answering many front-end compilation questions in my early work. Also thanks to Rick Hank, Dan Lavery, Scott Mahlke, David August, and Dan Connors for answering numerous IMPACT compilation questions and providing bug fixes over the years. Dan Connors and Matthew Merten developed the K6 trace simulation support, and provided much help with paper revisions and feedback on this research. Thanks to Brian Deitrich, Ben-Chung Cheng, and John Gyllenhaal for providing help with the Informix simulation project. Many thanks to Qudus Olaniran, John Sias, and Mike Thiems for their hard work on the x86 emulation project. I would also like to thank Rick Hank, David August, John Gyllenhaal, John Sias, and Marie Conte for their extensive workstation and PC administration efforts. Finally, thanks to everyone in the group for creating such an enjoyable work atmosphere, especially to Sabrina Hwu, Jaymie Braun, Dan Connors, and David August for being great friends and providing many good times. Sabrina Hwu has been a wonderful officemate, a supportive friend, and my partner-in-crime in our frequent practical jokes. Thanks to Dan Connors and David August for being such good-natured recipients of those jokes.

Also, thanks to Sandy Armstrong and Samantha Barrow for their many years of steadfast friendship, throughout undergraduate and graduate school, and for providing much needed mental breaks.

I would like to thank my parents, Louise and Glenn, for their love and encouragement throughout my life, especially during graduate school. My grandparents, Grace, Robert, Louise, and Harold, also provided invaluable love and support. I would also like to thank my sister, Susan, for her love and for being such a good friend, and my brother, Steven, for his love and support. Thanks to Bean, for helping me keep my sanity. Finally, I would like to thank Nat, for his love, patience, and support, and for reminding me what is important in life.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

As processor performance improvements outpace that of main memory [1], the cache miss penalty will dominate the cycle counts of many applications. The large improvements in processor performance are due both to better circuit design and fabrication technology, which reduce the cycle time, and to improved *instruction-level parallelism* (ILP) techniques, which increase the instructions executed per cycle. The growing disparity between processor and memory performance will make cache misses increasingly expensive. Not only do the cache misses result in more processor stall cycles, but in processors with dynamic scheduling, they can also disrupt the compiler-generated ILP schedule.

Caches seek to exploit the principle of locality. By storing a referenced item, caches exploit *temporal locality*, the tendency for that item to be rereferenced soon. Additionally, by storing multiple items adjacent to the referenced item, they exploit *spatial locality*, the tendency for neighboring items to be referenced soon. However, conventional caches are not always able to successfully exploit locality available for a particular reference. In this situation a cache miss will occur, resulting in added latency for the requesting load instruction. Cache misses are divided into three categories. *Cold start misses* (also called *compulsory misses*) occur on the first access to a cache block. *Capacity misses* occur when the cache cannot contain all the blocks accessed during program execution. *Conflict misses* occur in a non-fully-associative cache due to too many blocks mapping into a single cache set. In order to eliminate the execution cycles spent stalled on a memory access, it is necessary both to minimize the number of cache misses and to tolerate the memory latency incurred by the remaining misses.

Methods to improve memory access behavior include *static* (compiler-only) methods, *dynamic* (hardware-only) methods, and *hybrid* (both compiler and hardware) methods. For numeric programs, there are several known static compiler techniques for optimizing cache performance and tolerating the remaining latency. However, integer (nonnumeric) programs often have irregular access patterns that are more difficult for the compiler to detect and optimize. For example, the temporal and spatial locality of linked list elements and hash table data are often difficult to determine at compile time. This dissertation focuses on cache performance optimization for integer programs. Memory profiling could be used to guide compiler analysis for integer programs, however, only a limited study has been performed to demonstrate the reliability of memory profiling results over multiple input sets [2], [3]. Additionally, users are often unwilling to perform profiling. Dynamic hardware methods, where caching decisions and latency tolerating prefetches are initiated at run time, offer greater promise for integer programs. However, many of the previously proposed dynamic techniques have again focused on numeric codes. Other techniques have allowed only limited adaptability to run-time memory access patterns. Hybrid approaches would rely on the compiler to detect some static information, supplying hints to the hardware. For example, the HPL PlayDoh Architecture [4] specifies hints that can be added to load and store instructions, indicating where the corresponding data should be cached at run-time. Also, the PA 7200 contains a *spatial locality only* hint that could be specified in load and store instructions, indicating to the hardware that the data should bypass the main cache [5].

As the chip area increases, it makes sense to use dynamic hardware techniques to intelligently manage the cache hierarchy. Branch predictors often occupy a fairly significant portion of the chip area and are critical for reducing the branch misprediction stalls. As branch misprediction stalls are reduced, L1 cache miss stalls will become more significant. Therefore, it makes sense to also use some hardware structures on the chip to predict and reduce cache miss stalls. Additionally, the cache size can be constrained by access time considerations. In this case, adding a structure to manage the cache may supply the hit ratio of a larger, less intelligent cache with the access time of a smaller cache.

In order to increase cache effectiveness for integer programs, this dissertation investigates *run-time adaptive cache hierarchy management*, which seeks to proactively control the movement and placement of data[1] in the hierarchy based on the data usage characteristics. This dissertation presents microarchitecture schemes where the hardware guides data caching based on dynamic referencing behavior. The schemes are fully compatible with existing instruction set architectures (ISA).

Adaptive cache hierarchy management schemes seek to manage the cache in a manner that is sensitive to the usage patterns of the memory locations accessed. Since the number of memory locations is excessively large, the notion of a *macroblock* is introduced. A macroblock is a contiguous block of memory that is sufficiently large enough that the maintenance overhead is reasonable, but sufficiently small enough that the access pattern of the memory addresses within each macroblock is statistically uniform. A hardware mechanism called the *memory address table* (*MAT*) is introduced to maintain and utilize the access patterns of the macroblocks to direct data placement in the cache hierarchy. This extension to the cache microarchitecture

[1] In this dissertation, the term *data* is often used generically to refer to both instructions and data in the memory hierarchy.

significantly improves the overall performance of integer applications. The improvements are due to increased cache hit rates and reduced cache handling latencies. Analysis is presented to determine the theoretical upper bounds on the hit ratio that can be achieved with the optimizations. These upper bounds are independent of the prediction mechanism. Then, the performance of the hardware implementations described in this dissertation is compared to the upper bounds, and the opportunities missed by the implementation are examined.

1.1 Contributions

The four major contributions of this dissertation are described below:

- A new approach to cache management, *run-time adaptive cache hierarchy management*, is examined and developed. Previous approaches to improving cache performance have relied on compiler support, or have enabled only limited adaptability to run-time memory access patterns. The need for adaptive run-time control of cache management guided by memory usage patterns is demonstrated, and a framework for detecting and exploiting the run-time usage patterns is developed. Under this framework, memory is divided into regions called *macroblocks*, which allow cost-effective detection and monitoring of usage patterns via a memory access table.

- An adaptive mechanism for performing cache bypassing based on the detected access patterns is developed. Cache bypassing prevents data with less likelihood of in-cache reuse from displacing data with a greater likelihood of reuse. Both quantitative and analytical evaluations of the performance of cache bypassing using the framework developed in this dissertation are presented.

- An adaptive mechanism for performing spatial locality optimizations is developed. This technique allows the cache block sizes to dynamically adjust to the spatial locality detected for each accessed data item. A *spatial locality detection table* is developed to support the run-time detection of spatial locality. The benefit of this spatial locality optimization is both quantitatively and analytically evaluated.

- A dynamic intracache remapping scheme is developed. This technique allows the *pseudo-associativity* of each cache set to dynamically adjust to the run-time contention at that cache location. Quantitative and analytical evaluations are presented to show that this technique can allow caches with limited set associativities to perform better than caches with higher set associativities.

1.2 Overview

This dissertation is composed of seven chapters. Chapter 2 discusses various approaches to improving cache performance. Then, Chapter 3 describes the experimental framework used to generate the performance results in this dissertation.

Chapter 4 presents a case study of a particular benchmark that illustrates the need for one type of intelligent cache management, cache bypassing. This case study is also used to motivate the particular approach taken in this dissertation, and the cache-bypassing scheme is described in detail. Performance results demonstrate the improvements that can be achieved by cache-bypassing schemes on integer codes. An upper-bound performance analysis is used to compare the performance achieved against a theoretical upper bound.

Chapter 5 examines dynamic spatial locality optimizations. First, the spatial locality characteristics of several benchmarks are examined, and a routine from a particular benchmark is used to motivate the dynamic approach. Then, the spatial locality optimization is described in detail, and performance results are presented. An upper-bound analysis is used to compare the run-time decisions to the upper-bound decisions. The performance achieved by integrating the spatial locality optimization with cache bypassing, and the required hardware overhead, are compared to that of simply doubling the caches.

Chapter 6 describes a dynamic intracache remapping optimization, and performance results are presented. Analytical results demonstrate the upper bound on the performance achievable with greater set associativity. The performance achieved by integrating the remapping optimization with the bypassing and spatial locality optimizations is examined.

Conclusions and directions for future work are presented in Chapter 7.

CHAPTER 2

APPROACHES TO MEMORY ACCESS OPTIMIZATION

Several methods attempt to overlap memory accesses with other computation in the processor in order to hide the memory latency. Write buffers can often successfully hide the latency of write misses by buffering the write data until the bus is idle. Nonblocking caches allow multiple outstanding load misses without stalling the processor in order to overlap load miss latency with other computation that does not consume the result of an outstanding load miss [6], [7]. Prefetching attempts to fetch data from main memory to the cache before it is needed, which also overlaps the load miss latency with other computation. Hardware [8], [9], [10], [11], [12], [13], [14], [15]; software [16], [17], [18], [19], [20], [21]; and hybrid [22] prefetching methods for uniprocessor machines have been proposed. However, many of these methods focus on prefetching regular array accesses within well-structured loops,[1] which are access patterns primarily found in numeric applications.

Prefetching methods geared towards integer codes often focus on compiler-inserted [23], [24] or hardware-generated [13] prefetching of pointer targets. Another approach examined compiler heuristics to preload pointer references by scheduling the load further apart from its uses [25]. Reductions in the miss penalty were demonstrated; however, the comparison was made to code scheduled by the *Gcc* compiler on a single-issue machine. These techniques will need reevaluation on wider-issue machines, where there is less scheduling height available, and should

[1]Here *regular* means arrays indexed by the loop iteration variable or some other induction variable.

be compared to more aggressive compilers as well. Abraham et al. showed that fewer than 10 instructions account for the majority of misses in most of the SPEC89 benchmarks [26]. They suggest selectively scheduling the missing loads, identified through profiling, with the cache miss latency. Correlation-based prefetching attempts to pair references with prefetch addresses at run time, by examining the reference patterns and determining what addresses have temporally followed other addresses in the past [27], [28], [29], [30], [31]. Improvements in the hit ratios and in the coverage of miss requests by prefetches have been demonstrated for integer programs [30], [31].

There is also a great deal of prior work on prefetching in multiprocessor systems, but since their focus is even more on optimizing numeric applications, they will not be reviewed here.

Another approach is to reduce the number of misses. Victim caches attempt to reduce conflict miss effects in caches with low associativity [9]. The assist cache [5], is also a small fully associative cache, however, lines are placed in the assist cache before being placed in the main cache. An improved replacement policy was proposed for set-associative caches, utilizing profile-driven markings on load references, indicating the likelihood of in-cache reuse, to preferentially replace data with low likelihood of reuse [2]. Other methods to reduce conflict misses include pseudo-associativity [32], [33] and cache remapping, which are discussed in Chapter 6 of this dissertation, and cache bypassing [34], [35], [36], [37], [38], [39], which is discussed in Chapter 4 of this dissertation. Cache bypassing can also reduce capacity misses. Thiebaut and Stone showed how the footprints in the cache of data accessed by each process in a system can be used to estimate the reload transient, or the number of misses while each process is reloading in cache [40]. They showed how this transient could be used to select a cache size for a real time system running an interrupt-driven process, such that both running processes could remain cached simultaneously. However, many systems are developed to run a wide variety of processes, and this type of analysis will not be feasible, especially for on-chip caches.

It is useful to exploit data locality in order to improve memory access behavior. For example, exploitation of spatial locality can result in reduced or eliminated cache miss penalties for accesses to nearby data. Traditionally, there have been two main approaches used to exploit spatial locality. The first approach is to use larger cache blocks, which have a natural prefetching effect. However, large cache blocks can result in wasted bus bandwidth and poor cache utilization, due to fragmentation and underutilized cache blocks. Both negative effects occur when data with little spatial locality is cached. The second common approach is to prefetch multiple nearby blocks into the cache [14], [15], [16]. However, prefetching is only beneficial when the prefetched data is accessed in cache, otherwise the prefetched data may displace more useful data from the cache, in addition to wasting bus bandwidth. Similar issues exist with write allocate caches, which, in effect, prefetch the data in the cache block containing the written address. Particularly when using large block sizes and write allocation, the amount of prefetching is fixed. However, the spatial locality, and hence the optimal prefetch amount, varies across and often within programs. As the available chip area increases, it is meaningful to spend more resources to allow intelligent control over latency-hiding techniques, adapting to the variations in spatial locality. Methods for exploiting spatial locality intelligently are discussed in Chapter 5 of this dissertation.

CHAPTER 3

EXPERIMENTAL ENVIRONMENT

This chapter presents the environment used for experimental evaluation of the techniques proposed in this dissertation. The benchmark programs for this dissertation consist of a number of integer applications, that will be discussed below in Section 3.1. The experimental environment also includes compiler support, emulation to verify transformation correctness, and the simulation techniques used to generate experimental results.

3.1 Benchmarks

The benchmarks are summarized in Table 3.1. They include *026.compress*, *072.sc* and *085.cc1* from the SPEC92 benchmark suite using the SPEC reference inputs, and *099.go*, *147.vortex*, *130.li*, *134.perl*, and *124.m88ksim* from the SPEC95 benchmark suite using the training inputs. The next two benchmarks consist of modules from the IMPACT compiler [41]

Table 3.1 Benchmark descriptions.

Benchmark	Description	Instructions
026.compress	Performs adaptive Lempel-Ziv coding (SPEC reference input)	68 496 464
072.sc	Spreadsheet operations (SPEC reference input)	112 288 646
099.go	Artificial intelligence in game playing (training input)	533 320 157
147.vortex	Single-user object-oriented database transaction benchmark (training input)	2 414 315 114
Pcode	IMPACT front end (dependence analysis on *combine.c* file from GNU CC)	332 028 293
lmdes2_customizer	IMPACT machine description optimizer (SuperSPARC description opti)	15 344 404
085.cc1	Performs GNU C compilation (SPEC reference input)	134 618 782
130.li	Lisp interpreter (training input)	158 944 224
134.perl	Perl interpreter (training input)	1 099 810 064
124.m88ksim	Simulator for the 88100 microprocessor (training input)	122 642 228
Word	Microsoft Word Version 7.0a (search and replace)	4 629 027
Excel	Microsoft Excel Version 7.0a (spreadsheet cell updates)	4 931 857
Photo	Adobe PhotoDeluxe Version 1.0 (brightness and contrast adjustment)	7 816 881

that are representative of many real-world integer applications. The *Pcode* benchmark, the front end of IMPACT, is run performing dependence analysis with the internal representation of the `combine.c` file from GNU CC as input. The *lmdes2_customizer* benchmark, a machine description optimizer, is run optimizing the SuperSPARC machine description for efficient use by the IMPACT compiler. These optimizations operate over linked list and complex data structures, and utilize hash tables for efficient access to the information. In addition, hardware tracing facilities on an AMD K6 [42] system were used in conjunction with special hardware provided by AMD, called SpeedTracer, to take traces of instructions and memory addresses for several Windows NT applications, along with their corresponding operating system (OS) activities. The first trace, about 5 million instructions in length, was taken during the execution of Microsoft Excel Version 7.0a, while a large number of spreadsheet cells were being updated with the results of functions applied to those cells at the start of the tracing session. The second trace, over 4.6 million instructions in length, was taken during the execution of Microsoft Word Version 7.0a, while a find and replace operation was performed on a large document. The third trace, over 7.8 million instructions in length, was taken during the execution of Adobe PhotoDeluxe Version 1.0, while an automatic adjustment of the brightness and contrast of a picture was performed.

3.2 Compiler and Architecture

In order to provide a realistic evaluation of the techniques presented in this dissertation, code for all benchmarks (except the Windows NT benchmarks) was first optimized using the IMPACT compiler [41]. Classical optimizations were applied, followed by optimizations which increase instruction level parallelism such as loop unrolling and superblock formation [43].

Table 3.2 Instruction latencies for simulation experiments.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
Memory load	2	FP multiply	2
Memory store	1	FP divide (single prec.)	8
Branch	1 + 1 slot	FP divide (double prec.)	15

The code was scheduled, register allocated, and optimized for an eight-issue, scoreboarded, superscalar processor with register renaming. Each of the eight functional units are pipelined and can execute any type of instruction. The ISA is an extension of the HP PA-RISC instruction set to support compile-time speculation. Up to four memory accesses can be executed per cycle. The register file contains 64 integer registers and 64 double-precision floating-point registers.

3.3 Simulation of SPEC and IMPACT Benchmarks

To verify the correctness of the code transformations, emulation of the target processor architecture was performed for all input programs on a Hewlett-Packard PA-RISC workstation.

The emulator drives the simulator that models on a cycle-by-cycle basis the processor and the memory hierarchy (including all related busses) to determine application execution time, cache performance and bus utilization. The instruction latencies used are those of a Hewlett-Packard PA-RISC 7100 microprocessor, as given in Table 3.2. The base machine configuration is described in Table 3.3.

The memory hierarchy, summarized in Table 3.3, includes separate L1 instruction and data caches. The L1 instruction cache is a direct-mapped, 32-KB split-block cache with a 64-byte block size. A split-block instruction cache ensures that eight instructions can always be returned to the processor on each instruction cache hit. If the processor requests an instruction in the latter 32 bytes of a cache block, some instructions from the first half of the following cache block

Table 3.3 Base machine configuration.

L1 Icache	32-KB split-block, direct mapped, 64-byte block
L1 Dcache	16-KB non-blocking (50 max), direct mapped, 32-byte block, multiported, writeback, no write allocation
L1-to-L2 bus	8-byte bandwidth, split-transaction, 4-cycle latency, returns critical word first
L2 Cache	same as L1 Dcache except: Unified I&D, 256 KB, 64-byte block
System bus	same as L1-to-L2 Bus except: 100-cycle latency
Issue	8-issue uniform, except 4 memory ops/cycle max
Registers	64 integer, 64 double precision floating-point

can also be returned, so that eight sequential instructions are supplied to the processor. The L1 data cache is a direct-mapped,[1] 16-KB nonblocking cache with a 32-byte block size. The data cache is a multiported, write-back, no write-allocate cache that satisfies up to 4 load or store requests per cycle from the processor and has streaming support. Up to 50 load misses can be outstanding simultaneously on the bus connecting the L1 and L2 data caches. An eight-entry write buffer combines write requests to the same cache block. The instruction cache and data cache share a common, split-transaction L1-L2 bus, with a four-cycle latency and 8 bytes/cycle data bandwidth. The memory hierarchy also includes a direct-mapped, 256-KB nonblocking uniform (instruction and data) L2 cache with a 64-byte block size. This cache is also write-back and no write allocate. Up to 50 load misses can be outstanding simultaneously from the L2 data cache on the system bus, which is split-transaction with a 100 cycle latency to memory and 8 bytes/cycle data bandwidth.

A direct-mapped branch target buffer with 1024 entries is used to perform dynamic branch prediction using a 2-bit counter. Hardware speculation is supported, and the branch misprediction penalty is approximately two cycles.

[1] Initially both data caches are direct-mapped, but later sections examine the effects of set-associative data caches.

Since simulating the entire applications at this level of detail would be impractical, uniform sampling is used to reduce simulation time [44], [45]; however, emulation is still performed between samples in order to update the CPU state so that accurate trace information can be generated during later samples. The simulated samples are 500 000 instructions in length and are spaced evenly across program execution, yielding at least a 1% sampling ratio. For smaller applications, the time between samples is reduced to maintain at least 50 samples (25 000 000 instructions), except in the case of *lmdes2_customizer*, where the total instruction count is around 15 000 000 and the entire application is simulated. To evaluate the accuracy of this technique, several configurations were simulated both with and without sampling, and the performance improvements were found to be very similar. For each optimization discussed in Chapters 4-6, the performance improvements obtained with and without sampling for one optimized configuration will be compared across the benchmarks.

3.4 Simulation of Windows NT Benchmarks

Because hardware traces were taken, no compilation was performed for the Windows NT benchmarks. Also, no sampling was performed during simulation of these benchmarks.

CHAPTER 4

CACHE BYPASSING

As discussed in Chapter 1, data caches are not always used efficiently. In numeric programs there are several known compiler techniques for optimizing data cache performance. However, integer programs often have irregular access patterns that are more difficult for the compiler to analyze and optimize. This chapter presents a microarchitecture scheme where the hardware dynamically determines data placement based on dynamic referencing behavior, in a manner that is sensitive to the usage patterns of the memory locations accessed. In particular, intelligent cache bypassing techniques are examined.

Some modern processors contain mechanisms to disable caching. For example, the Pentium Pro utilizes a *page-level cache disable* (PCD) bit in each page directory and page table entry, which controls the cacheability of data in the corresponding page [46]. Currently, the PCD bit is used by the OS to disable caching for memory-mapped input-output (I/O) ports. It could also be set manually by low-level programmers to disable caching of other data.

Other methods for both static and dynamic cache bypassing have been investigated in the past. These methods attempt to bypass data with little likelihood of in-cache reuse, and base the bypassing decisions either on the particular load referencing that data [34], [36], [37], or on the address being accessed [38].

Chi and Dietz proposed unified management of registers and cache memory [34], that uses cache analysis to determine when data allocated to a register can safely bypass the cache. They used register liveness and data dependence analysis to determine which data is unambiguously aliased, and then mark the corresponding loads for bypass. In nonnumeric code there will be less unambiguously aliased data and, therefore, less opportunities for this optimization. Also, the small line size used meant that the baseline cache did not exploit spatial locality, and, therefore, aggressive bypassing was not as likely to degrade performance. With the current standard block sizes of around 32 bytes, spatial locality will be more of an issue.

McFarling proposed a dynamic exclusion scheme for instruction caches [35]. A finite state machine was used to detect common instruction execution patterns, typically found in loops, and dynamically bypass some instructions from the cache to reduce conflicts. This scheme performed well for instruction caches, but not for data caches.

González et al. proposed using a *locality prediction table* to detect both spatial and temporal locality in numeric codes [37]. The table detects array lengths and strides, and if the array accesses are found to have large strides and to result in self-interference the data is bypassed.

Tyson et al. proposed a method where loads are marked for cache bypass either statically by the compiler or dynamically at run time [36]. While this dissertation also investigates cache bypassing, the work differs in several key aspects. First, Tyson et al. use miss behavior of the load as the main decision metric, and reuse behavior as a secondary metric, for determining whether to bypass that load's data. This dissertation focuses on the reuse behavior, because data that tend to miss may still have high locality that would result in reuse while in the cache. Second, they decide whether to bypass data based on the particular load referencing that data. As will be shown in Section 4.1.1, a single load instruction may reference data with widely varying access patterns. Therefore, the mechanism described in this dissertation is designed to determine whether to bypass based on the data address. As a result, this dissertation achieves an increase in cache hit ratios, a decrease in the bus traffic, and a decrease in total cycle counts, while they achieve a decrease in the bus traffic at the expense of a small drop in the cache hit ratios, which may result in performance degradation.

Rivers and Davidson [38] presented a hardware bypassing mechanism based on reuse behavior of cached locations that was proven effective for numeric programs. Their mechanism performs bypassing of the first-level data cache only, whereas this dissertation studies bypassing of both cache levels. It is important to improve second-level cache performance, since system bus latencies are large. Also, their scheme marks a cache block for bypass until the corresponding block is replaced from the second-level cache, at which time all reuse information is lost. This dissertation overcomes this limitation by monitoring reuse behavior in a separate structure, and by allowing bypass decisions to vary based on dynamic accessing behavior.

Some of the techniques described in this dissertation, in particular the bypass buffer presented in Section 4.2.2, could be used in synergy with many of these schemes to obtain improved performance.

4.1 Concepts

4.1.1 Case study

To understand some of the inefficiencies of conventional cache hierarchies, it is helpful to first examine the accessing behavior of a particular application in detail. Figure 4.1 shows the main loop body of the *026.compress* program from the SPEC92 benchmark suite [47]. Over 90% of the execution time for *026.compress* is in this loop body. Many of the memory accesses in *compress* are to its hash tables, `htab` and `codetab` (the lines containing the hash table load

```
    while ( (c = getchar()) != EOF ) {
        in_count++;
        fcode = (long) (((long) c << maxbits) + ent);
        i = ((c << hshift) ^ ent);
1.      if ( htabof (i) == fcode ) {
2.          ent = codetabof (i);
            continue;
        } else if ( (long)htabof (i) < 0 ) goto nomatch;
        disp = hsize_reg - i;
        if ( i == 0 ) disp = 1;
probe:
        if ( (i -= disp) < 0 ) i += hsize_reg;
3.      if ( htabof (i) == fcode ) {
4.          ent = codetabof (i);
            continue;
        }
        if ( (long)htabof (i) > 0 ) goto probe;
nomatch:
        output ( (code_int) ent );
        out_count++;
        ent = c;
        if ( free_ent < maxmaxcode ) {
            codetabof (i) = free_ent++;
                                /* code -> hashtable */
            htabof (i) = fcode;
        }
        else if ( (count_int)in_count >= checkpoint
                                && block_compress )
            cl_block ();
    }
```

Figure 4.1 Compress main loop code.

accesses are numbered[1]). Due to the large hash table sizes (`htab` and `codetab` are roughly 270 KB and 135 KB, respectively) and the fact that the hash table accesses have little temporal or spatial locality, there is very little reuse in a first-level data cache.

Table 4.1 shows the hash table loads' dynamic execution counts, miss ratios and reuse ratios obtained via memory access profiling. A simple cache simulation was performed to determine whether each of the accesses was a first-level cache hit or miss in a direct-mapped 16-KB cache

[1]The other load accesses to `htab` in this loop can be eliminated through load elimination optimizations.

Table 4.1 Profiling statistics for hash table load accesses (direct-mapped, 16-KB data cache with 32-byte lines, single-issue processor). Two-way and four-way cache profiles exhibit similar behavior.

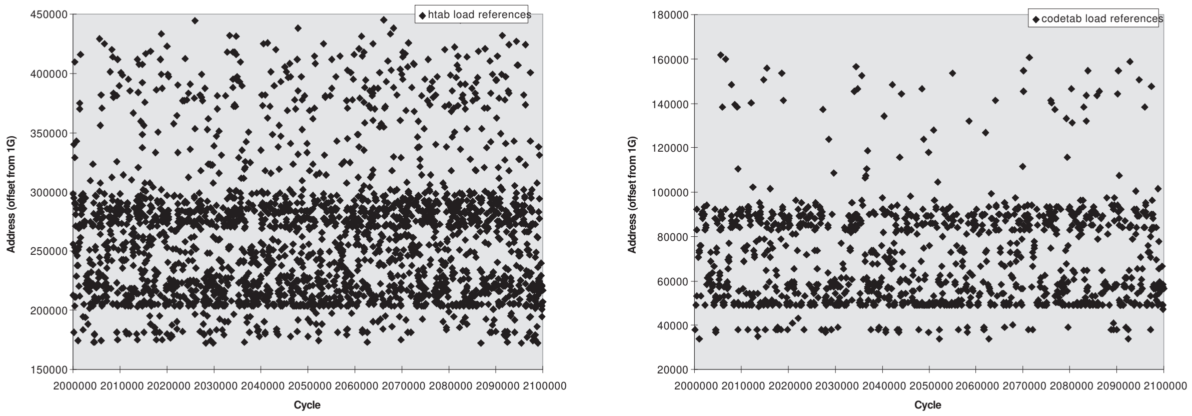
Line	Hash Table	Dynamic Execution Count	Miss Ratio	Reuse Ratio
1	htab	999 999	78.9%	29.2%
2	codetab	566 776	70.8%	30.0%
3	htab	1 803 911	91.4%	15.6%
4	codetab	182 336	89.1%	11.5%

with 32-byte lines.[2] Also, the profiler monitored reuse ratios.[3] The table shows that, indeed, the hash table load accesses have high miss ratios and little reuse of the accessed data.

In order to obtain a clearer picture of how the hash tables are accessed throughout the dynamic execution of the program, the accesses were profiled as explained above and plotted the address distribution for a given execution phase. The profiling results for a 100 000-cycle sample of *compress* are shown in Figure 4.2. The memory access distribution for `htab` is shown in Figure 4.2(a), where `htab` starts at address 171680 (all addresses are offsets from a base address of 1073741824, or 1 GB). As the `htab` distribution shows, much of `htab` is relatively sparsely accessed, except for two bands that are heavily accessed. These bands are located roughly from addresses 200000 to 220000 and 257000 to 300000. Looking at several other execution phases of *compress* shows that this pattern is consistent throughout the execution.

[2]This profiler is a simplified version of the detailed simulator described in Chapter 3. Unlike the simulator, the profiler assumes a single-issue, in-order machine and zero-cycle load latencies to simplify handling back-to-back accesses to the same cache block.

[3]The reuse ratio is calculated in the following way. If load A accesses a cache block (whether a hit or miss), on a following hit by load B to that cache block, the reuse counter for load A is incremented once. If another access by load C is a hit to the same cache block, the counter for load B is incremented, and so on. The total number of reuses counted for a load, divided by its dynamic execution count, is that load's reuse ratio. Therefore, some of the hits will have reuse, as will some of the misses.



(a) `htab` (b) `codetab`

Figure 4.2 Memory access distributions for `htab` and `codetab`.

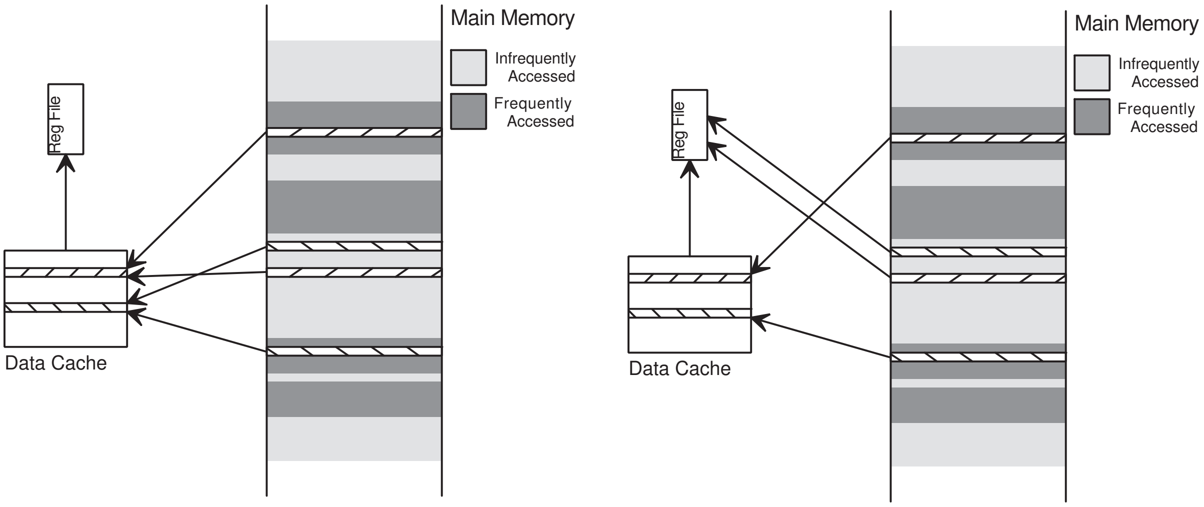
Analogous to Figure 4.2(a), Figure 4.2(b) shows the access distribution for `codetab`. The access patterns of the two figures look similar since `codetab` is accessed with the same index as `htab`.

There are two main aspects on which the bypassing decisions can be based. One possibility is to base the decision on the program counter of the load generating the access. A program-counter-based scheme would be preferred if the same memory address had different amounts of locality, depending on the code surrounding each load reference. The other possibility is to base the decision on the memory address of the accessed data. An address-based scheme would be preferred if the same memory address has similar amounts of locality in each phase of the program execution, especially if the code surrounding the load instructions does not dictate the data locality. The memory access distributions of Figure 4.2 would be better predicted by the latter approach. In *compress* there are only two load instructions in the main loop body that access `htab`. However, the distributions show that these loads can access data with dramatically different usage patterns, even during small time intervals. Schemes that decide where to place the data in the cache hierarchy based on the load instruction accessing that data, whether in a static or dynamic manner, must face the challenge of giving each dynamic instance of a load instruction different treatment. Otherwise, information is lost, and some data will be mishandled. While other benchmarks may contain patterns better predicted by the program counter, the *compress* behavior motivated the use by this dissertation of memory addresses in the bypass decision. Another possibility would be to use some hybrid of the program counter and memory address, which would be an interesting area of future research.

Figure 4.3 shows how accesses to data with different usage frequencies should be handled. In Figure 4.3(a) accesses to differently accessed regions of memory will map into the same cache lines, causing conflict or capacity misses. Assume that an access to a block in an infrequently accessed region of the memory misses in cache and that the conflicting block that would be replaced from the cache under a normal cache management policy is from a heavily accessed region. Rather than replacing the heavily accessed block, which has a much greater chance of being reused in the near future, the missing block would instead bypass the cache. In this case, the missing block would not be placed in the cache, as illustrated in Figure 4.3(b). Bypassing infrequently accessed data when it conflicts with much more frequently accessed data will result in less cache pollution, and therefore increased reuse of more frequently accessed data, resulting in an overall increase in the hit ratio.

4.1.2 Upper bound analysis

In order to gauge the potential effectiveness of cache bypassing, an upper-bound analysis was performed. This section describes the algorithms used to determine an upper bound on the hit ratios that can be obtained with cache bypassing. Optimal replacement algorithms with bypass have been developed in the past [48], [49]; however, these only consider the basic



(a) conflicts between data with different usage patterns

(b) bypassing less frequently accessed addresses

Figure 4.3 Conflict misses in *compress*.

case where bypassed data is not buffered. A new algorithm was developed for this dissertation with extendability to the case of a *bypass buffer*, described later, in mind. Additionally, this dissertation did not have the requirement of forward-processing the trace, as in other work [49]; therefore, the resulting algorithm is much less complex.

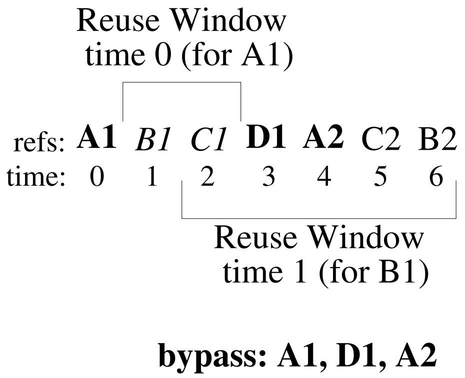
To find the optimal bypassing decisions for all references mapping to each associative cache set, the trace is processed in reverse chronological order, and each reference is compared to all later references within the potential *reuse window* for the corresponding cache set. A *reuse window* for a cache set at time t is defined as the set of chronologically later references which could reuse the data supplied by reference R_t, given the bypassing decisions that have already been determined. In other words, the data supplied by R_t is in the cache during all accesses in its reuse window, but is displaced by the last reference in its reuse window. All references to the same cache set occurring between R_t and the last reference in the reuse window must also be in the reuse window.

Figure 4.4(a) shows a sample reference stream, where all references are assumed for simplicity to map to the same two-way associative cache set. References to the same cache block in memory are labeled with the same letter, and distinct references to the same cache block are numbered for clarity. Assume that $B2$ is the last reference in the trace, and that it has already been determined that $D1$ and $A2$ should bypass the cache. Consider the reuse window at time 1, corresponding to the references that can reuse reference $B1$. At time 1, reference $B1$ will be cached in the least-recently used (LRU) entry, and will become the most-recently used (MRU) entry. At time 2, reference $C1$ will be cached in the new LRU entry, becoming MRU. Therefore, at the end of time 2, the cache set will contain $\{B,C\}$. Next, $D1$ and $A2$ bypass the cache, so the set contents are unchanged. Then, $C2$ reuses the data cached by $C1$, and the cache contents are again unchanged. Finally, $B2$ reuses the data cached by $B1$. Therefore, the reuse window at time 1 contains all later references in the sample reference stream.

Now consider reference $A1$ at time 0. Reference $A1$ will be cached in the LRU entry and become MRU. Next, $B1$ is cached in the new LRU entry and becomes MRU. The cache set after time 1 contains $\{A,B\}$. Next, $C1$ will displace the LRU entry, which is now the data cached by $A1$. Therefore, no references after $C1$ (time 2) can possibly reuse $A1$ with the given caching decisions, and $A1$ can also bypass the cache. The optimality of this approach will be proven later in this section.

The main steps that must be completed by the algorithm for a given reference R_t are the following:

1. Determine if any reference in the corresponding cache set's reuse window has reuse with R_t.



(a) reference stream and reuse windows

Alg Iter	Time t	Ref$_t$	Reuse Window$_t$	Cache Set	Ref Type	Hit/ Miss
1	6	B2			N	H
2	5	C2	B2		N	H
3	4	**A2**	C2 B2		N	M
4	3	**D1**	**A2** C2 B2		N	M
5	2	*C1*	**D1** **A2** C2 B2		R	M
6	1	*B1*	*C1* **D1** **A2** C2 B2	*C*	R	M
7	0	**A1**	*B1* *C1*	*B, C*	N	M

(b) illustrated bypassing algorithm

Figure 4.4 Example reference stream and corresponding reuse window and LRU cache set at each step in the algorithm. All references in the stream are assumed to map into the same cache set, which is two-way set associative.

2. Update the reuse window for the next previous reference R_j ($j < t$) to the same cache set.

During the first step, R_t will be marked as one of two types of references:

- $R{-}ref$: A reference that has a corresponding later reuse.
- $N{-}ref$: A reference that has no corresponding later reuse.

Therefore, an $N{-}ref$ that misses can bypass the cache without harming the hit ratio. As a result, the reuse window can contain an arbitrary number of $N{-}refs$, possibly greater than the cache associativity, since they do not require cache space. In the example of Figure 4.4(a), references $B1$ and $C1$, shown in italics, are both $R\text{–}refs$, since they are reused by references $B2$ and $C2$, respectively. All other references are $N\text{–}refs$. Each $R\text{–}ref$ has only one corresponding reuse, and its corresponding reuse may also be an $R\text{–}ref$. Therefore, each set of references that access a particular block in the cache before that block is replaced form a *reuse chain* of the form $(R-ref\rightarrow)^{*}N-ref$. Note that the number of cache hits in each *reuse chain* is equal to the number of $R-ref$s, although the first $R-ref$ is a miss, and the $N-ref$ is a hit if there was at least one $R-ref$. In Figure 4.4(a), the reuse chains are $\{\{A1\}, \{B1 \rightarrow B2\}, \{C1 \rightarrow C2\}, \{D1\}, \{A2\}\}$. References $C2$ and $B2$ will be cache hits, even though they are $N\text{–}refs$. Reuse chains that do not contain any $R-refs$ ($\{A1\}$, $\{D1\}$ and $\{A2\}$) will miss, and simply bypass the cache.

The algorithm assumes that all potential cache reuses have equal weight, and is detailed in Figure 4.5. The reuse windows are maintained in an array of *cache_queues*, one for each set in the cache. For each reference in the trace the corresponding reuse window is traversed once. The references in the reuse window are examined in chronological order, and the addresses are compared to determine if the current reference will result in a reuse within the window. If so, it is marked as an $R\text{–}ref$. At the same time, a temporary cache set is maintained to determine when the cache set will fill, forcing the displacement of any earlier references. This allows the second step described above to be performed in the same pass through the reuse window. Each reference is added to the set, if the address is not already in the set and if it is marked for reuse ($R\text{–}ref$). Once the size of the set matches the associativity, none of the subsequent references in the reuse window can obtain reuse with any references prior to the current reference, since any earlier references must necessarily be displaced from the cache under optimal bypassing before

```
/* define */
CA_ASSOC = cache associativity;
CA_ENTRIES = number of sets in cache;
CACHE_INDEX(address) = set index corresponding to address;

algorithm upper_bound_bypass
{
    cache_queues[0..CA_ENTRIES] = NULL;

    for each address (reverse order) do:
    {
        new_entry->flag = N-ref;
        set = NULL;
        ca_entry = CACHE_INDEX(address);
        done = FALSE;

        for each entry in cache_queues[ca_entry] do:
        {
            if (!done)
            {
                if (new_entry->flag == N-ref && address == entry->address)
                    new_entry->flag = R-ref;

                if ((entry->addr not in set) && (entry->flag == R-ref))
                {
                    add entry to set;
                    if (|set| == CA_ASSOC) done = 1;
                }
            }
            else dequeue entry;
        }
        enqueue new_entry in cache_queues[ca_index];
    }
    dequeue all remaining entries in cache_queues[0..CA_ENTRIES];
}
```

Figure 4.5 Algorithm to determine optimal bypassing decisions.

the subsequent references occur. Therefore, the subsequent references are simply discarded from the reuse window.

Proof. The proof of the optimality of this algorithm can be shown by induction. Assume for simplicity that all references in the trace map to a single set in the cache.

Basis. If the trace is of length T, then the algorithm will process reference R_T at step 0. For step 0, the reuse window is empty, as there are no later references, and R_T has no reuse and is marked an $N{-}ref$. For step 1 the reuse window contains R_T, and if reference R_{T-1} matches

R_T, then there is a reuse, and R_{T-1} is an $R{-}ref$. Otherwise it is an $N{-}ref$. It is trivially true that there can be no more reuse than is currently marked. Therefore, the algorithm is optimal for $k = 1$.

Induction. Now consider step $k > 1$, and assume the first k assignments are optimal, and assume the reuse window contains w references. If R_{T-k} has reuse in the reuse window, it is an $R{-}ref$, and a cache hit has been added. Since every $R{-}ref$ can only create one hit, and any change to the decisions in the first k steps will only reduce the number of hits (since the first k steps are optimal), this is again optimal. If R_{T-k} does not have reuse in the reuse window or anywhere later in the trace, then there is no possibility of reuse, and it is marked an $N{-}ref$ without any loss of optimality. However, consider if R_{T-k} does not have reuse in the reuse window, but does have a match in the trace after the reuse window, say with R_l, where $l - k > w$. Since R_l is not in the reuse window, then the reuse window must contain A unique $R{-}refs$, where A is the cache associativity. In order to force R_{T-k} to have reuse with R_l, one cache location in the set must be taken from an $R{-}ref$, say R_i, in the reuse window (it could also require taking the cache location from later $R{-}refs$ outside the reuse window, but that would clearly be unoptimal and is not considered here). This will add one cache hit for the reuse pair $R_{T-k} \rightarrow R_l$, but remove the hit to R_j corresponding to R_i. Figure 4.6 shows the example reference stream with reference R_i in the reuse window, and a solid directed line pointing to its reuse with R_j. The potential reuse between R_{T-k} and R_l is denoted by the dashed directed line. The only way a net gain in cache hits can occur is if the removal of the reuse $R_i \rightarrow R_j$ allows another reuse $R_m \rightarrow R_n$ to occur. The $R_m \rightarrow R_n$ must require the same cache location as, and overlap with, $R_i \rightarrow R_j$. Otherwise, $R_m \rightarrow R_n$ could have been achieved without removing reuse $R_i \rightarrow R_j$, in which case the first k assignments were not optimal, violating the assumption.

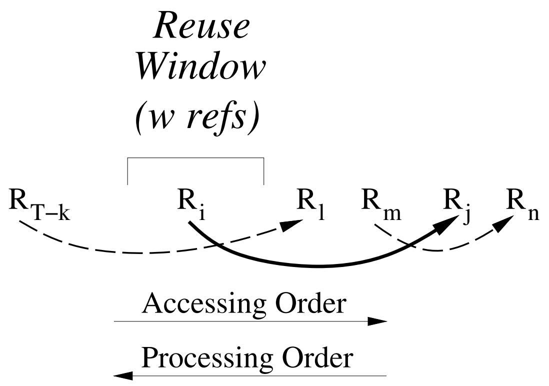


Figure 4.6 Example reference stream used in proof of the optimality of algorithm `upper_bound_bypass`. Reuses and potential reuses between reference pairs are shown with arrows.

Therefore, $m < j$. Additionally, $R_m \rightarrow R_n$ must utilize the same cache location as $R_{T-k} \rightarrow R_l$, in which case $l \leq m$. Therefore, $i < l \leq m < j$. The potential reuse pair R_m and R_n are also shown in Figure 4.6. Because the algorithm works backwards and assigns reuses greedily, if $i < m < j$, then the algorithm would have seen reuse $R_m \rightarrow R_n$ before $R_i \rightarrow R_j$, as illustrated in Figure 4.6, and would have assigned the cache location to $R_m \rightarrow R_n$, not $R_i \rightarrow R_j$. Hence, forcing reuse between R_{T-k} and R_l cannot result in a net gain in cache hits, given that the first k assignments were optimal.

Figure 4.4(b) shows the reuse window and the cache set contents at each iteration of the algorithm. An examination of this stream will show that the hit ratio in a conventional cache would be 0. Reference $B2$ is examined first, as it is the last reference in the stream. At this point the reuse window is empty, so there is no reuse, and $B2$ is marked as an $N\!-\!ref$. As such, it is not placed in the cache set. Reference $C2$ does not match any address in the reuse window, which simply contains $B2$, and is also marked as an $N\!-\!ref$. This continues until $C1$ is examined and found to match $C2$ in the reuse window. Therefore, $C1$ is marked as an $R\!-\!ref$, and $C2$ will be a cache hit. Because $C1$ is an $R\!-\!ref$, it is placed in the cache set.

However, the cache set does not overflow, so no dequeuing is necessary. Next, $B1$ is found to be an $R{-}ref$, matching $B2$. Now the cache set contains both B and C, which equals the cache associativity, and would displace from the set any earlier references ($A1$), which can no longer be reused by references after $C1$. Therefore all references following $C1$ are removed from the reuse window. Note that for efficiency the algorithm will actually dequeue the later references from the reuse window during the next iteration; however, for simplicity the example is shown as if the dequeueing occurred at the end of the current iteration. Finally, when $A1$ is examined, the reuse window no longer includes $A2$; therefore, $A1$ is marked as an $N{-}ref$. The table also shows these reference markings and the new cache access results for each corresponding reference. In the ideal case, $N{-}refs$ that miss in the cache would bypass, in this case $A1$, $D1$ and $A2$. The new hit ratio is 2/7.

While the bypassing algorithm described above will reduce misses by not caching data that has no reuse, it is possible that multiple cache blocks, all with potential reuse, compete for the same direct-mapped line. Therefore, Section 4.2 will propose adding a *bypass buffer* to the cache hierarchy. Bypassed data is placed in this small set-associative buffer, which is then accessed in parallel with the data cache. In this situation, some data with a few short-term reuses can be bypassed into the buffer, which exploits the reuse, while not displacing more frequently-accessed data from the main cache. With a bypass buffer, the upper-bound analysis becomes significantly more complex. Now each entry can be placed in one of two cache sets, and the set of references with which a given reference will conflict in the different cache sets are not identical. It is also unclear when to assign a reuse chain to the main cache or the bypass buffer.

Consider the case when a reuse is detected between two references in both caches, meaning that the reuse could be exploited in either cache. One possibility would be to assign the reuse to $cache_1$ and remove the references from the reuse window in $cache_2$. However, it is possible that another earlier (chronologically) reference accesses the same data, but the intermediate references are such that the data would be displaced from $cache_1$ before the reuse, but not from $cache_2$. An example of this situation is shown in Figure 4.7(a), where $cache_1$ is direct-mapped, and $cache_2$ is two-way set-associative. Assume that addresses A and B map to the same set in both caches. Reuse $A2 \rightarrow A3$, found during step 1, can occur in either cache. If $cache_1$ is chosen, and the references are removed from the reuse window of $cache_2$, then no reuse is detected for reference $A1$, since $B1$ displaces the references to A from the reuse window of direct-mapped $cache_1$. Since $cache_2$ is two-way set associative, the reuse $A1 \rightarrow A2$ could have been exploited in $cache_2$.

On the other hand, if the reuse is left in both caches' reuse windows, since the reuse chain must eventually be assigned to a single cache, the reuse window of the other cache will be pessimistically restricted by the data. An example of this situation is shown in Figure 4.7(b), for the same cache associativities described above. Assume that A, B, and C map to the same sets in both caches. The reuses $A1 \rightarrow A2$ and $B1 \rightarrow B2$ will be detected in both caches. If these references are left in both caches' reuse windows, the reuse $C1 \rightarrow C2$ will not be detected, as $C2$ would have been pessimistically replaced from both reuse windows before $C1$ is processed.

Therefore, some simplifications to the algorithm are made, which result in an upper bound that may not always be achievable in practice. However, this should still result in a useful comparison of the bypass mechanism described in Section 4.2 to an upper bound.

Now, each reference can be marked as one of five types of references:

accessing order $\longrightarrow$

processing order $\longleftarrow$

A1 B1 A2 A3

step	*cache1 reuse window*	*cache2 reuse window*
0	A3	A3
1	A2,A3	–
2	B1	B1

(a) reuse assigned to $cache_1$ and removed from $cache_2$

accessing order $\longrightarrow$

processing order $\longleftarrow$

C1 B1 B2 A1 A2 C2

step	*cache1 reuse window*	*cache2 reuse window*
0	C2	C2
1	A2	A2,C2
2	A1,A2	A1,A2,C2
3	B2	B2,A1,A2
4	B1,B2	B1,B2,A1,A2

(b) reuses left in both caches

Figure 4.7 Example reference streams and corresponding $cache_1$ and $cache_2$ reuse windows at each step in the algorithm, for two algorithms. Here, $cache_1$ is direct-mapped, and $cache_2$ is two-way set associative.

- $R_1{-}ref$: A reference that has a corresponding later reuse in $cache_1$.
- N_1-ref: A reference that has no corresponding later reuse, but itself reuses data in $cache_1$.
- $R_2{-}ref$: A reference that has a corresponding later reuse in $cache_2$.
- N_2-ref: A reference that has no corresponding later reuse, but itself reuses data in $cache_2$.
- $N{-}ref$: A reference that has no corresponding later reuse, and does not reuse data itself in either cache.

The algorithm is outlined in Figures 4.8-4.9.

```
/* define */
CA1_ASSOC = cache 1 associativity;
CA2_ASSOC = cache 2 associativity;
CA1_ENTRIES = number of sets in cache 1;
CA2_ENTRIES = number of sets in cache 2;
CACHE1_INDEX(address) = set index in cache 1 corresponding to address;
CACHE2_INDEX(address) = set index in cache 2 corresponding to address;

algorithm upper_bound_bypassbuf
{
    cache1_queues[0..CA1_ENTRIES] = NULL;
    cache2_queues[0..CA2_ENTRIES] = NULL;

    for each address (reverse order) do:
    {
        new_entry->flag = N-ref;
        ca1_entry = CACHE1_INDEX(address);
        ca2_entry = CACHE2_INDEX(address);

        /* Give priority to bypass buffer (cache 2) reuse */
        check_cache(2,1,1);

        /* If no reuse found yet, check cache1 */
        if (new_entry == N-ref)
            check_cache(1,2,2);

        /* If no reuse found yet, check for C2->C1 reuses */
        if (new_entry == N-ref)
            check_cache(2,1,2);

        /* If no reuse found yet, check for C1->C2 reuses */
        if (new_entry == N-ref)
            check_cache(1,2,1);

        if (new_entry->flag == N-ref)
        {
            enqueue new_entry in cache1_queues[ca1_index];
            enqueue new_entry in cache2_queues[ca2_index];
        }
    }
    dequeue all remaining entries in cache1_queues[0..CA1_ENTRIES];
    dequeue all remaining entries in cache2_queues[0..CA2_ENTRIES];
}
```

Figure 4.8 Algorithm to determine upper-bound bypassing decisions in the presence of a bypass buffer.

```
algorithm check_cache(i,j,j)
{
    set = NULL;
    done = FALSE;

    /* Check cachei */
    for each entry in cache(i)_queues[ca(i)_entry] do:
    {
            if (!done)
            {
                if (new_entry->flag == N-ref && address == entry->address &&
                        (entry->flag != Nk-ref || entry->flag != Rk-ref))
                {
                    new_entry->flag = Ri-ref;
                    if (entry->flag == N-ref) entry->flag = Ni-ref;
                    enqueue new_entry in cache(i)_queues[ca(i)_index];
                    enqueue new_entry in cache(j)_queues[ca(j)_index];
                }

                else if ((entry->addr not in set) &&
                        (entry->flag == Ni-ref || entry->flag == Ri-ref))
                {
                    add entry to set;
                    if (|set| == CAi_ASSOC) done = 1;
                }
            }
            else dequeue entry from cache(i)_queues[ca(i)_entry];
    }
}
```

Figure 4.9 Routine to support upper-bound bypassing decisions.

Figure 4.8 shows the main body of the bypassing upper-bound algorithm. Now two arrays of reuse windows are maintained, one for each cache, the main data cache ($cache_1$) and the bypass buffer ($cache_2$). Many of the steps are the same as those shown in the bypassing algorithm of Figure 4.5, except that two caches must be checked and maintained. Because the bypass buffer is designed to hold data with shorter-term temporal locality, the algorithm first checks the bypass buffer for reuse. If none is found, the main cache is checked. The checking is performed in the `check_cache` routine, shown in Figure 4.9. Although it is a heuristic to check the bypass buffer first, this heuristic will rarely result in less than the maximum achievable performance, as will be explained later. The last two calls to `check_cache` will be explained later. The `check_cache` routine in Figure 4.9 is similar to the checking loop for the bypassing algorithm of Figure 4.5.

The simplification made to solve the problem described earlier is to place references in both of the caches' reuse windows, however, this reference is only counted for the purposes of updating the reuse window in the cache containing the reuse. Therefore, the reuse windows contain extra references, allowing more reuses to be detected by the algorithm than can possibly occur. However, this prevents a reuse which could have occurred in both caches from pessimistically shortening one of the windows and at the same time allows the reference to be "reassigned" to the other cache if it is later found to result in a longer reuse chain. Returning to the example shown in Figure 4.7(b), reuses $A1 \rightarrow A2$ and $B1 \rightarrow B2$ would be left in both caches' reuse windows, but would be assigned to one cache, and not counted in the other cache's reuse window. An examination of the reference stream will show that, regardless of the cache to which the reuses $A1 \rightarrow A2$ and $B1 \rightarrow B2$ are assigned, with this algorithm $C2$ would not be displaced from both reuse windows before $C1$ is processed, and the reuse $C1 \rightarrow C2$ will be detected.

Algorithm `upper_bound_bypassbuf` first checks the two caches for reuse to a reference assigned to the same cache. However, if no reuse is found, the algorithm then checks each cache for reuse to a reference already assigned to the other cache. This is performed by the last two calls to `check_cache`, which are called with slightly different arguments. If one such reuse is found, the tail reference of the new reuse is "reassigned" to this cache. There is one aspect of this simplification that can still make the results pessimistic. When references are "reassigned" to another cache as described above, the reuse window originally containing the reference may have been pessimistically shorted before the reassignment. However, it was found that reassignments generally constituted less than 1% of the accesses in each benchmark. Each reassignment can result in at most one extra (optimistic) hit in the cache to which the reuse chain is reassigned, and at most one extra (pessimistic) miss in the cache to which the reuse chain was previously assigned. Therefore, if the optimistic and pessimistic effects do not completely cancel each other out, the effect on the hit ratio will be minimal, which the results below will demonstrate.

Additionally, if no reuse was found to occur for a reference, it is inserted into both caches' reuse windows by algorithm `upper_bound_bypassbuf`. It is then not counted in either reuse window until it is assigned to one cache by an earlier reference found to have reuse with this reference in a particular cache. References that are neither reused by a later reference nor found to reuse earlier references remain $N{-}refs$, and therefore are treated as references which bypass without being placed in the bypass buffer. This again can result in an unachievable upper bound with this algorithm.

Table 4.2 presents the hit ratios that are obtained performing bypassing both with and without bypass buffers. Results for a direct-mapped 16-KB cache with 32-byte lines are shown. The bypass buffer, when used, is four-way set associative with thirty-two 32-byte lines. Also shown are the hit ratios obtained with a conventional direct-mapped cache of the same size (*base hit ratio*). The percentage of accesses resulting in reassignments, causing both optimistic and pessimistic effects on the hit ratio as described above, is shown after the $\pm$ when using a bypass buffer. The results show that bypassing can potentially obtain significant improvements in the hit ratios. However, a bypass buffer is required, even under optimal bypassing, to obtain uniformly significant improvements across all benchmarks.

Therefore, cache bypassing can achieve significant improvements in the cache hit ratios of integer benchmarks. The following sections describe one implementation of a cache bypassing

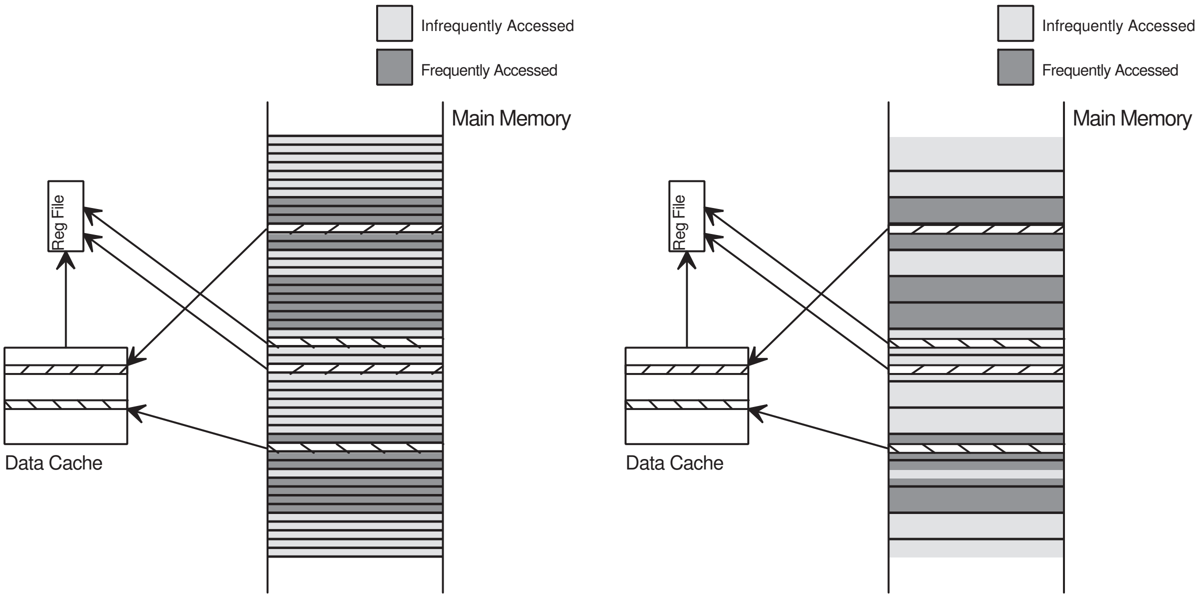
Table 4.2 Hit ratios for 16-KB direct-mapped cache with 32-byte lines both with and without bypassing.

Benchmark	Base Hit Ratio	Bypassing Hit Ratio: No Bypass Buffer	Bypassing Hit Ratio: With 1-KB Bypass Buffer
026.compress	69.12%	75.26%	76.38±0.32%
072.sc	93.02%	93.45%	94.78±0.62%
099.go	93.42%	94.95%	98.33±0.32%
147.vortex	95.24%	96.54%	98.80±0.19%
Pcode	80.50%	83.12%	87.48±1.26%
lmdes2_customizer	85.17%	88.25%	91.96±0.53%
085.cc1	93.71%	94.46%	96.37±0.45%
130.li	93.59%	93.96%	95.66±1.18%
134.perl	95.16%	95.57%	97.42±0.10%
124.m88ksim	98.53%	98.81%	99.44±0.02%
word	91.55%	93.65%	97.35±0.43%
excel	93.68%	94.90%	97.78±0.29%
photo	99.11%	99.29%	99.67±0.07%

optimization which attempts to achieve these upper-bound hit ratios. After simulation results show the performance of this implementation in Chapter 3, the performance will be compared to the upper bounds presented in this section, and the opportunities missed by the implementation are examined.

4.1.3 Macroblocks

Performing selective cache bypassing requires some method of monitoring memory access behavior. As noted in Section 4.1.1, the scheme examined in this paper is address based rather than program-counter based, and therefore must monitor the behavior of different memory regions. Ideally, the usage frequencies of all cache block size data in memory would be monitored, as illustrated in Figure 4.10(a). While this would provide the most accurate information, it would result in an unmanageably large amount of information. Instead, groups of adjacent cache block size data are combined into larger blocks called *macroblocks*, as shown in Figure 4.10(b). The size of the macroblocks should be large enough so that the total number of macroblocks



(a) tracking each cache block (b) tracking macroblocks

Figure 4.10 Tracking usage frequencies.

residing in the accessed portion of memory is not excessively large, but small enough so that the accessing frequency of the cache blocks contained within each macroblock is relatively uniform. If the accessing frequency of each macroblock is monitored through some hardware mechanism, then it is possible to determine on a macroblock basis whether to cache the contained data.

In order to determine the best size of a macroblock in practice, the uniformity of cache block access frequencies within the macroblocks was studied for several macroblock sizes. The number of accesses to each cache block in memory was first profiled. Then, for each macroblock size, the mean and standard deviation of the number of accesses to the cache blocks contained within each macroblock in memory were computed. For a high intramacroblock accessing uniformity most of the macroblocks should have relatively small standard deviations.

Figure 4.11 shows the results for macroblock sizes of 256 bytes, 1 KB, 4 KB, and 16 KB for *compress*. The y-axis is a $\log_{10}$ scale of the standard deviation divided by the mean (in order to normalize the results), and the x-axis is the percentage of macroblocks with a standard deviation

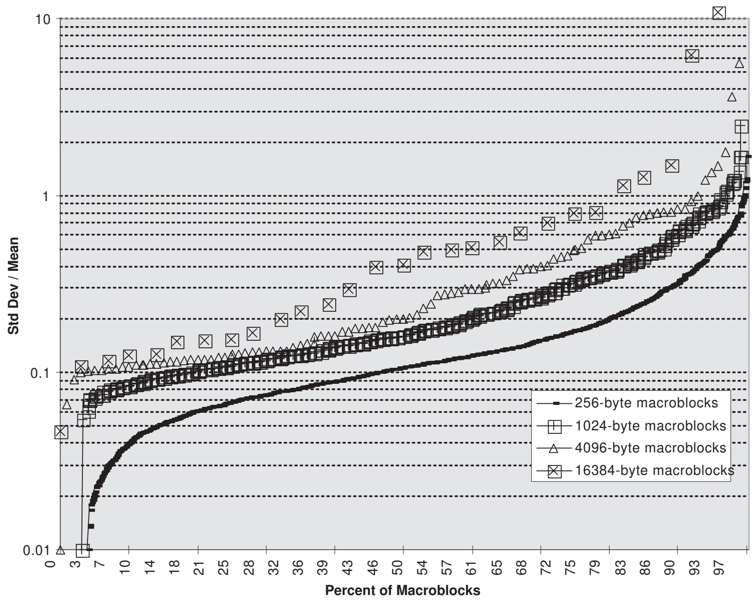
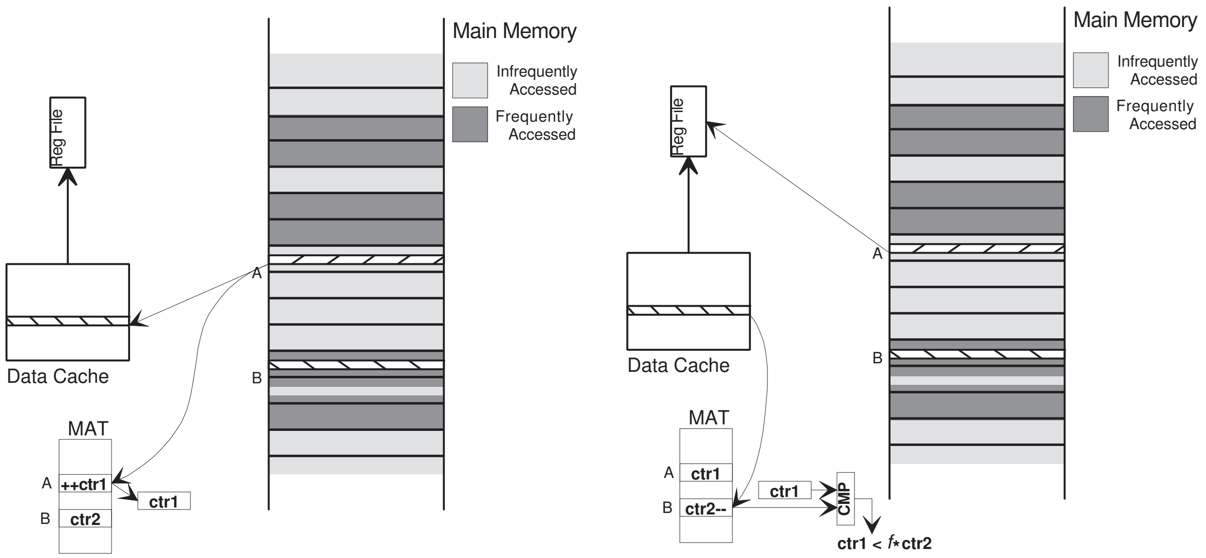


Figure 4.11 Macroblock access uniformity.

divided by mean less than or equal to the plotted value. The curve for 256-byte macroblocks has the lowest standard deviations; however, using 256-byte macroblocks may result in too many total macroblocks to feasibly monitor. Using 1-KB macroblocks, which would include four times as many cache blocks per macroblock, may be much more feasible and still result in most macroblocks having high uniformity. For this size, about 60% of the macroblocks lie within 20% of the mean, with almost 90% within 50% of the mean. The 4-KB curve is only slightly higher, but the 16-KB curve does not look quite as good. The curves for other benchmarks have similar characteristics, and will not be presented here due to space constraints.

Section 4.3 examines the overall performance effects of varying the granularity of the macroblocks.



(a) accessing *ctr1* (b) accessing *ctr2*

Figure 4.12 MAT operation.

4.2 Hardware

4.2.1 Memory address table

The macroblocks are monitored at run time using a table in hardware called a *memory address table*, or *MAT*. The MAT ideally contains an entry for each macroblock. Each entry in the table contains a saturating counter, where the counter value represents the frequency of accesses to the corresponding macroblock.

On a memory access, a lookup in the MAT of the corresponding macroblock entry is performed in parallel with the data cache access. If no entry is found, a new entry is allocated and initialized, as will be discussed later. If an entry is found, the counter is incremented. Also, the counter value (*ctr1*) must be saved in a register for possible use in the next step. An example of this operation is shown in Figure 4.12(a), where data in macroblock A is accessed.

If the data cache access resulted in a hit, the access proceeds as normal, and the counter value is ignored. If the access resulted in a cache miss, the cache controller must look up the

MAT counter corresponding to the cache block that would be replaced to determine which data is more heavily accessed, and therefore more likely to be reused in cache, as shown in Figure 4.12(b) where data in macroblock B would be replaced. This counter value (*ctr2*) is then decremented and compared to the counter value corresponding to the missing access. The actual comparison performed is

$$ctr1 < f * ctr2 \tag{4.1}$$

where f is some fraction. If the above inequality is satisfied the fetched data will bypass the cache. Otherwise, it replaces the existing cached data as normal.

As mentioned above, the counter corresponding to the currently cached block (*ctr2*) is decremented. This is to ensure that the counter values will eventually decrease, so that after a transition to another phase of the program execution, new data can replace data that is now unused. The rationale for decrementing counters on missing accesses to conflicting data is that the data currently residing in the cache must justify remaining cached when there is heavy contention for that cache location. Therefore, the heavier the contention for a particular cache location, the more the cached data must be reused to maintain a counter large enough to satisfy Equation (4.1).

Rather than compare *ctr1* and *ctr2* exactly, in Equation (4.1) *ctr1* is compared to some fraction f of *ctr2*. Choosing f less than 1.0 will conservatively prevent bypassing when the counter values are almost the same.

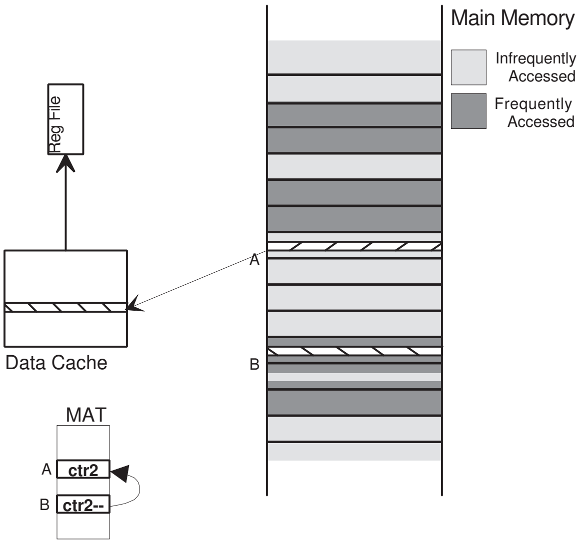
Also, bypassing is avoided when the MAT contains no entry for one of the macroblocks, so that the cache will then default to its normal replacement behavior when there is insufficient information. In the case where there is no counter for the address being accessed this can be achieved by setting all bits to 1 in the register which holds *ctr1* for the comparison. When there is no counter found for the data residing in cache (*ctr2*), *ctr1* is compared to 0. Both of these are simply a matter of multiplexing in either the counter value read from the MAT or the appropriate constant, using the valid bit as a selector.

When a new MAT entry is allocated, the counter value must be initialized to some value. One possibility is to initialize it to 0. This will result in more aggressive bypassing, since it will take longer for the new counter to "catch up" to the older cached data's counter. Another possibility is to initialize to the conflicting data's counter value, as shown in Figure 4.13(a). The new counter will start at an equal position, resulting in more selective bypassing. Simulations showed that initialization of *ctr1* with *ctr2* tends to produce better bypassing decisions.

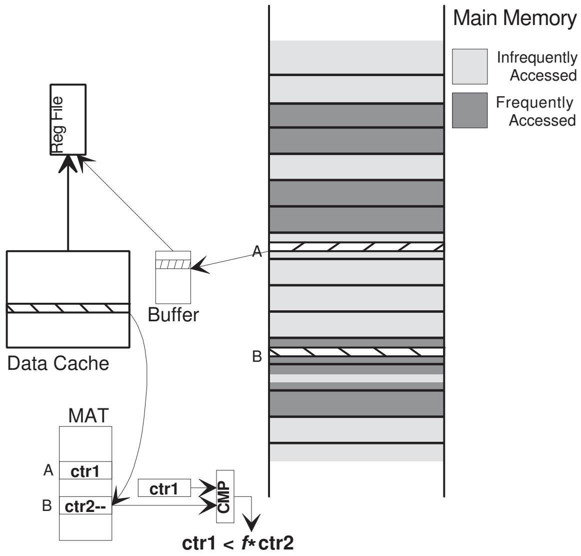
4.2.2 Bypass buffer

The mechanism described in Section 4.2.1 uses accessing frequency to estimate the likelihood of data locality. However, there may be temporal or spatial locality within the block, even when the total accessing frequency is relatively low. In this case the MAT scheme will bypass some data that may have otherwise had a few hits before being displaced from the cache. More than one additional miss will be incurred by not caching that data, whereas only one miss is removed by not displacing the much more frequently accessed data.

To optimize performance in this situation, bypassed data is placed in a small set-associative buffer, as shown in Figure 4.13(b). This *bypass buffer* will be accessed in parallel with the main cache. As a result, the bypassed data is held close to the processor for a short time, allowing much of the data locality of the infrequently accessed data to be exploited. However, unlike victim caches, data will never be swapped into the data cache on hits to the bypass buffer, as the low accessing frequency indicates less likelihood of long-term reuse. As illustrated in



(a) counter initialization to *ctr2* value



(b) MAT operation with a buffer

Figure 4.13 MAT operation details.

Section 4.1.2, adding a bypass buffer significantly increases the potential improvements from cache bypassing, even when the inaccuracies of the implemented prediction mechanism are filtered out.

The cost of the MAT hardware will be analyzed in Section 5.4.2.

4.3 Experimental Evaluation

4.3.1 Bypassing simulation

The first set of configurations use the MAT scheme presented in Section 4.2.1. In this scheme, two independent MATs are used, an L1 MAT and an L2 MAT. Each MAT operates independently of the other, so the L2 MAT only reflects the accesses that missed in the L1 cache and therefore accessed the L2 cache. Unless otherwise stated, both MATs use $f = 1.0$, 8-bit access counters, 1-KB macroblocks, and have infinite entries. The effect of varying each of these parameters is examined.

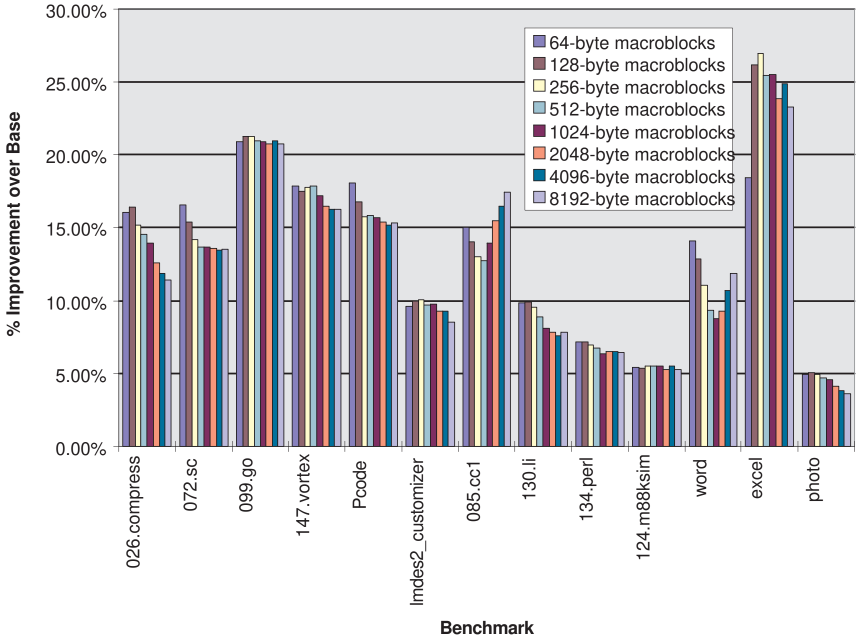


Figure 4.14 Bypassing performance improvements using different macroblock sizes and an infinite-entry MAT.

Because the upper-bound analysis presented in Section 4.1.2 showed that bypass buffers are necessary to obtain uniformly significant hit ratio improvements, all configurations examined in this dissertation utilize bypass buffers. The four-way set-associative buffers used to hold the bypassed data at the L1 and L2 caches contain 32 and 256 entries, respectively. The effect of varying the bypass buffer sizes and associativities is examined, however. Results are shown first for an infinite-entry MAT, and later the effects of limiting the number of MAT entries are examined.

4.3.2 Results

4.3.2.1 Performance of various bypassing configurations

Figure 4.14 shows the speedup of each benchmark for the MAT scheme described above, for a range of macroblock sizes using an infinite-entry MAT. Most benchmarks achieve over 10%

Table 4.3 Number of macroblocks accessed.

	Macroblocks Accessed					
	64-byte macroblocks		1024-byte macroblocks		8192-byte macroblocks	
	L1 MAT	L2 MAT	L1 MAT	L2 MAT	L1 MAT	L2 MAT
026.compress	8533	8691	549	563	72	74
072.sc	4505	5633	329	433	56	76
099.go	2129	6073	247	531	63	104
147.vortex	9697	13911	2194	2652	622	718
Pcode	27406	28775	2389	2541	397	433
lmdes2_customizer	4822	5821	342	420	51	63
085.gcc	9669	15539	1069	1671	238	345
130.li	7217	7857	692	761	119	130
134.perl	17546	18987	3802	3953	1323	1356
Word	4608	30822	1170	5299	581	1790
Excel	5556	20468	853	3310	385	1168
Photo	1984	14344	572	2729	303	1045

improvement in execution time, reaching a 25% improvement in the case of Excel. The results do vary across the different macroblock sizes, and generally become worse as the macroblock size is doubled. In the case of *085.cc1*, the performance improves as the macroblock size grows, which is an anomaly resulting from the more conservative bypassing decisions that tend to result with larger macroblock sizes. The actual performance improvement comes from the L2 cache level, whereas the more conservative bypassing decisions degrade the L1 cache performance.

However, while smaller macroblocks give improved resolution, these results do not take into account the larger number of macroblocks that must be monitored for the smaller macroblock sizes. Table 4.3 shows the number of macroblocks accessed by the L1 and L2 MATs for three different macroblock sizes. The L2 MAT accesses more macroblocks because it is a unified instruction and data cache. With 64-byte macroblocks, the L1 cache often monitored several thousand macroblocks, and the unified L2 cache monitored up to 30 822 macroblocks. Therefore, Figures 4.15-4.17 show the performance improvements for the same range of macroblock sizes, with 1024-byte, 512-entry, and 256-entry direct-mapped MATs, respectively. In these cases, the MATs were far too small to monitor significant numbers of macroblocks for the

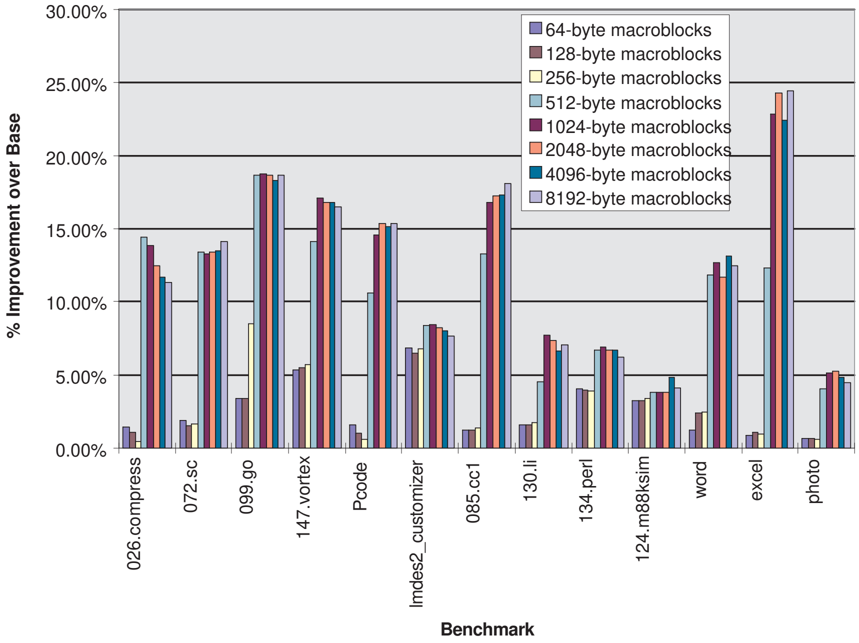


Figure 4.15 Bypassing performance improvements using different macroblock sizes and a 1024-entry direct-mapped MAT.

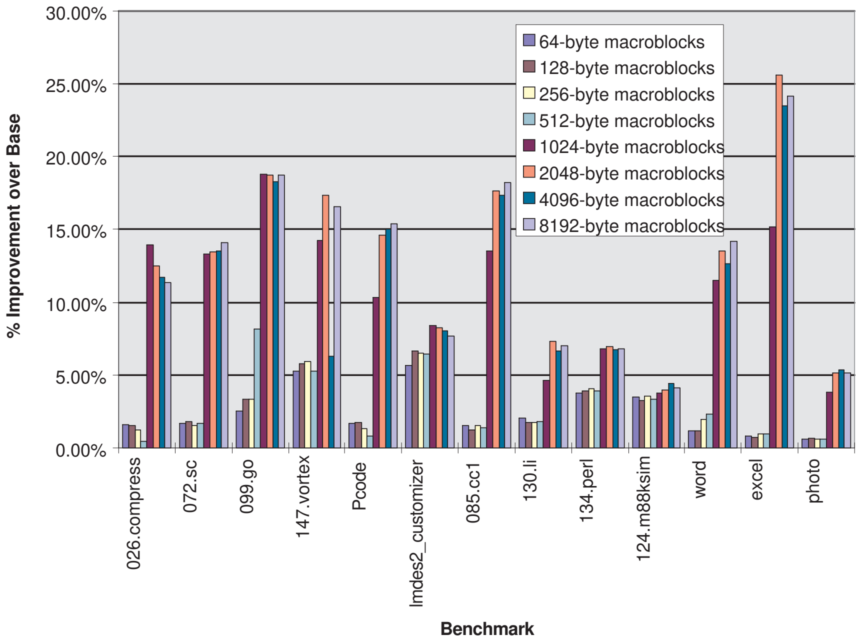


Figure 4.16 Bypassing performance improvements using different macroblock sizes and a 512-entry direct-mapped MAT.

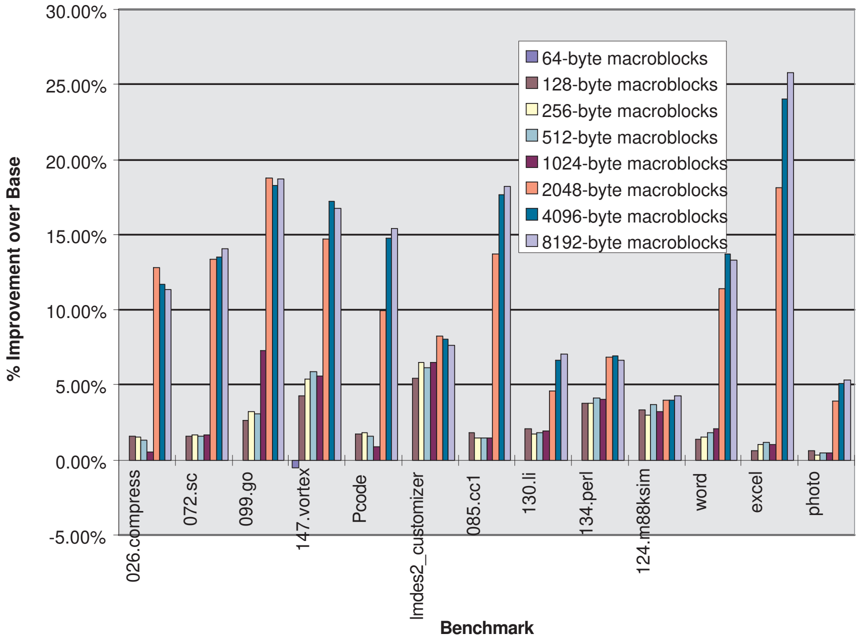


Figure 4.17 Bypassing performance improvements using different macroblock sizes and a 256-entry direct-mapped MAT.

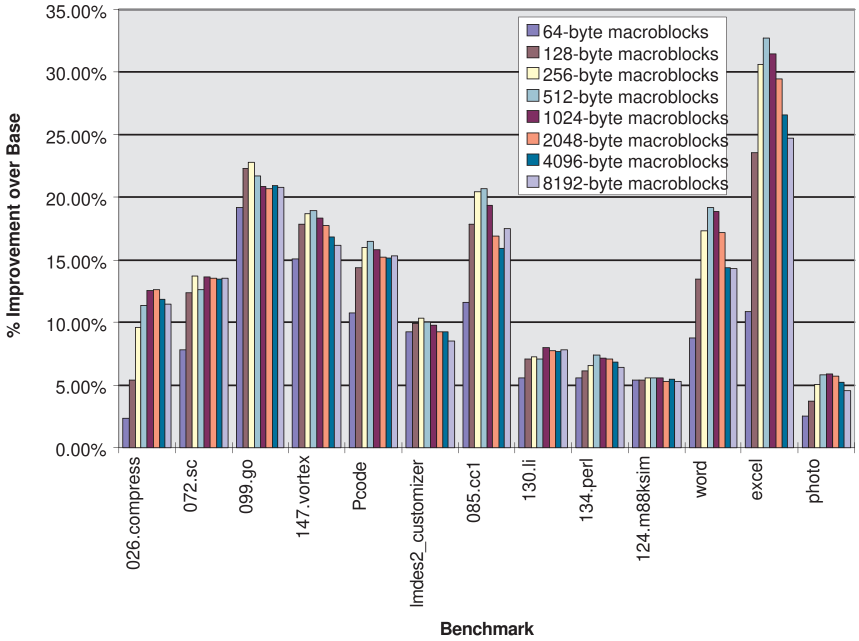


Figure 4.18 Bypassing performance improvements using different macroblock sizes and a 512-entry fully associative MAT.

smaller macroblock sizes. As the number of MAT entries shrink, the best results are achieved with larger macroblock sizes, generally 1 KB, 2 KB, or 4 KB. With these macroblock sizes, the number of macroblocks is manageable, but the accuracy is still high enough to result in good bypassing decisions. Another anomaly exists with the *085.cc1* performance. The performance of the larger macroblock sizes actually increases when using the smaller MATs. As noted above, the larger macroblock sizes decreased the performance of the L1 cache for *085.cc1* due to the more conservative bypassing decisions. With limited MAT entries, the displacement and reallocation of the L1 MAT entries actually results in more aggressive bypassing at the L1 level, increasing the overall performance of the benchmark. Finally, Figure 4.18 shows the results for 512-entry MATs that are fully associative. With higher MAT associativity, the best macroblock size is somewhat smaller.

Figure 4.19 shows the performance of the MAT scheme using 1-KB macroblocks and infinite-entry MATs using different values of f, which is used to modify the counter comparisons. With smaller f, the likelihood of bypassing reduces. The results show that, as suspected, more aggressive bypassing is desired. This is due to the use of bypass buffers, which help guard against negative effects from overly aggressive bypassing.

Figure 4.20 shows the the performance effects of different access counter sizes. As the counter size is reduced, the counter is able to more quickly respond to variations in access behavior. However, smaller counters also reduce the resolution of access information. The results show that in most cases, 8-bit counters improve performance. However, occasionally 8-bit counters degrade performance significantly, such as in the case of *word*. For this reason, in order to reduce the hardware overhead of the MAT, 4-bit counters may be preferred.

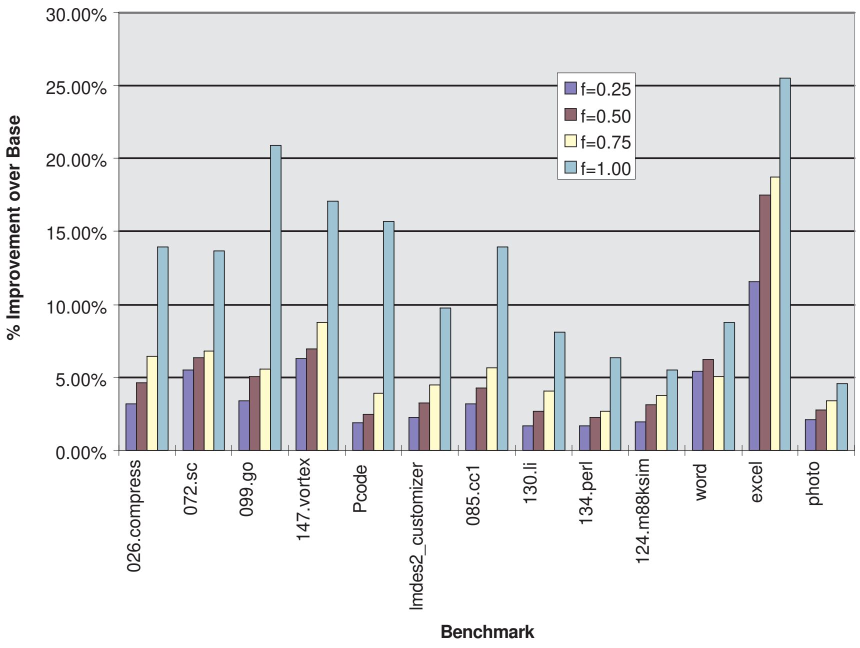


Figure 4.19 Performance improvements for different values of f.

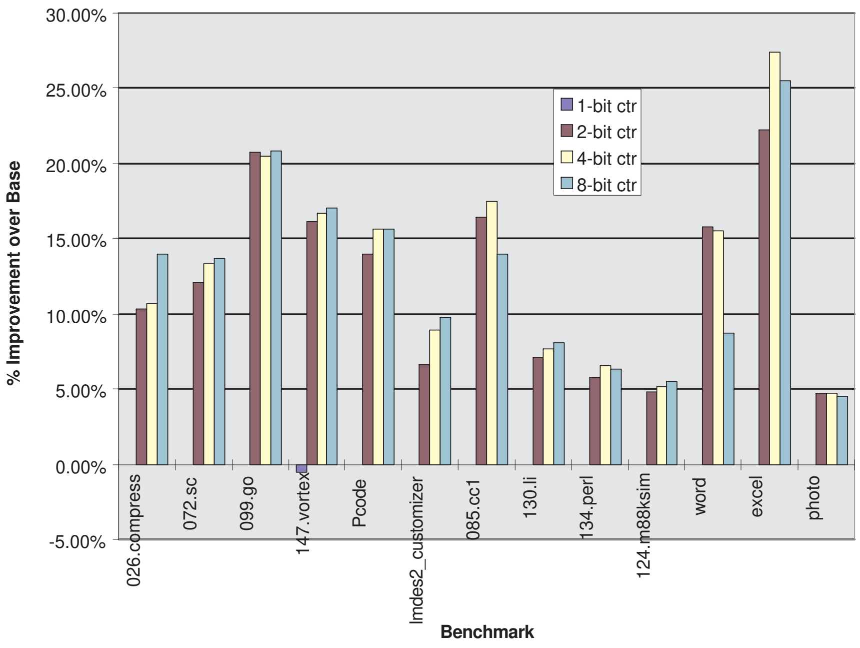


Figure 4.20 Performance improvements for different access counter sizes.

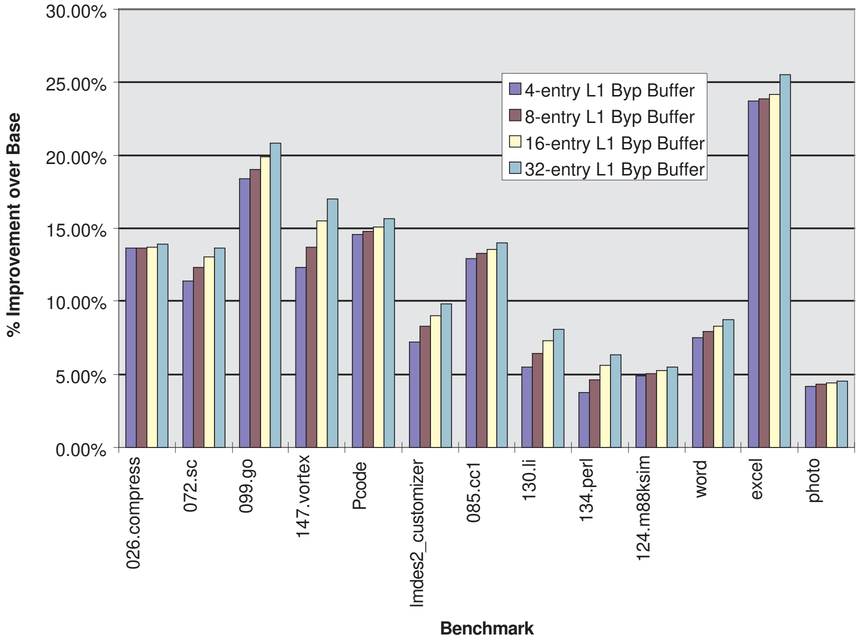
Figures 4.21(a) and (b) show the effects of varying the L1 bypass buffer size and associativity, respectively. Figures 4.22(a) and (b) show the same results for the L2 bypass buffer. Overall performance is not significantly reduced for smaller L1 bypass buffer sizes, even when only using a four-entry four-way set-associative L1 bypass buffer. The effect of reducing the L2 bypass buffer size can be much more significant. However, as the results in Section 4.3.2.2 will demonstrate, the majority of the performance improvement comes from performing bypassing at the L2 level, so the reductions in L1 performance are just as significant in terms of the effect on that level of the hierarchy. For both levels of the hierarchy, the associativity chosen for the default sizes is not as critical.

4.3.2.2 Bypassing performance improvement details

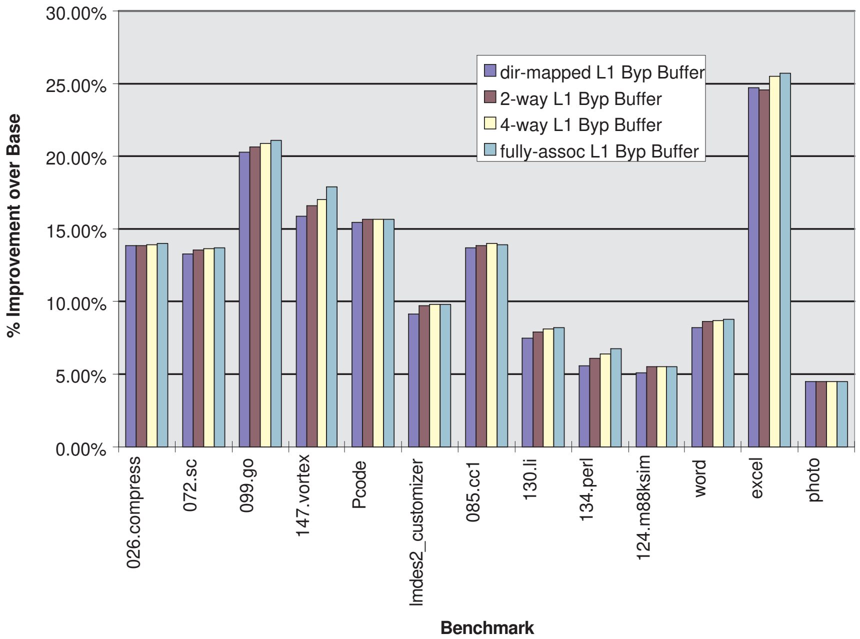
Figures 4.23(a) and (b) show the utilization of the L1-L2 bus and the system bus (connecting the L2 cache with the main memory), respectively, for the MAT scheme with an infinite number of entries. The reductions in bus traffic are due to the improvements in the hit ratio that will be demonstrated later in Section 4.3.2.5.

Figure 4.24(a) breaks the performance improvement into the portion achieved by performing bypassing at the L1 level alone, and that portion that is due to performing bypassing at the L2 level. The results show that while L1 bypassing does result in performance improvement, it is much more critical to perform bypassing at the L2 level. This makes sense given the large penalty incurred by L2 cache misses.

Nonblocking caches can result in different amounts of effective memory latency seen by the processor, depending on how many miss requests are outstanding when an access occurs. Therefore, the hit ratios may not be indicative of the overall performance. A more meaningful

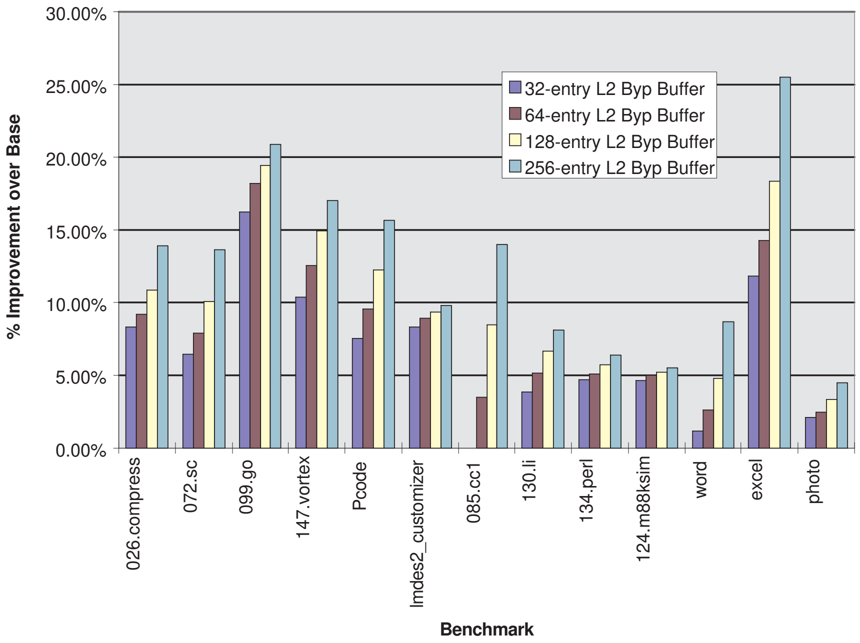


(a) varying L1 bypass buffer size

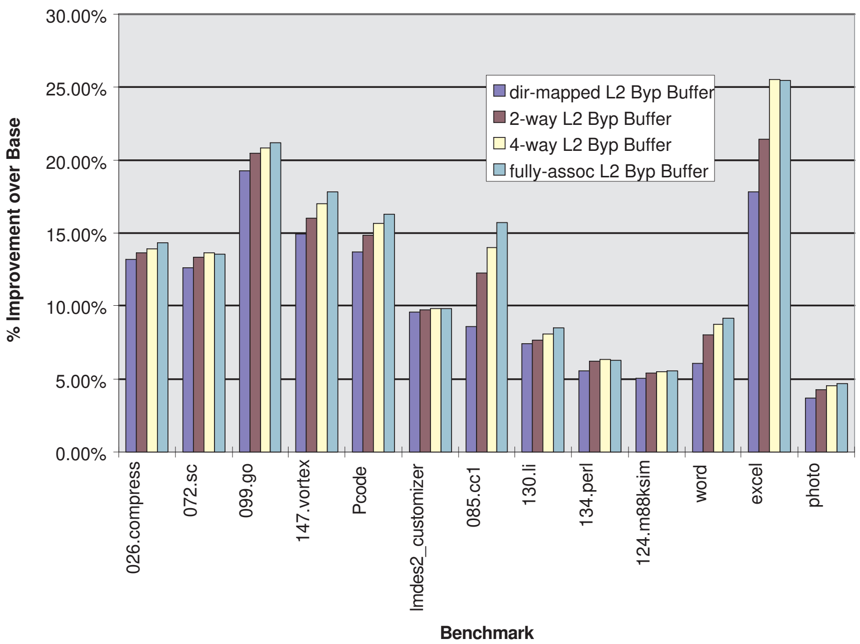


(b) varying L1 bypass buffer associativity

Figure 4.21 Performance improvements using different L1 bypass buffer sizes and associativities (the default is a 32-entry four-way set-associative L1 bypass buffer).

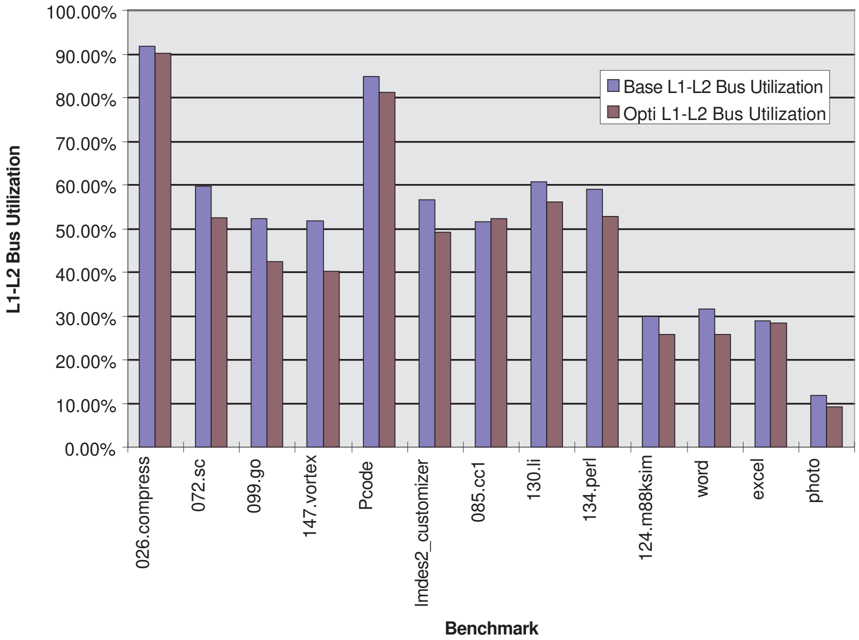


(a) varying L2 bypass buffer size

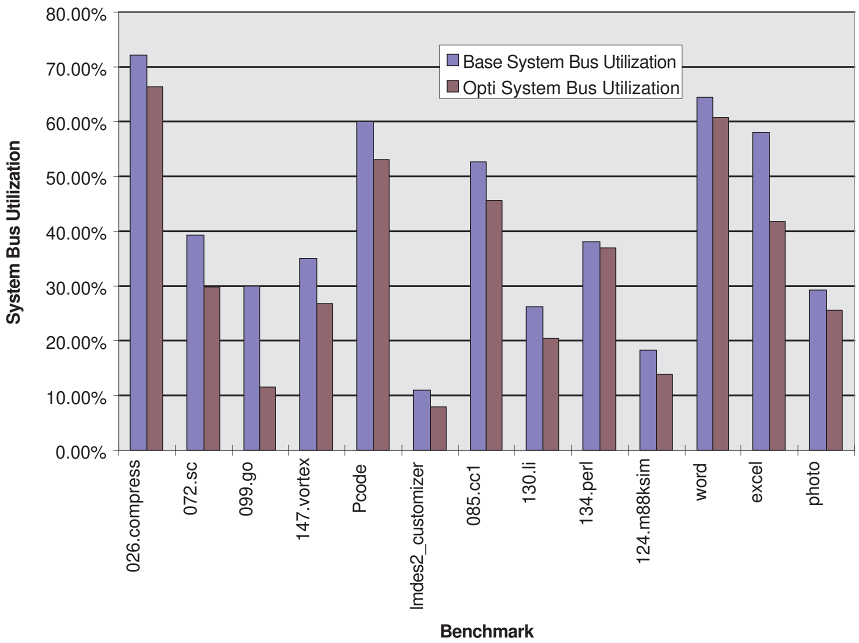


(b) varying L2 bypass buffer associativity

Figure 4.22 Performance improvements using different L2 bypass buffer sizes and associativities (the default is a 256-entry four-way set-associative L2 bypass buffer).

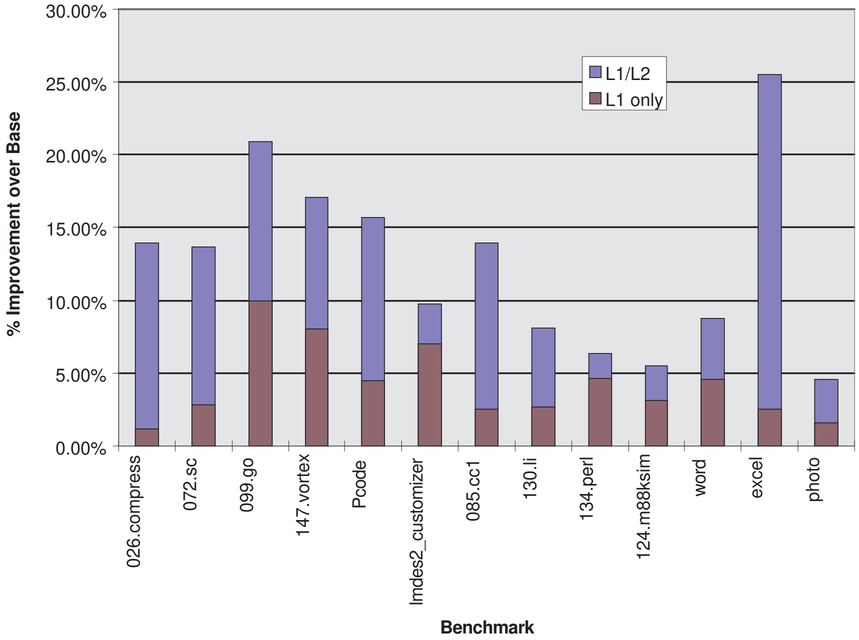


(a) L1-L2 bus utilization

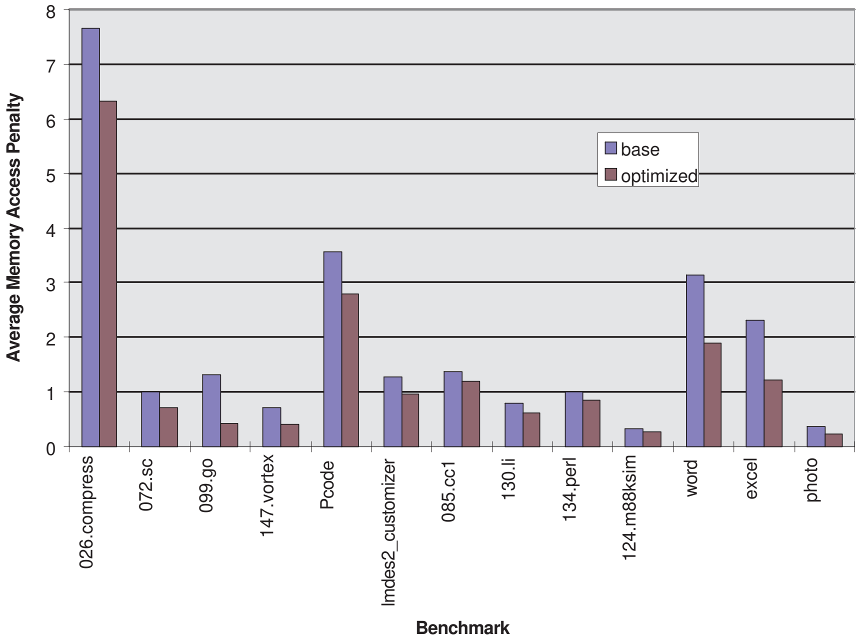


(b) system bus utilization

Figure 4.23 Bus utilization for the bypassing optimization (infinite-entry MAT).



(a) L1/L2 performance improvement breakdown



(b) average miss penalty per load access

Figure 4.24 L1/L2 performance breakdown and average miss penalty for the bypassing optimization (infinite-entry MAT).

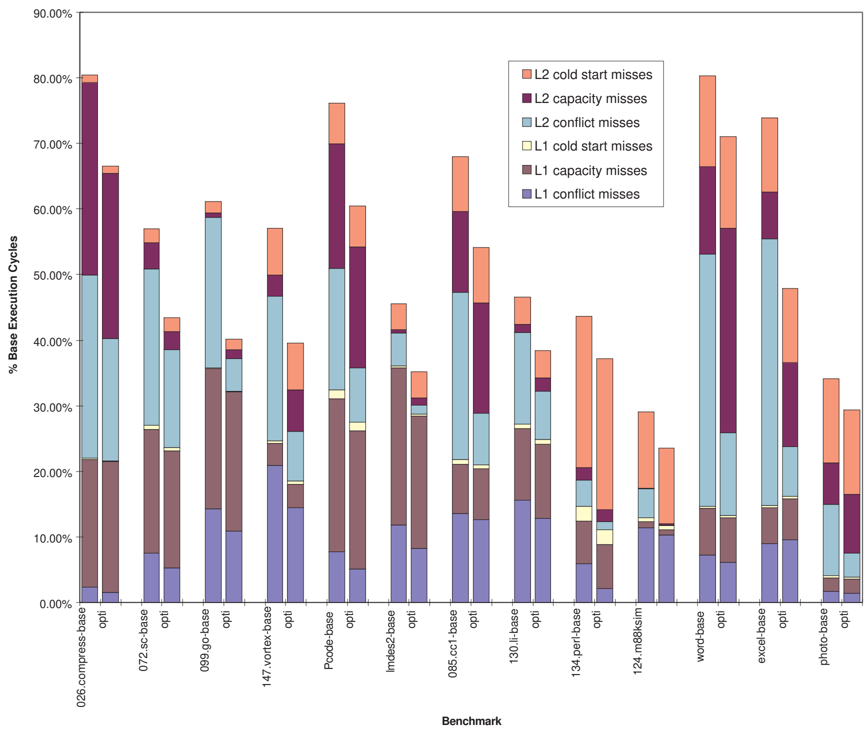


Figure 4.25 Breakdown of stall cycles incurred by each level of the memory hierarchy for both the base configuration and the MAT scheme (infinite-entry MAT).

metric is the *average miss penalty*, or the average number of cycles the processor is stalled on the data cache per load access. Figure 4.24(b) shows that the average miss penalties correspond with the speedup results shown in Figure 4.14.

Figure 4.25 shows the breakdown of processor execution cycles into the portions spent stalled waiting for cache misses to be serviced. The L1 portions include stall cycles from both the L1 instruction and data caches, and the L2 portions include stall cycles from both the instructions and data in the L2 unified cache. In general, the performance improvements at both cache levels come from reductions in conflict miss stall cycles.

Figure 4.26 illustrates that the uniform sampling methods described in Chapter 3, and utilized to reduce simulation time, do not greatly affect the resulting performance improvements. In fact, for all benchmarks except *026.compress* and *099.go*, the sampled results underestimate

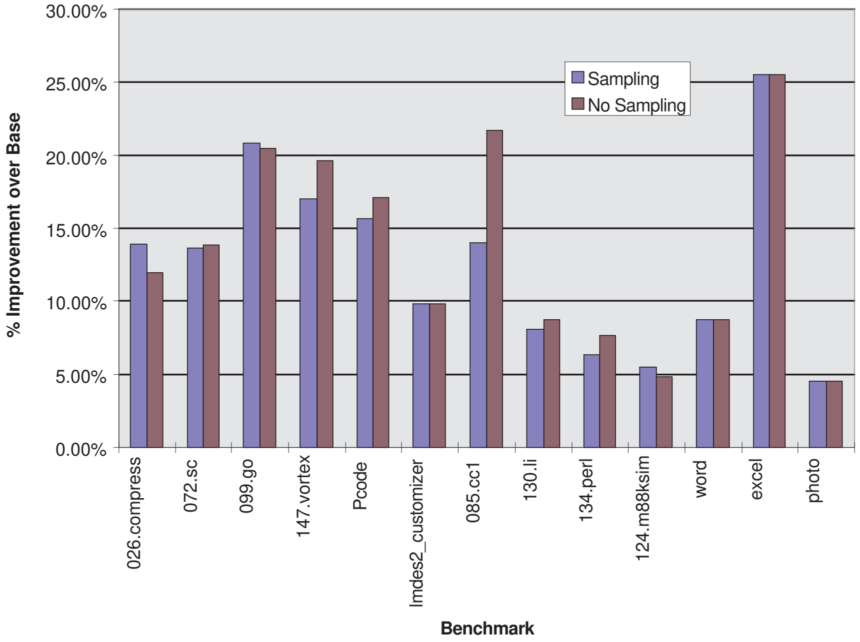
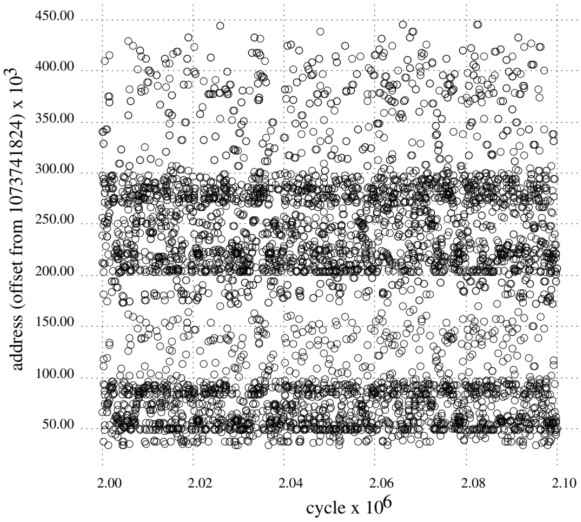


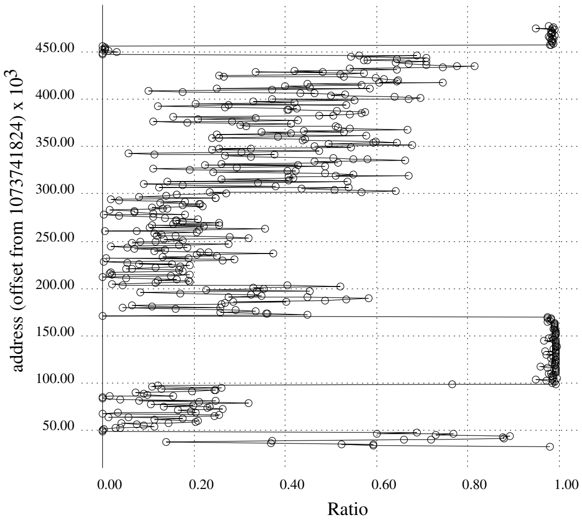
Figure 4.26 Comparison of performance improvements achieved using the sampling methodology described in Chapter 3 to that achieved when completely simulating the benchmarks.

the performance improvement obtained via cache bypassing. This makes sense due to the fact that the MAT counters accumulate statistics based on recent accessing behavior. When jumping to a new sample, these statistics are less likely to be correct, and may generate less optimal bypassing decisions. It is possible to take corrective measures on sample boundaries, such as clearing the MATs and the caches, and allowing them to "warm up" before collecting more performance statistics [44]. However, due to the closeness of the results shown in Figure 4.26, these measures did not seem critical.

Figure 4.27 suggests that the MAT is correctly deciding which data to bypass for *compress*. Figure 4.27(a) shows the memory access distribution for both htab and codetab, for the same execution phase shown in Figure 4.2. Figure 4.27(b) has the same y-axis and lines up with Figure 4.27(a). The x-axis of Figure 4.27(b) is the ratio of times that a missing access was



(a) memory access distribution



(b) bypassing ratio

Figure 4.27 Compress distribution and bypassing ratio comparison (MAT scheme, infinite-entry MAT).

instructed by the MAT to bypass the cache. Figure 4.27 shows that the very heavily accessed portions of the hash table bypass very rarely. The sparsely accessed portion of codetab, roughly between address offsets 100000 and 170000, bypasses almost 100% of the time.

4.3.2.3 Growing memory latency effects

As discussed in Chapter 1, memory latencies are increasing, and this trend is expected to continue. Figure 4.28 shows the speedups of the MAT scheme applied to direct-mapped data caches for three different latencies, the 100-cycle latency used in all previous configurations, as well as 200-cycle and 300-cycle latencies. Most of the benchmarks see much higher performance improvements from the MAT scheme when the memory latencies increase, with speedups over 35% for *099.go* and Excel for a 300-cycle memory latency. This is to be expected, as the number of cycles spent waiting for outstanding cache misses will increase, and optimizations which decrease the number of cache misses will become increasingly important.

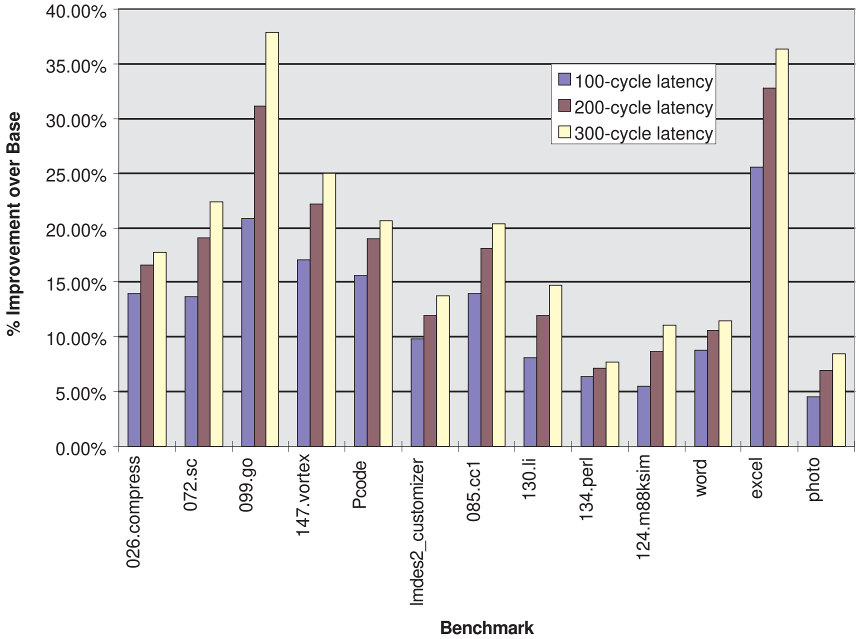


Figure 4.28 Performance improvement effects of longer memory latency.

4.3.2.4 Set-associative data caches

Increasing the set associativity of the data caches can reduce the number of conflict misses, which may in turn reduce the advantage offered by the MAT. To investigate these effects, the data cache configuration discussed in Chapter 3 was slightly modified to have a two-way set-associative L1 and L2 caches.

Figure 4.29 shows the performance improvements for the direct-mapped caches used earlier, as well as for two-way set-associative caches. As noted in Section 4.1.1, *compress* has very large hash tables, resulting in many capacity misses. For this reason, its performance improvement decreases little when increasing the cache associativity. However, the improvement due to bypassing is significantly reduced for many of the benchmarks, and in a few cases, bypassing degrades the performance. Because the higher associativity filters out some of the conflict misses that were bypassed in the case of direct-mapped caches, it may be beneficial to bypass

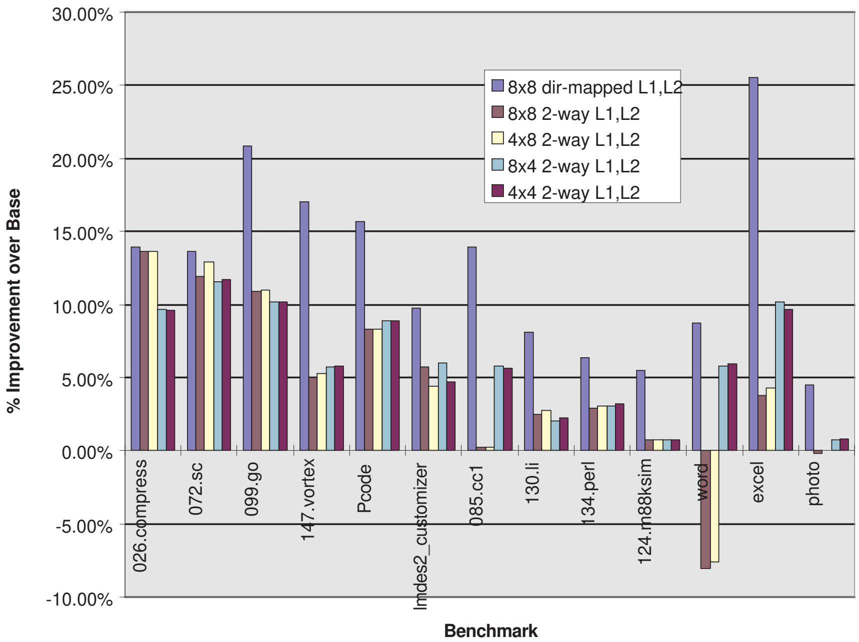


Figure 4.29 Performance improvement effects of higher cache associativity. The legend denotes (num L1 MAT counter bits)×(num L2 MAT counter bits).

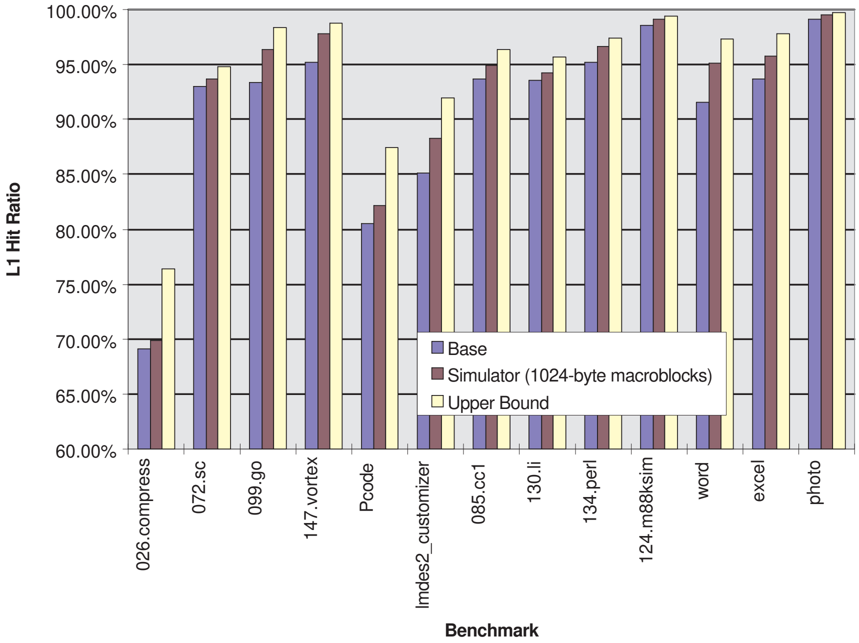
less aggressively. One way to bypass less frequently is to have fewer bits in the MAT access counter. Because the counters will saturate faster, and at a lower value, it is less likely that one macroblock will have a much larger *ctr* value for very long. For example, the previous configuration, containing 8-bit L1 and L2 access counters, was modified so that one or both of the MAT utilized 4-bit counters. These configurations are also shown in Figure 4.29. The results show that a smaller L2 MAT counter can be beneficial, particularly for *085.cc1* and Word, but a smaller L1 MAT counter does not usually have much effect. With increasing memory latency these speedups are likely to increase.

4.3.2.5 Comparison to upper bound

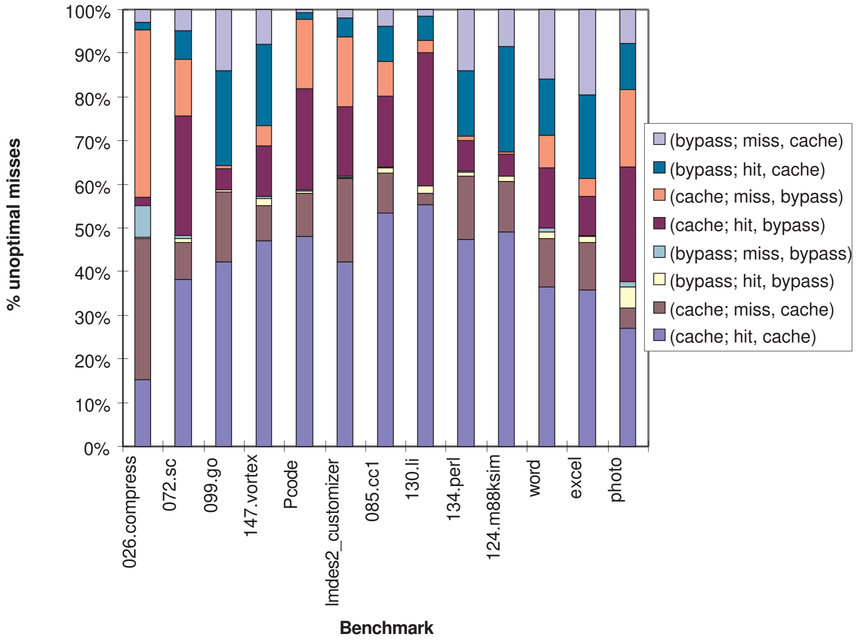
The L1 data cache read hit ratios for the same configuration, with 1-KB macroblocks and an infinite-entry MAT, are shown in Figure 4.30(a). Also shown for reference are the L1 hit ratios for the base configuration, as well as the upper-bound hit ratios reported in Table 4.2. In most cases the implementation described in Section 4.2 achieves roughly half of the upper-bound hit ratio improvement. A few benchmarks, most notably *compress* and *Pcode*, achieve very little of the upper-bound performance. These cases will be examined in greater detail below.

Figure 4.30(b) shows more information about the accesses R_2, which had reuse with earlier references R_1 in the upper-bound analysis, but which were displaced from the cache without reuse in the simulated implementation. In all cases R_2 missed in the simulator, but R_1 may have hit or missed. The bottom four sections of each bar represent those cases when the reference R_1 that should have been reused was placed in the same cache (main cache or bypass buffer) as in the upper-bound analysis. The reason for its displacement would be that either too much data was assigned to the same cache by the implementation, or that the upper-bound analysis was too optimistic. These cases make up around 50% of the misses examined.

For the remaining misses, the reference R_1, which had reuse in the upper-bound analysis was located in a different cache than that assigned by the upper-bound analysis. The graph shows that in many of these cases reference R_1 was located in the bypass buffer in the simulator, but not in the upper-bound analysis, and was later displaced from the bypass buffer before the reuse by R_2 could occur. In some of these cases R_1 missed and was bypassed, and in other cases R_1 hit, meaning that the decision to bypass the corresponding data was made by an earlier missing reference. In order to understand why the decision was made to bypass R_1 in the cases where it missed, the counter values used in the comparisons for several benchmarks were examined. The values of *ctr1* and *ctr2* from Equation (4.1) were examined for both the reference R_1 that was incorrectly bypassed, and the reference R_2 which later missed in the simulator (but hit in the upper bound analysis). Table 4.4 shows the average difference between $ctr2^{R_1}$ and $ctr1^{R_1}$,



(a) L1 data cache read hit ratio



(b) breakdown of unoptimal misses

Figure 4.30 L1 data cache read hit ratio comparison to upper bound, and breakdown of accesses that had reuse in the upper bound analysis but not in the simulated implementation. The legend for (b) denotes (R_1 upper bound cache/bypass; R_1 simulator hit/miss, R_1 simulator cache/bypass).

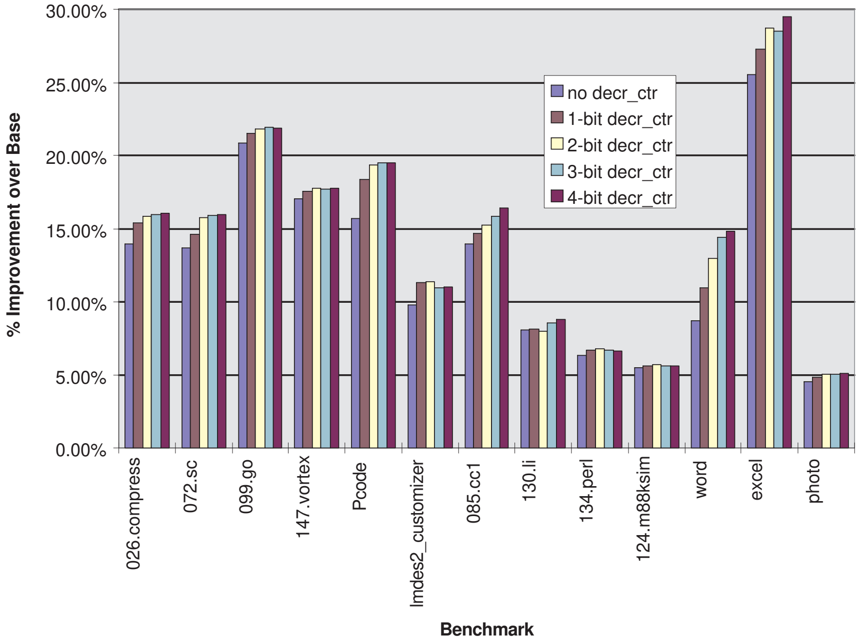
the average increase in *ctr1* between R_1 and R_2, and the average increase in *ctr2* between R_1 and R_2. Note that $ctr1^{R_1}$ and $ctr1^{R_2}$ must be the same physical counter, since R_1 and R_2 reference the same cache block. However $ctr2^{R_1}$ and $ctr2^{R_2}$ may not be the same physical counter, depending on the accesses and cache replacements made between R_1 and R_2.

Table 4.4 shows that the average difference between the counter values used in the decision to bypass R_1 can be large. Also, because *ctr2* decreases between the two accesses for most benchmarks, the corresponding macroblock is not being accessed as heavily as the macroblock containing the accessed data. However, the average *ctr1* increases and *ctr2* decreases often do not come close to covering the gap between the two counters. This suggests that the counter values are taking too long to converge. There are several possible changes that could increase the speed at which the two counters converge. One is to simply reduce the number of bits in the counters. However, Figure 4.20 showed that this tended to reduce performance. Another is to modify the rate at which *ctr2* is reduced. It is possible that the heuristic used to decrement the access counters could be improved over simply decrementing by one on every conflict. For example, another small counter could be added to each entry, that detects when cached data from the macroblock is no longer being actively accessed. The saturating counter, called *decr_ctr*, could be incremented by one on every conflict for a cache location held by its macroblock (i.e., whenever its access counter is decremented), and cleared to zero whenever its macroblock is accessed (i.e., whenever its access counter is incremented). In other words, the value of *decr_ctr* is the number of conflicting accesses to other macroblocks that have occurred since the last access to this macroblock. Then, the access counter could be decremented by one plus the value of *decr_ctr* on a conflicting access to another macroblock, rather than by one, effectively enabling access counters of macroblocks no longer being accessed to decrease faster.

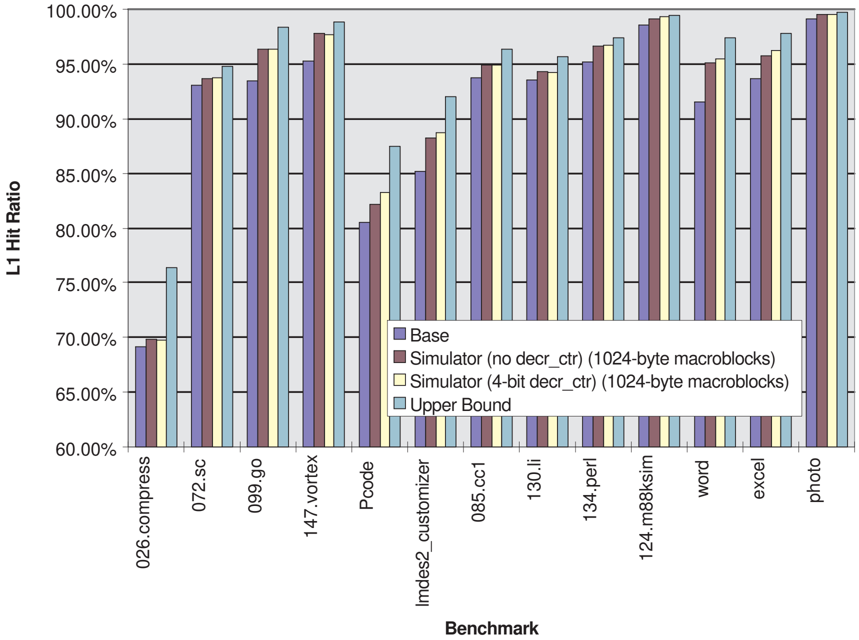
Table 4.4 Average difference between counter values for references which were not bypassed and had reuse in the upper-bound analysis, but which were bypassed and did not have reuse in the simulated implementation. R_1 is the reference which missed and should not have bypassed, and R_2 is the reference which should have reused R_1 but did not.

Benchmark	Ave($ctr2^{R_1}$-$ctr1^{R_1}$)	Ave($ctr1^{R_2}$-$ctr1^{R_1}$)	Ave($ctr2^{R_2}$-$ctr2^{R_1}$)
026.compress	10.04	2.08	-1.68
072.sc	27.72	18.94	-8.05
099.go	13.19	6.88	-0.70
147.vortex	37.05	3.68	-0.65
Pcode	6.44	2.75	-3.78
lmdes2_customizer	12.41	4.29	-1.71
085.cc1	15.96	8.44	-2.49
130.li	22.16	17.20	-9.12
134.perl	50.48	15.60	-4.95
124.m88ksim	83.64	49.59	1.84
Word	89.56	5.43	-0.02
Excel	83.16	4.60	0.21
Photo	117.89	7.74	1.47

Though smaller access counters produce quicker response times in the counter values, such counters also reduce the resolution of the comparisons. On the other hand, the variable counter decrement scheme, using a *decr_ctr*, increased performance in almost all cases. Figure 4.31(a) shows the performance improvements using 1-KB macroblocks and infinite-entry MATs for 1- to 4-bit *decr_ctr* sizes in both the L1 and L2 MATs. These results show that accelerating the access counter decrements for less-actively-accessed macroblocks can significantly improve performance. Figure 4.31(b) shows the same hit ratios shown in Figure 4.30(a), with the hit ratios using a 4-bit *decr_ctr* added. Surprisingly, the hit ratios have not increased as much as the performance improvements in Figure 4.31(a) would suggest, implying that even small improvements in the hit ratios can translate into good performance improvements. The hit ratio for *026.compress* actually decreases, even though the execution cycles are reduced. In this case, the new stream of references sent to the L2 cache have improved locality at the L2 level, and result in fewer L2 misses (even when the L2 cache does not use a *decr_ctr* and therefore does not change).



(a) performance improvement



(b) L1 data cache read hit ratio

Figure 4.31 Performance improvements and cache hit ratios using variable access counter decrements.

CHAPTER 5

SPATIAL LOCALITY OPTIMIZATIONS

As discussed in Chapter 2, exploitation of spatial locality existing in data can reduce both the number of cache misses and the cache miss penalty. Traditionally, exploitation of spatial locality is achieved either by using larger cache block sizes or by prefetching additional nearby cache blocks.

Several studies have examined the performance effects of cache block sizes [50], [51]. One of the studies allowed multiple consecutive blocks to be fetched with one request [50] and found that for data caches the optimal statically determined fetch size was generally twice the block size. This dissertation also examines fetch sizes larger than the block size; however, the fetch size is allowed to vary based on the detected spatial locality. Dahlgren et al. presented a method that allows the number of blocks fetched on a miss to vary across program execution, but not across different data [15]. That work was geared towards numeric codes running on multiprocessors. Other work that examined variable line sizes allowed either a single small block, or a larger virtual line size containing multiple cache blocks to be fetched on a miss, depending on a spatial tag attached to the memory reference. These tags were set by the compiler, using static analysis which examined array reference subscripts within loops in numeric codes. Adjusting the block sizes to reduce false sharing and coherence traffic on multiprocessors has been studied by Dubnicki and LeBlanc [52]. In that method, the focus was not on improving cache utilization for data with little temporal locality; therefore, subblocking was used to support the different block sizes, which could result in significant amounts of wasted cache capacity. The run-time detection mechanism determined when multiple processors were simultaneously accessing different halfs of the current blocksize, and halved the blocksize in that case. Smaller adjacent blocks of equal size could also be merged into a larger block when processors were detected accessing both smaller blocks. Results showed that average latency and traffic were reduced over fixed block sizes for a number of numeric codes. Another work examined a variable line size cache, whereby a line size table indicates how many blocks should be fetched on a miss to different regions in memory [53]. The table entries must be explicitly modified by instructions; however, no compiler methods for determining the optimal line size or setting up the table entries were presented.

Stream buffers enable aggressive sequential prefetching without polluting the cache [9], [54], [55]. However, the number of unique reference streams which can be prefetched is limited by the number of stream buffers present in the hardware. Improved stream buffer designs have been proposed by Palacharla and Kessler [56], both to allow prefetching of non-unit strides via stride-detecting finite state machines, as well as to avoid wasting the limited stream buffer resources and reduce unnecessary prefetching via filters that detect nonspatial misses. Similarly, miss-history driven stream buffer prefetching [57] utilizes a history buffer to detect forward streaming, backward streaming, and prefetching strides. However, these methods best detect patterns where large arrays are being accessed with regular strides, and therefore tend to work best on numeric codes.

Other prefetching methods were overviewed in Chapter 1.

The dual data cache [37] attempts to intelligently exploit both spatial and temporal locality, however the temporal and spatial data must be placed in separate structures, and therefore the relative amounts of each type of data must be determined a priori, when choosing the dual cache sizes. Also, the spatial locality detection method was tuned to numeric codes with constant stride vectors. In integer codes, the spatial locality patterns may not be as regular. The split temporal/spatial (STS) cache [58] is similar in structure to the dual data cache; however, the run-time locality detection mechanism is quite different from that of both the dual data cache and this dissertation. In the STS cache, spatial cache blocks can be retagged as temporal if the block accesses are predominantly to one half of the cache block. However, no mechanism for retagging temporal data as spatial was introduced. Also, when data is retagged, the new spatial or temporal marking must be saved with the instruction.

5.1 Spatial Locality Analysis

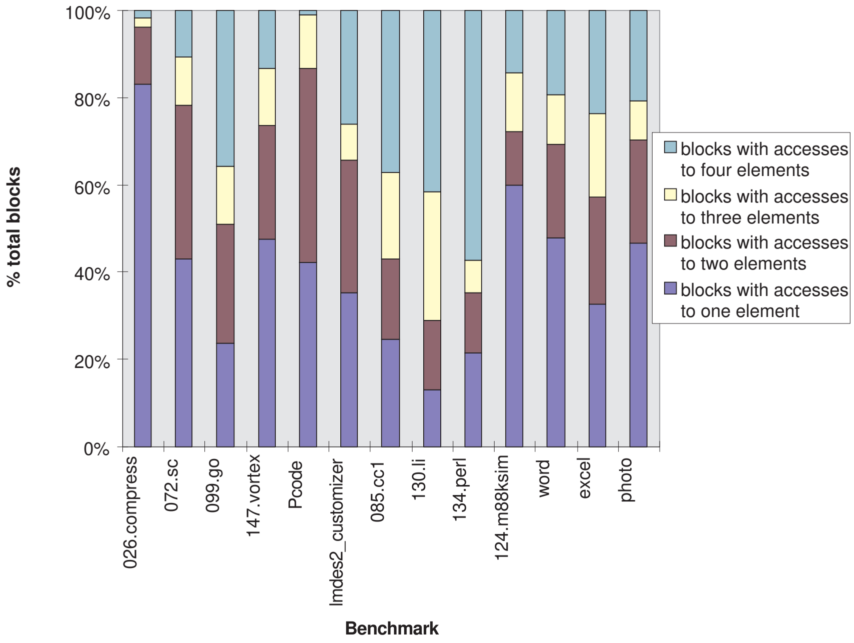
The following terms are defined as they will be used throughout this chapter:

- Element: A data item of the maximum size allowed by the ISA, which is 8 bytes in the system assumed in this dissertation.
- Temporal reuse: A reference to a cached element which was fetched into the cache as a result of an earlier reference to that same element.
- Spatial reuse: A reference to a cached element other than the element which caused the referenced element to be fetched into the cache.

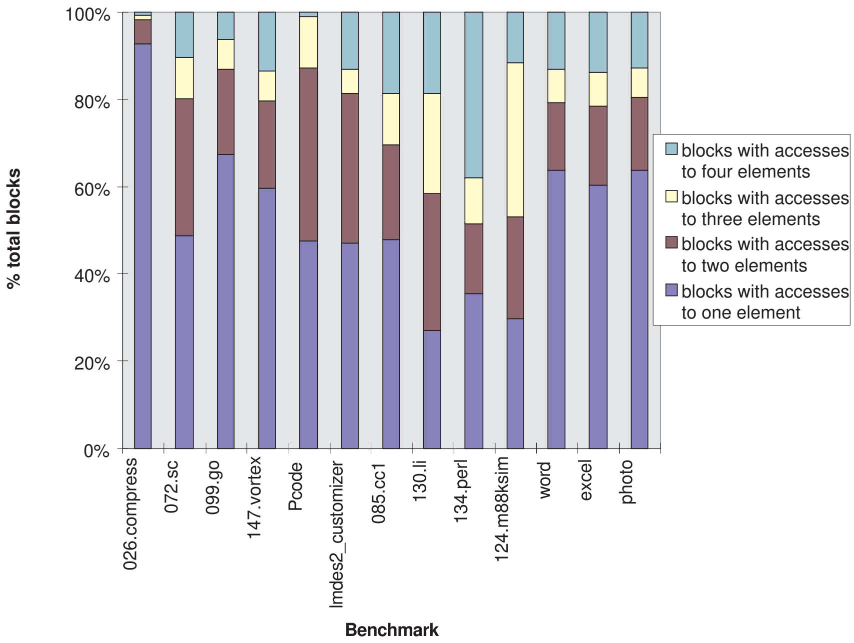
The spatial locality in an application's data set can predict the effectiveness of spatial locality optimizations. Unfortunately, no quantitative measure of spatial locality exists, and this dissertation is forced to adopt indirect measures. One indirect measure of the amount of spatial locality is via its inverse relationship to the distance between references in both space and time. With this in view, the spatial reuses were measured in a 64-KB fully associative cache with 32-byte lines. This provides an approximate time bound (the time taken for a block to be displaced), and a space bound (within 32-byte block boundaries). This block size was chosen because past studies have found that 16 or 32-byte block sizes maximize data cache performance [50]. These measurement techniques differ from those in [59], which explicitly measure the reuse distance (in time). The goal of this dissertation is to measure both the reused and unused portions of the cache blocks, for different cache organizations.

Figure 5.1(a) shows the spatial locality estimates for the fully associative cache. The number of dynamic cache blocks is broken down by the number of 8-byte elements that were accessed during each block's cache lifetime. Blocks where only one element is accessed have no spatial locality within the measured context. This graph does not show the relative locations of the accessed elements within each 32-byte cache block. Figure 5.1(a) shows that 13-83% of the cached blocks have no spatial reuse. Figure 5.1(b) shows how this distribution changes for a 16-KB direct-mapped cache. In this case 30-93% of the blocks have no spatial reuse.

For a 32-byte cache block, over half the time the extra data fetched into the cache simply wastes bus bandwidth and cache space. Similar observations have been made for numeric codes [59]. Therefore, it would be beneficial to tune the block size to the spatial locality available in the data. This optimization is investigated in this chapter. Several issues involved with varying fetch sizes are discussed, including cost efficient and accurate spatial locality detection, fetch size choice, and cache support for varying fetch sizes.



(a) 64-KB fully associative cache



(b) 16-KB direct mapped cache

Figure 5.1 Breakdown of blocks cached in L1 data cache by how many 8-byte elements were accessed while each block was cached. The results for two cache configurations are shown, each with 32-byte blocks.

```
/* RTL expression ("rtx"). */
typedef struct rtx_def
{
  /* The kind of expression this is. */
  enum rtx_code code : 16;
  /* The kind of value the expression has. */
  enum machine_mode mode : 8;
  /* Various bit flags */
  unsigned int jump : 1;
  unsigned int call : 1;
  unsigned int unchanging : 1;
  unsigned int volatil : 1;
  unsigned int in_struct : 1;
  unsigned int used : 1;
  unsigned integrated : 1;
  /* The first element of the operands of
     this rtx. The number of operands and
     their types are controlled by the 'code'
     field, according to rtl.def. */
  rtunion fld[1];
} *rtx;
```

```
/* Common union for an element
   of an rtx. */
typedef union rtunion_def
{
  int rtint;
  char *rtstr;
  struct rtx_def *rtx;
  struct rtvec_def *rtvec;
  enum machine_mode rttype;
} rtunion;
```

Figure 5.2 Gcc *rtx* definition.

5.1.1 Code example

In this section a code example from SPEC92 *085.gcc* is used to illustrate the difficulties involved with static analysis and annotation of spatial locality information, motivating the dynamic approach proposed in this dissertation.

One of the main data structures used in *gcc* is a register transfer language (RTL) expression, or *rtx*, whose definition is shown in Figure 5.2. Each *rtx* structure contains a 2-byte code field, a 1-byte mode field, seven 1-bit flags, and an array of operand fields. The operand array, *fld*, is defined to contain only one 4-byte element; however, each *rtx* is dynamically allocated to contain as many array elements as there are operands, depending on the *rtx* code, or RTL expression type. Therefore, each *rtx* instance contains 8 or more bytes.

In the frequently executed `_rtx_renumbered_equal_p` routine, which is used during jump optimization, two *rtx* structures are compared to determine if they are equivalent. Figure 5.3 shows a slightly abbreviated version of the `_rtx_renumbered_equal_p` routine. After checking if

```
int rtx_renumbered_equal_p (rtx x, rtx y)
{
  register int i;
  register RTX_CODE code = GET_CODE (x);
  register char *fmt;
  if (x == y) return 1;
  if ((code == REG || (code == SUBREG && GET_CODE (SUBREG_REG (x)) == REG))
      && (GET_CODE (y) == REG || (GET_CODE (y) == SUBREG &&
      GET_CODE (SUBREG_REG (y)) == REG)))
      {   ... /* Rarely entered */ ...    }
  if (code != GET_CODE (y)) return 0;                    Exits here
                                                         448 times
  switch (code) {
    case PC: case CC0: case ADDR_VEC: case ADDR_DIFF_VEC:
      return 0;
    case CONST_INT:
      return x->fld[0].rtint == y->fld[0].rtint;         Exits here
                                                         29096 times
    case LABEL_REF:
      return (next_real_insn(x->fld[0].rtx) == next_real_insn(y->fld[0].rtx));
    case SYMBOL_REF:
      return x->fld[0].rtstr == y->fld[0].rtstr;
  }
  if (GET_MODE (x) != GET_MODE (y)) return 0;
  /* Compare the elements. If any pair of corresponding
   * fail to match, return 0 for the whole thing. */
  fmt = GET_RTX_FORMAT (code);
  for (i = GET_RTX_LENGTH (code) - 1; i >= 0; i--) {
      register int j;
      switch (fmt[i]) {
        case 'i':
            if (x->fld[i].rtint != y->fld[i].rtint) return 0;
            break;
        case 's':
            if (strcmp(x->fld[i].rtstr, y->fld[i].rtstrs)) return 0;
            break;                                       Case matches
        case 'e':                                        33060 times
          if (!rtx_renumbered_equal_p(x->fld[i].rtx, y->fld[i].rtx))
            return 0;                                    Exits here
                                                         30014 times
            break;
        case 'E':
            ... /* Accesses *({x,y}->fld[i].rtvec) */ ...
            break;
      }
  }
  return 1;
}
```

Figure 5.3 Gcc `_rtx_renumbered_equal_p` routine, executed 63 173 times.

the code and mode fields of the two *rtx* structures are identical, the routine then compares the operands, to determine if they are also identical. Four branch targets in Figure 5.3 are annotated with their execution weights, derived from execution profiles using the SPEC reference input. Roughly 1% of the time only the code fields of the two *rtx* structures are compared before exiting. In this case, only the first two bytes in each *rtx* structure is accessed. About 46% of the time x and y are `CONST_INT` *rtx*, and only the first operand is accessed. Therefore, only the first eight bytes of each *rtx* structure is accessed, and there is spatial locality within those eight bytes.

For many other types of RTL expressions, the routine will use the `for` loop to iterate through the operands, from last to first, comparing them until a mismatch is found. In this case there will be spatial locality, but at a slightly larger distance (in space) than in the previous case. Most instruction types contain more than one operand. The most common operand type in this loop is an RTL expression, which results in a recursive call to `_rtx_renumbered_equal_p`.

This routine illustrates that the amount of spatial locality can vary for particular load references, depending on the function arguments. Therefore, if the original access into each *rtx* structure in this routine is a miss, the optimal amount of data to fetch into the cache will vary correspondingly. For example, if the access `GET_CODE(y)` on line 11 of Figure 5.3, which performs the access `y->code`, misses in the L1 cache, the spatial locality in that data depends on whether the program will later fall into a *case* body of the *switch* statement on line 12 or into the body of the `for` loop on line 26, and on the *rtx* type of x which determines the initial value of i in the `for` loop. However, at the time of the cache miss on line 11 this information is not available, as it is highly data-dependent. As such, neither static analysis (if even possible) nor profiling will result in definitive or accurate spatial locality information for the load instructions. Dynamic analysis of the spatial locality in the data offers greater promise. For this routine, dynamic analysis of each *rtx* instance accessed in the routine would obtain the most accurate spatial locality detection. Also, dynamic schemes do not require profiling, which many users are unwilling to perform, or ISA changes.

This code example will be revisited in Section 5.3.2.1, and the accuracy of the fetch sizes chosen by the optimization presented in this chapter is examined for the access `y->code`.

5.1.2 Applications

Aside from varying the data cache load fetch sizes, the spatial locality optimizations described in this chapter could be used to control instruction cache fetch sizes, write allocate versus no-allocate policies, and bypass fetch sizes when bypassing is employed. In this chapter, the application of these techniques to control the fetch sizes into the L1 data cache and the L2 unified cache is examined. These optimizations are also studied in conjunction with cache bypassing, in Section 5.4.1.

5.2 Techniques

5.2.1 Support for varying fetch sizes

The varying fetch size optimization could be supported using subblocks. In that case the block size is the largest fetch size and the subblock size is $gcd(fetch\_size_0, ..., fetch\_size_n)$, where n is the number of fetch sizes supported. Currently, only two power-of-two fetch sizes are supported for each level of cache, so the subblock size is simply the smaller fetch size. However, the cache lines will be underutilized when only the smaller size is fetched.

Instead, this dissertation uses a cache with small lines, equal to the smaller fetch size, and optionally fills in multiple, consecutive blocks when the larger fetch size is chosen. This approach is similar to that used in some prefetching strategies [60]. As a result, the cache can be fully utilized, even when the smaller sizes are fetched. It also eliminates conflict misses resulting from accesses to different subblocks. However, this approach makes detection of spatial reuses much more difficult, as will be described in Section 5.2.2. Also, smaller block sizes increase the tag array cost, which is addressed in Section 5.4.2. In this scheme, the *max_fetch_size* data

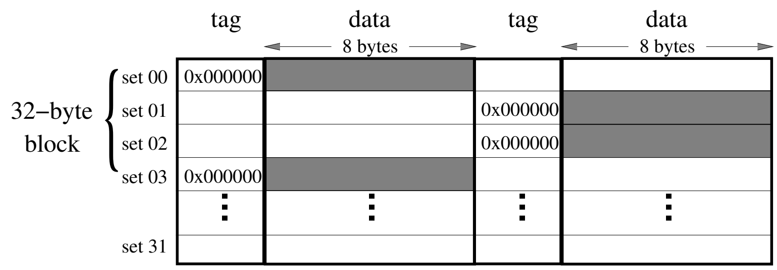


Figure 5.4 Layout of 8-byte subblocks from the 32-byte block starting at address 0x00000000 in a 512-byte two-way set-associative cache with 8-byte lines. The shaded blocks correspond to the locations of the four 8-byte subblocks.

is always aligned to *max_fetch_size* boundaries. As a result, these techniques will fetch data on either side of the accessed element, depending on the location of the element within the *max_fetch_size* block. Experiments have shown that spatial locality in the data cache can be in either direction (spatially) from the referenced element.

5.2.2 Spatial locality detection table

To facilitate spatial locality monitoring, a *spatial counter*, or *sctr*, is included in each MAT entry. The role of the *sctr* is to monitor the medium to long-term spatial locality of the corresponding macroblock, and to make fetch size decisions, as will be explained in Section 5.2.3. This counter will be incremented whenever a *spatial miss* is detected, which occurs when portions of the same larger fetch size block of data reside in the cache, but not the element currently being accessed. Therefore, a hit might have occurred if the larger fetch size was fetched, rather than the smaller fetch size. When using small cache block sizes, where multiple cache blocks are filled when the larger fetch size is chosen, a spatial miss is not trivial to detect. If the cache is not fully associative, the tags for different blocks residing in the same larger fetch size block will lie in consecutive sets, as shown in Figure 5.4, where the data in one 32-byte block is highlighted. Searching for other cache blocks in the same larger fetch size block of data will require access to the tags in these consecutive sets, and thus either additional cycles to access, or additional hardware support. One possibility is a restructured tag array design allowing efficient access to multiple consecutive sets of tags. Alternatively, a separate structure can be used to detect this information, which is the approach investigated in this dissertation.

This structure is called the *spatial locality detection table* (*SLDT*) and is designed for efficient detection of spatial reuses with low hardware overhead. The role of the SLDT is to detect spatial locality of data while it is in the cache, for recording in the MAT when the data is displaced. The SLDT is basically a tag array for blocks of the larger fetch size, allowing single-cycle access to the necessary information. Figure 5.5 shows an overview of how the SLDT interacts with the MAT and L1 data cache, where the double-arrow line shows the correspondence of four L1 data cache entries with a single SLDT entry. In order to monitor all cache blocks, the SLDT would need N entries, where N is the number of blocks in the cache. This represents the worst case of having fetched only smaller (line) size blocks into the cache, all from different larger size blocks. However, in order to reduce the hardware overhead of the SLDT, a much smaller number of entries is used, which also will help filter out many of the longer-term spatial reuses since the SLDT entries are displaced much more often. The same SLDT could be used to monitor the spatial locality aspects of all structures at the same level in the memory hierarchy, such as the data cache, the instruction cache, and, when performing bypassing, the bypass buffer.

The SLDT tags correspond to maximum fetch size blocks. The *sz* field is one bit that indicates if the larger size block was fetched into the cache or if only smaller blocks were fetched. The *vc* (*valid count*) field is $\log(max\_fetch\_size/min\_fetch\_size)$ bits in length, and indicates how many of the smaller blocks in the larger size block are currently valid in the data cache. The actual number of valid smaller blocks is $vc + 1$. An SLDT entry will only be

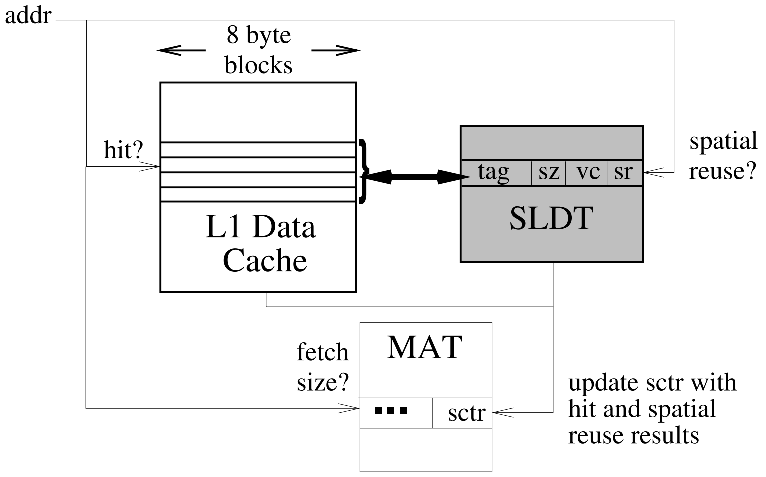


Figure 5.5 SLDT and MAT hardware.

valid for a larger size block when some of its constituent blocks are currently valid in the data cache. A bit mask could be used to implement the *vc*, rather than the counter design, to reduce the operational complexity. However, for large maximum-to-minimum fetch-size ratios, a bit mask will result in larger entries. Finally, the *sr* (*spatial reuse*) bit will be set if spatial reuse is detected, as will be discussed later.

When a larger size block of data is fetched into the cache, an SLDT entry is allocated (possibly causing the replacement of an existing entry) and the values of *sz* and *vc* are set to 1 and $max\_fetch\_size/min\_fetch\_size - 1$, respectively. If a smaller size block is fetched, and no SLDT entry currently exists for the corresponding larger size block, then an entry is allocated and *sz* and *vc* are both initialized to 0. If an entry already exists, *vc* is incremented to indicate that there is now an additional valid constituent block in the data cache. For both fetch sizes, the *sr* bit is initialized to 0. When a cache block is replaced from the data cache, the corresponding SLDT entry is accessed, and its *vc* value is decremented if it is greater than 0. If *vc* is already 0, then this was the only valid block, so the SLDT entry is invalidated. When an SLDT entry is invalidated its *sr* bit is checked to see if there was any spatial reuse while the data was cached. If not, the corresponding entry in the MAT is accessed, and its *sctr* is decremented, effectively depositing the information in the MAT for longer-term monitoring. Because the SLDT is managed as a cache, entries can be replaced, in which case the same actions are taken.

An *fi* (*fetch initiator*) bit is added to each data cache tag to help detect spatial hits. The *fi* bit is set to 1 during the cache refill for the cache block containing the referenced element (i.e., the cache block causing the fetch), otherwise it is reset to 0. Therefore, a hit to any block with a *fi* bit of 0 is a spatial hit, as this data was fetched into the cache by a miss to some other element.

Table 5.1 summarizes the actions taken by the SLDT for memory accesses. The *sr* bit, which was initialized to zero, is set for all types of both spatial misses and spatial hits. Two types of spatial misses are detected. The first type of spatial miss occurs when other portions of the same larger fetch size block were fetched independently, indicated by a valid SLDT entry with a *sz* of 0. Therefore, there might have been a cache hit if the larger size block was fetched, so the corresponding entry in the MAT is accessed, and its *sctr* is incremented. The second type can occur when the larger size block was fetched, but one of its constituent blocks was displaced from the cache, as indicated by a cache miss and a valid SLDT entry with a *sz* of 1. It is not trivial to detect if this miss is to the element which caused the original fetch, or to some other element in the larger fetch size block. The *sr* bit is conservatively set, but the *sctr* in the corresponding MAT entry is not incremented.

A spatial hit can occur in two situations. If the larger size block was fetched, then the *fi* bit will only be set for one of the loaded cache blocks. A hit to any of the loaded cache blocks without the *fi* bit set is a spatial hit, as described earlier. The *sctr* is not incremented on spatial

Table 5.1 SLDT actions. A dash indicates that there is no corresponding value, and a blank indicates that the value does not matter.

Cache Access	SLDT Access	*fi*	*sz*	*vc*	Action
Miss	Hit	-	0		$sr = 1$; $sctr++$
		-	1		$sr = 1$
Hit	Hit	0			$sr = 1$
			0	>0	$sr = 1$
Hit	Miss	0	-	-	alloc SLDT entry; $sz = 1$; $sr = 1$
		1	-	-	alloc SLDT entry
Cache entry replaced			$vc > 0$		$vc--$
			$vc == 0$		invalidate SLDT entry
SLDT entry replaced or invalidated			$sr == 0$		$sctr--$
			$sr == 1$		no action

hits, because the fetch size was correct. The *sctr* is only updated when the fetch size should be changed in the future. When multiple smaller blocks were fetched, a hit to one of these is also characterized as a spatial hit. This case is detected by checking if *vc* is larger than 0 when *sz* is 0. However, the *sctr* is not incremented in this case either because a spatial miss would have been detected earlier when a second element in the larger fetch size block was first accessed (and missed).

5.2.3 Fetch size decisions

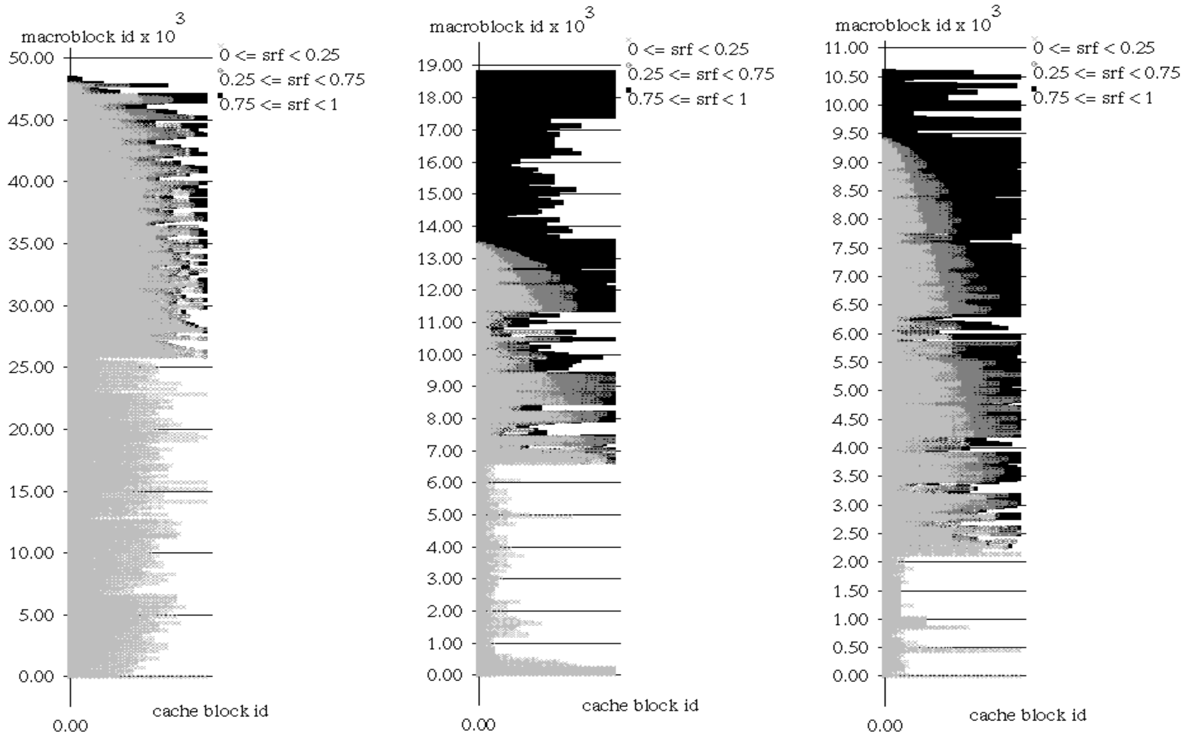
On a memory access, a lookup in the MAT of the corresponding macroblock entry is performed in parallel with the data cache access. If an entry is found, the *sctr* value is compared to some threshold value. The larger size is fetched if the *sctr* is larger than the threshold, otherwise the smaller size is fetched. If no entry is found, a new entry is allocated, and the *sctr* value is initialized to the threshold value, and the larger fetch size is chosen. In this dissertation, the threshold is 50% of the maximum *sctr* value.

5.3 Experimental Evaluation

5.3.1 Macroblock spatial locality variations

Before presenting the performance improvements achieved by this optimization, the accuracy of the macroblock granularity for monitoring spatial locality is first explored. It is important to have accurate spatial locality information in the MAT for this scheme to be successful. This means that all data elements in a macroblock should have similar amounts of spatial locality at each phase of program execution.

After dividing main memory into macroblocks, as described in Section 4.2.1, the macroblocks can be further subdivided into smaller sections, each the size of a 32-byte cache block. These smaller sections will simply be called *blocks*. In order to determine the dynamic cache block spatial locality behavior, the accesses to each of these blocks was profiled, gathering information every 100 000 simulated instructions. At the end of each 100 000-instruction phase, the fraction of times that each block in memory had at least one spatial reuse each time it was cached during that phase was computed. This is called the *spatial reuse fraction* for that block. Figure 5.6 shows a graphical representation of the resulting information for three programs. Each row in the graph represents a 1-KB macroblock accessed in a particular phase. For every phase in which a particular macroblock was accessed, there will be a corresponding row. Each row contains one data point for every 32-byte block accessed during the corresponding phase that lies in that macroblock. For the purposes of clarity, the rows were sorted by the average of the block spatial reuse fractions per macroblock. The averages increase from the bottom to the top of the graphs. The cache blocks in each macroblock were also sorted so that their spatial reuse fractions increase from left to right. Some rows are not full, meaning that not all of their



(a) *026.compress* (b) *134.perl* (b) *085.gcc*

Figure 5.6 Spatial reuse fractions (*srf*) for cache-block-sized-data in the accessed macroblocks for three applications.

blocks were accessed during the corresponding phase. Finally, the cache blocks with spatial reuse fractions falling within the same range were plotted with the same marker.

Figure 5.6(a) shows the spatial locality distribution for *026.compress*. Most of the blocks, corresponding to the lighter gray points, have spatial reuse fractions between 0 and 0.25, meaning that there was spatial reuse to those blocks less than 25% of the time they were cached. Very few of the blocks, corresponding to the black points, had spatial reuse more than 75% of the time they were cached. This represents a fairly optimal scenario, because most of the macroblocks contain blocks which have approximately the same amount of reuse. Figure 5.6(b) shows the distribution for *134.perl*. Around 34% of the macroblocks (macroblocks 0 to 6500) contain only blocks with little spatial reuse, their spatial reuse fractions all less than 0.25. About 29% of the macroblocks (macroblocks 13500 to 18900) contain only blocks with large fractions of spatial reuse, their spatial reuse fractions all over 0.75. About 37% of the macroblocks contain cache blocks with differing amounts of spatial reuse. The medium gray points in some of these rows correspond to blocks with spatial reuse fractions between 0.25 and 0.75. However, this information does not reveal the time intervals over which the spatial reuse in these blocks varies. It is possible that in certain small phases of program execution the spatial locality behavior is uniform, but that it changes drastically from one small phase of execution to another. This type of behavior is possible due to dynamically allocated data, where a particular section of memory may be allocated as one type of data in one part of the program, then freed and reallocated as another type later. Finally, Figure 5.6(c) shows the distribution for *085.gcc*, which has similar characteristics to *134.perl*, but has more macroblocks with non-uniform spatial reuse fractions.

5.3.2 Performance improvements

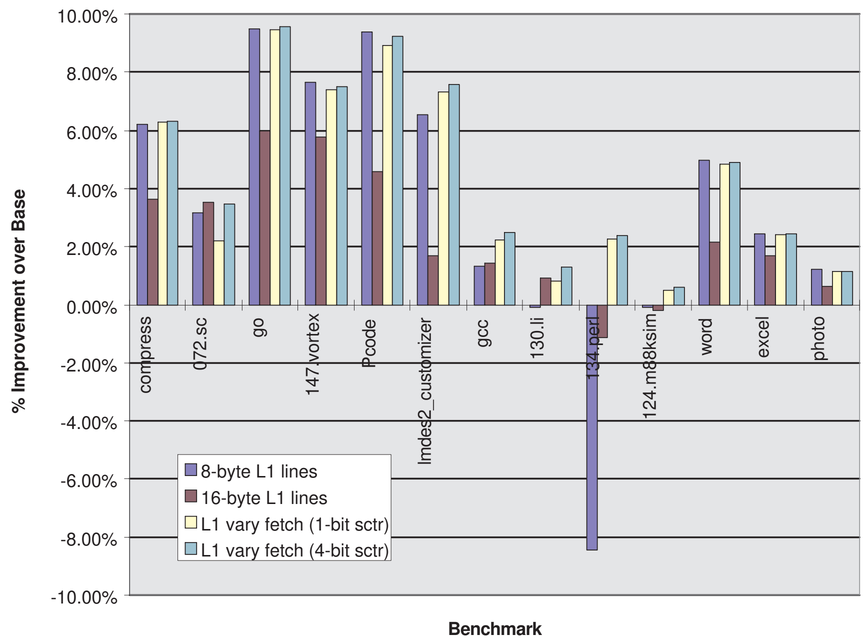
This section examines the performance improvement, or the execution cycles eliminated, over the base eight-issue configuration described in Chapter 3, with one minor change. Because Section 5.3.2.1 will examine the performance of an L2 unified cache with 32-byte lines, rather than the 64-byte line size utilized in Chapters 4 and 6, to avoid unnecessary complication the L1 instruction cache line size is reduced from 64 bytes to 32 bytes.

To support varying fetch sizes, an SLDT and an MAT are employed at each level of the cache hierarchy. The L1 and L2 SLDTs are direct-mapped with 32 entries. Variations in the SLDT sizes and associativities will be examined later in this section. The L1 and L2 MATs initially utilize 1-KB macroblocks, and both one and four-bit *sctr*s are considered. Results are first presented for infinite-entry MATs, then the effect of limiting the number of MAT entries is studied.

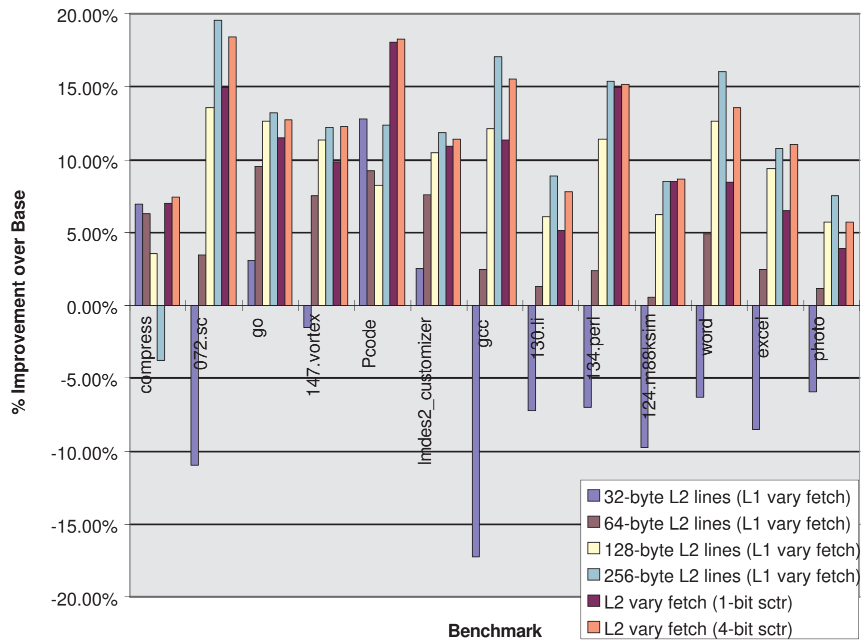
5.3.2.1 Static versus varying fetch sizes

The left bar for each benchmark in Figure 5.7(a) shows the performance improvement achieved by using 8-byte L1 data cache blocks with a static 8-byte fetch size, over the base 32-byte block and fetch sizes. These bars show that the better choice of block size is highly application dependent. The right bars show the improvement achieved by the spatial locality optimization described in this chapter at the L1 level only, using an 8-byte L1 data cache block size, and fetching either 8 or 32 bytes on an L1 data cache miss, depending on the value of the corresponding *sctr*. The results show that the scheme proposed in this dissertation is able either to obtain almost all of the performance or to outperform the best static fetch size scheme. In most cases the 1- and 4-bit *sctr*s perform similarly, but in one case the 4-bit *sctr* achieves around 1.5% greater performance improvement.

The four leftmost bars for each benchmark in Figure 5.7(b) show the performance improvement using different L2 data cache block and (static) fetch sizes, and the L1 spatial locality optimization with a 4-bit *sctr*. The base configuration is again the configuration described in Chapter 3, which has 64-byte L2 data cache block and fetch sizes. These bars show that, again, the better static block/fetch size is highly application dependent. For example, *134.perl* achieves much better performance with a 256-byte fetch size, while *026.compress* achieves its best performance with a 32-byte fetch size, obtaining around 4% performance degradation with 256-byte fetches. The rightmost two bars in Figure 5.7(b) show the performance improvement achieved with the L2 spatial locality optimization, which uses a 32-byte L2 data cache block



(a) L1 Trends



(b) L2 Trends (with L1 varying fetches)

Figure 5.7 Performance for various statically-determined block/fetch sizes and for the spatial locality optimization using both one and four-bit *sctr*s (infinite-entry MAT and 1-KB macroblocks). The spatial locality optimizations use 32-entry direct-mapped SLDTs. All configurations use a 16-KB direct-mapped L1 data cache, and a 256-KB direct-mapped L2 data cache.

size and fetches either 32 or 256 bytes on an L2 data cache miss, depending on the value of the corresponding L2 MAT *sctr*. Again, the spatial locality optimizations are able to obtain almost the same or better performance than the best static fetch size scheme for all benchmarks. In this case a 4-bit *sctr* is clearly better than a 1-bit *sctr*.

Figure 5.8 shows the breakdown of processor stall cycles attributed to different types of data cache misses, as a percentage of the total base configuration execution cycles. The left and right bars for each benchmark are the stall cycle breakdowns for the base configuration and the spatial locality optimization, respectively. The spatial locality optimizations were performed at both cache levels, using the same configuration as in Figure 5.7(b) with a 4-bit *sctr*. For the benchmarks that have large amounts of spatial locality, as indicated from the results of Figure 5.7, large reductions in L2 cold start stall cycles are obtained by fetching 256 bytes on L2 cache misses. At the same time, the L2 conflict miss stall cycles are often reduced, because other data with little spatial locality fetch only a single 32-byte block. The benchmarks with little spatial locality in the L1 data cache, such as *026.compress* and *Pcode*, obtained reductions in L1 capacity miss stall cycles from fetching fewer small cache blocks on L1 misses. Often the L1 conflict miss stall cycles also diminished. In some cases, the L1 cold start stall cycles increase, indicating that the L1 optimizations are less aggressive in terms of fetching more data, however these increases are generally more than compensated by reductions in other types of L1 stall cycles. The L1 conflict miss stall cycles increase for *lmdes2_customizer*, because it tends to fetch fewer blocks on an L1 miss, exposing some conflicts that were interpreted as capacity misses in the base configuration.

Revisiting the example of Section 5.1.1, the access `y->code` on line 11 of Figure 5.3 missed 11 223 times, and the spatial locality optimization fetched 32 bytes for 47% of the misses, and

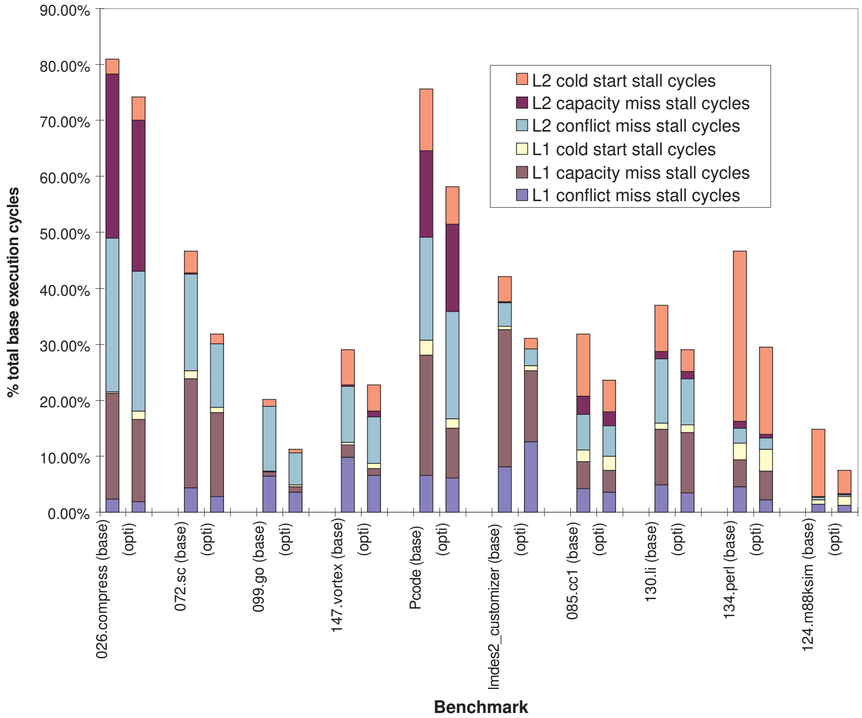


Figure 5.8 Stall cycle breakdown for base and the spatial locality optimizations.

8 bytes for the remaining 53%. On average, 0.99 spatial hits and only 0.02 spatial misses to the resulting data occurred per miss, illustrating that the technique proposed in this dissertation successfully chooses the appropriate amount of data to fetch on a miss.

Figure 5.9 shows a comparison of the performance improvements obtained by the spatial locality optimization when utilizing the uniform sampling methods described in Chapter 3 and when simulating the benchmarks completely. As with the results shown for cache bypassing in Figure 4.26, the results tend to be very similar. However, in several cases, sampling overestimated the performance improvement, generally by less than a 1% delta, but in two cases by close to a 2% delta.

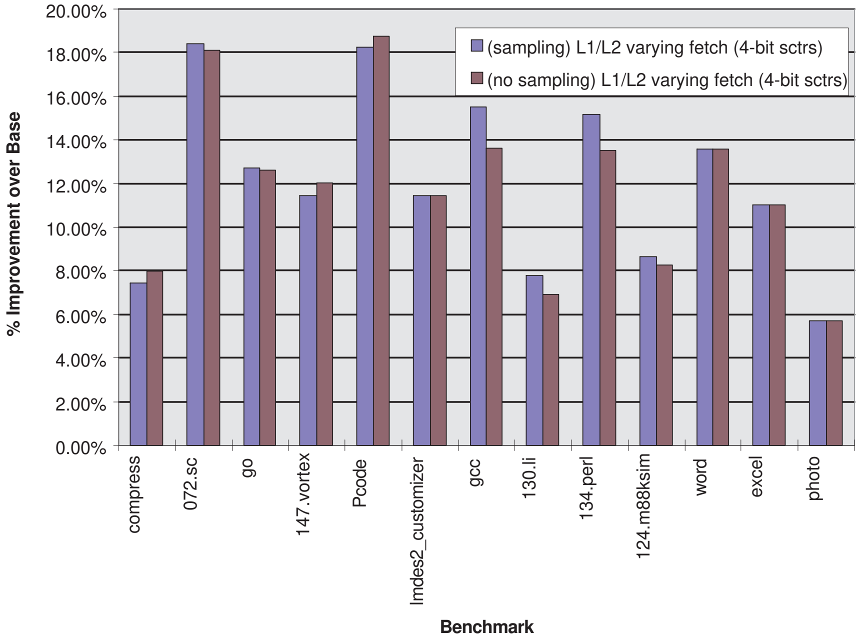


Figure 5.9 Comparison of performance improvements achieved using the sampling methodology described in Chapter 3 to that achieved when completely simulating the benchmarks.

5.3.2.2 Performance of various optimization configurations

Figure 5.10 shows the performance improvement for each benchmark from the spatial locality optimization, using the configuration described in Section 5.3.2.1 with 4-bit *sctrs*, for a range of macroblock sizes using an infinite-entry MAT. For many benchmarks, the performance varies little across the different macroblock sizes. However, for the other benchmarks, there is no set pattern to the performance variations across the different macroblock sizes. In the case of *lmdes2_customizer* and *Photo*, the performance tends to worsen as the macroblock sizes is increased, most likely due to the coarser granularity at which the spatial locality information is maintained. In other cases, such as *026.compress* and *Pcode*, the performance tends to worsen at the smallest macroblock sizes. The reasons for the different trends is that at the smaller macroblock sizes more spatial fetches tend to be performed, especially at the L2 cache level,

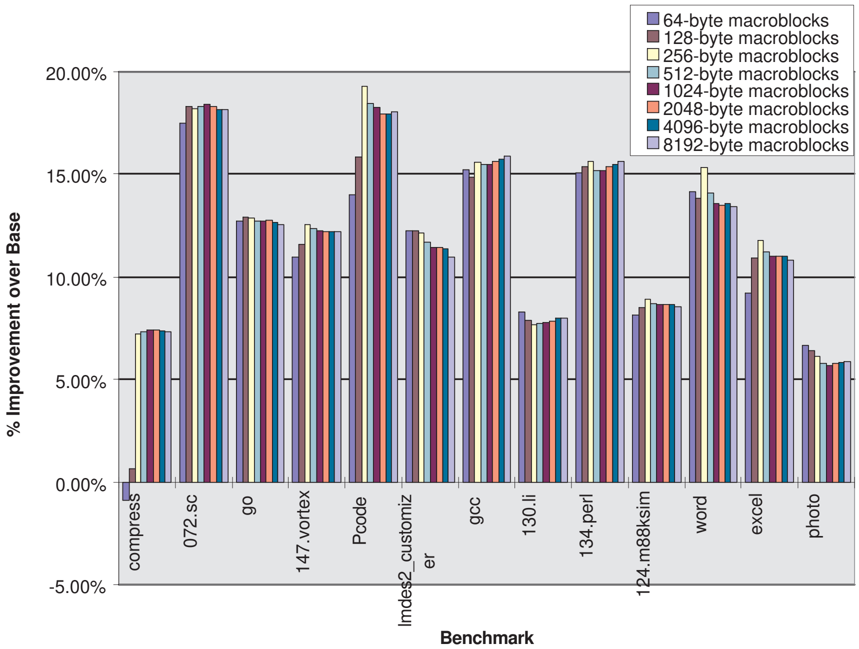


Figure 5.10 Spatial locality optimization performance improvements using different macroblock sizes and an infinite-entry MAT.

which hurts benchmarks that have less spatial locality, such as *26.compress* and *Pcode*, and helps those benchmarks with more spatial locality. Although the smaller macroblock sizes provide finer granularity, the counter values take longer to adapt to the observed patterns, because less accesses are made to each macroblock. This is especially true at the L2 cache level, as most references hit in the L1 cache, and therefore it takes longer to train the L2 *sctrs*. Because the *sctr* values are initialized to the threshold value described in Section 5.2.3, which is the lowest *sctr* value at which spatial fetches are made, the decisions tend to be slightly biased towards making spatial fetches, which hurts benchmarks with little spatial locality. It is possible that if smaller macroblock sizes are to be used, a smaller *sctr* size may be beneficial.

Figures 5.11-5.13 show the performance improvements for the different macroblock sizes, with 1024-entry, 512-entry, and 256-entry direct-mapped MATs, respectively. For the benchmarks with less spatial locality, such as *026.compress* and *Pcode*, the performance grows worse

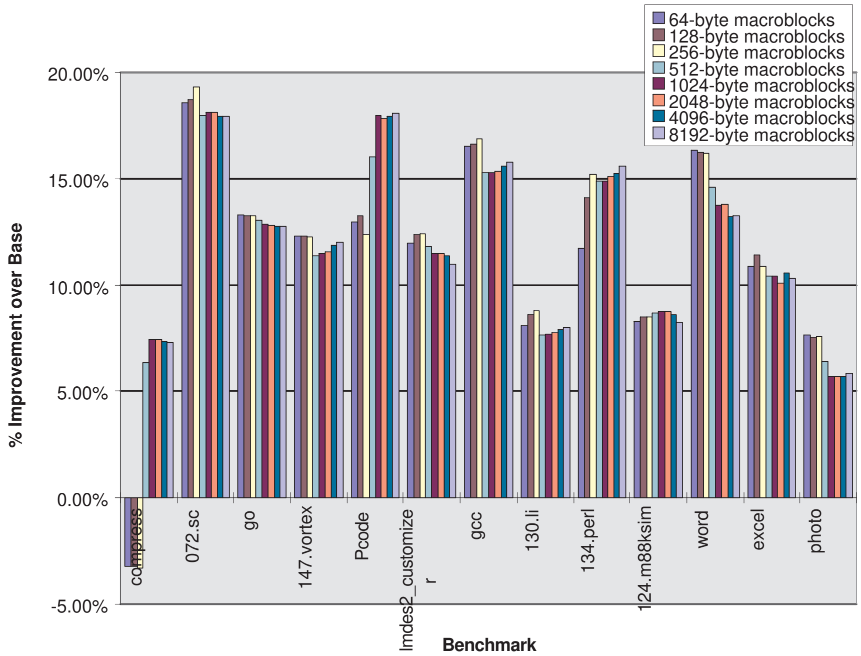


Figure 5.11 Spatial locality optimization performance improvements using different macroblock sizes and a 1024-entry direct-mapped MAT.

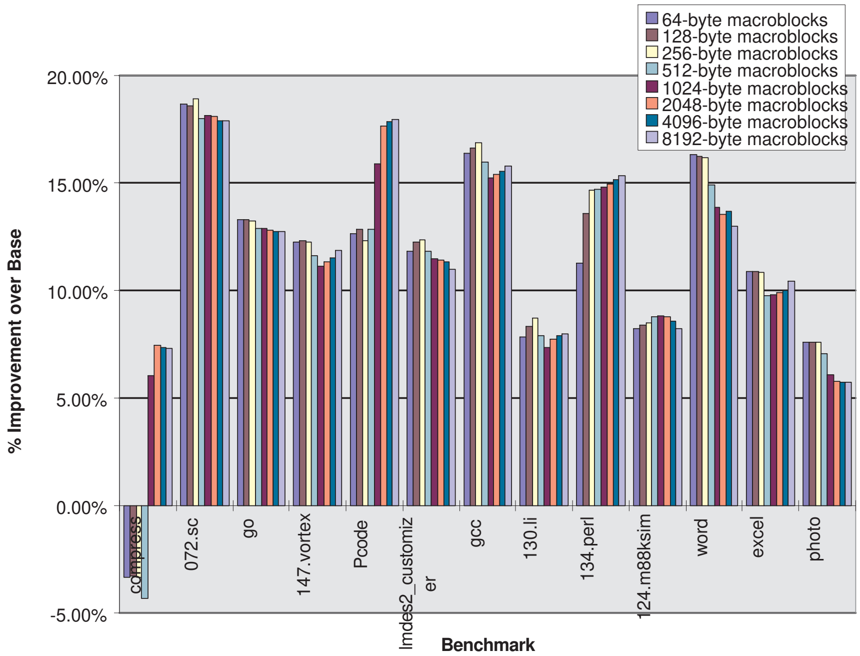


Figure 5.12 Spatial locality optimization performance improvements using different macroblock sizes and a 512-entry direct-mapped MAT.

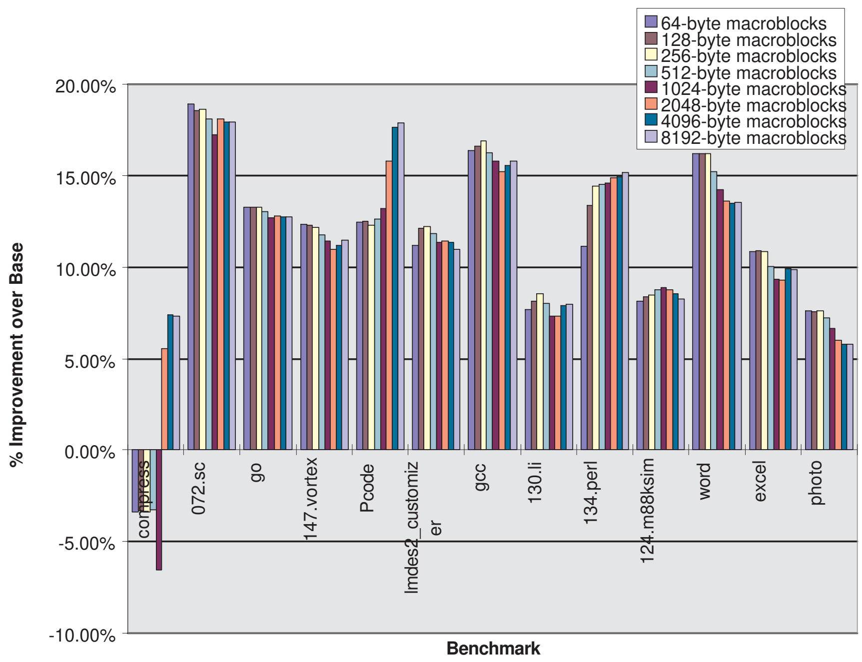


Figure 5.13 Spatial locality optimization performance improvements using different macroblock sizes and a 256-entry direct-mapped MAT.

at the smaller macroblock sizes, due to the fact that MAT entries must be reallocated more often, and as described earlier, the initial *sctr* value is biased towards spatial fetches. For many of the other benchmarks that have more spatial locality, the performance improvement with the finite-entry MATs is slightly better with the smaller macroblock sizes, for similar reasons.

Figure 5.14 shows the performance improvements for the different macroblock sizes, with 512-entry fully associative MATs. Unlike the effect on bypassing, shown in Figure 4.18, greater MAT associativity has very little effect on the spatial locality optimization performance.

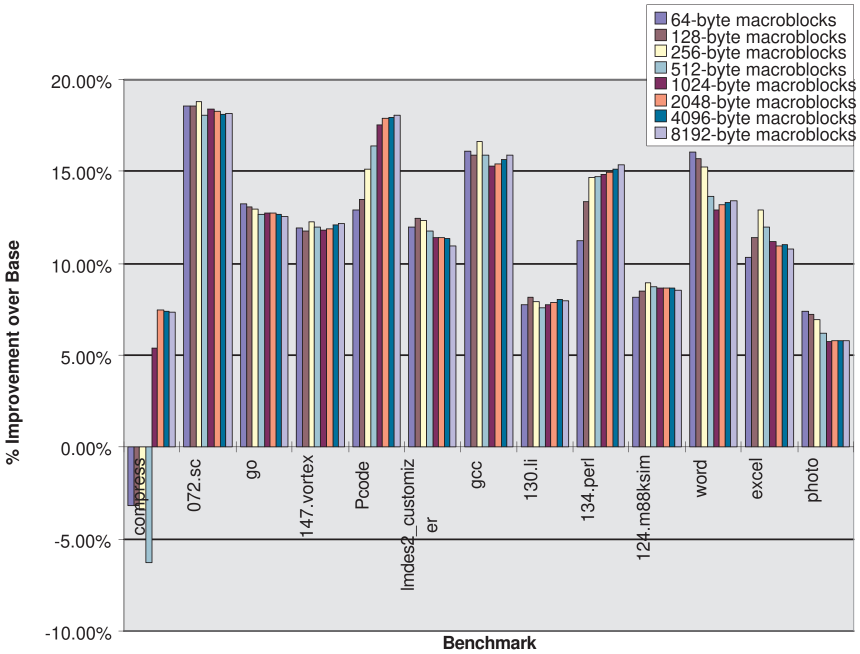
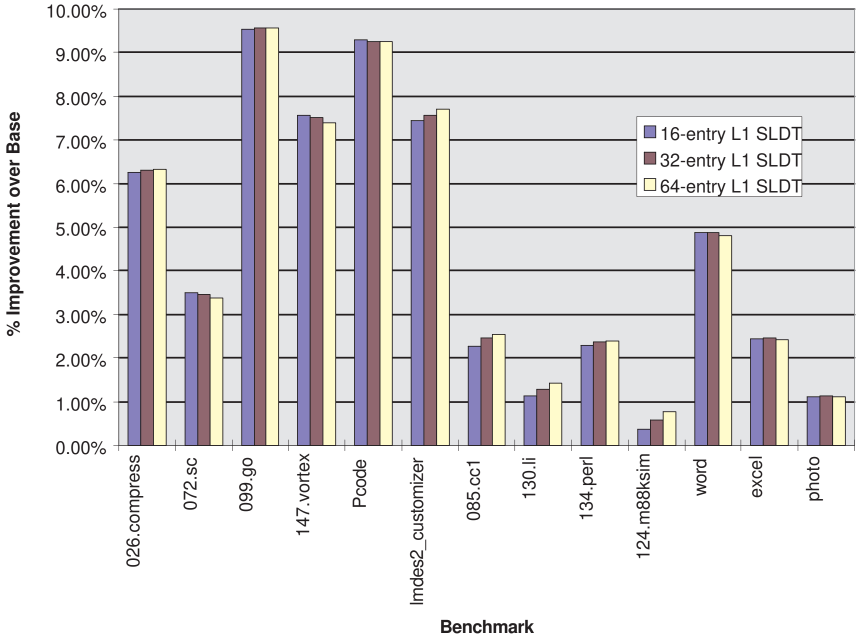
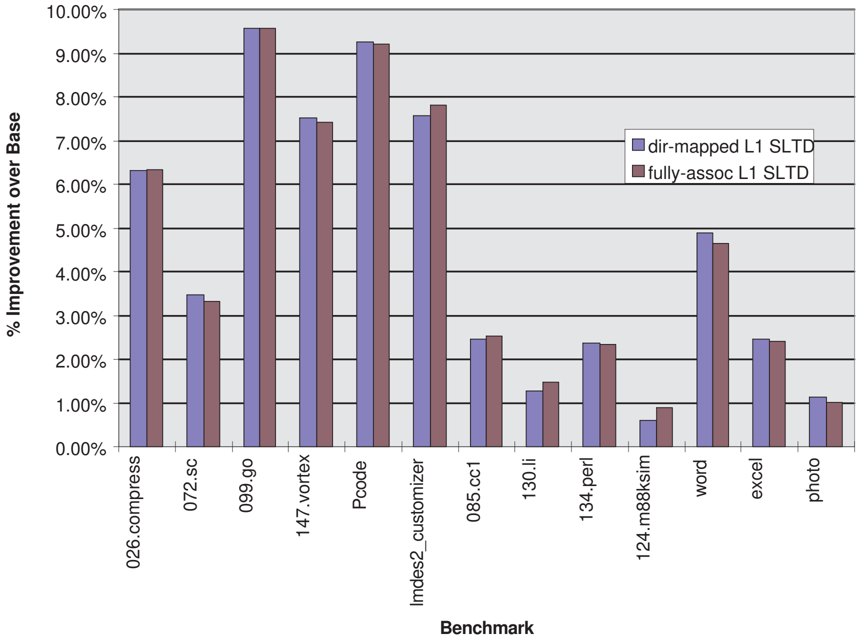


Figure 5.14 Spatial locality optimization performance improvements using different macroblock sizes and a 512-entry fully associative MAT.

Figures 5.15(a) and (b) show the effects of varying the L1 SLDT size and associativity, respectively, using an infinite-entry MAT with 1-KB macroblocks and 4-bit *sctrs*. A static 64-byte L2 block size was utilized in this case. In general, the performance varies little for different SLDT sizes, and when increasing the associativity. Therefore, the L1 SLDT size could be reduced to 16 entries, in order to reduce the hardware overhead, with little performance effect. Figures 5.16(a) and (b) show the effects of the same variations in the L2 SLDT, using a 32-entry direct-mapped L1 SLDT. The performance again varies little with increased associativity, but is slightly more affected by the variations in L2 SLDT size. This makes sense, since the SLDT entries monitor the cache blocks, and there are far more blocks in the L2 cache.

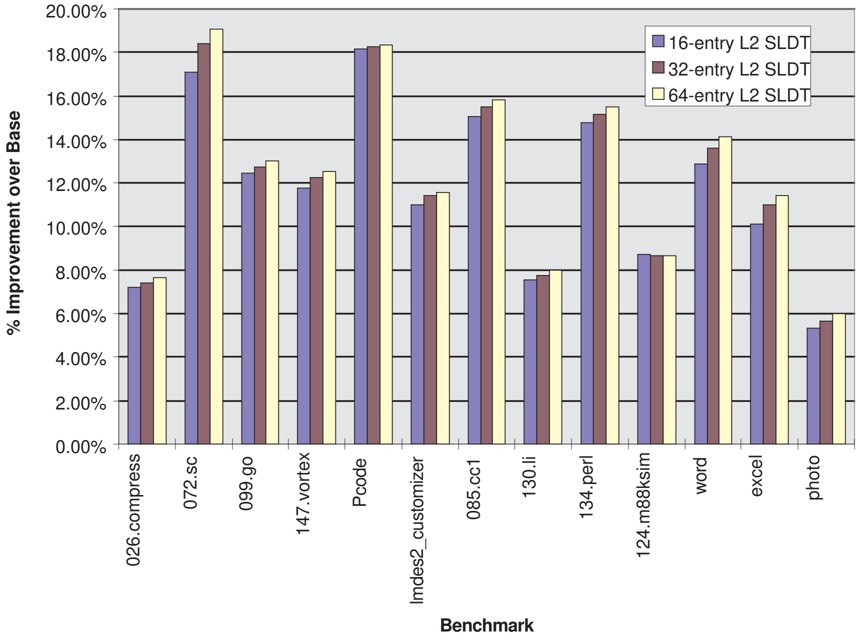


(a) varying L1 SLDT size

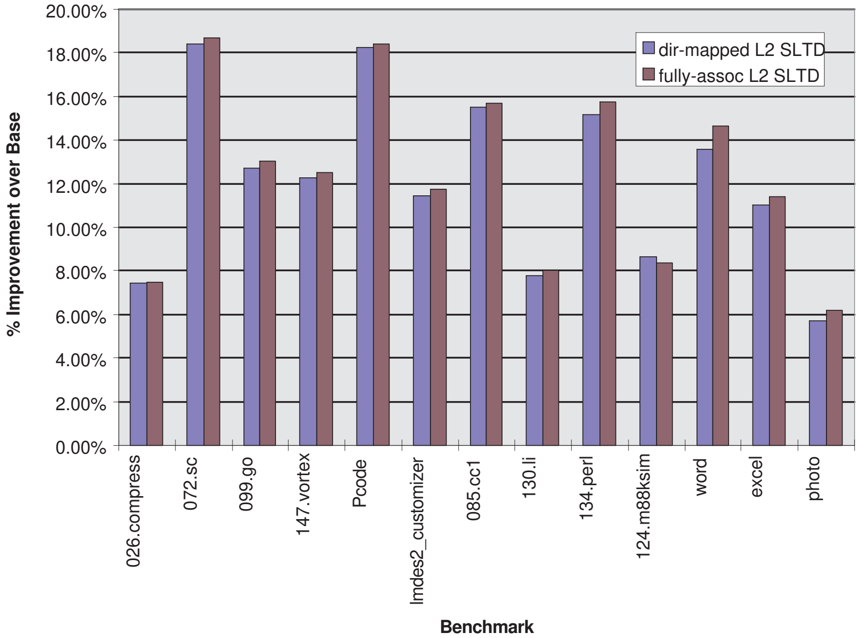


(b) varying L1 SLDT associativity

Figure 5.15 Performance improvements using different L1 SLDT sizes (the defaults is a 32-entry direct-mapped L1 SLDTs). A static 64-byte L2 block size is used.



(a) varying L2 SLDT size



(b) varying L2 SLDT associativity

Figure 5.16 Performance improvements using different L1 and L2 SLDT sizes (the defaults are 32-entry direct-mapped L1 and L2 SLDTs).

5.3.2.3 Optimization performance improvement details

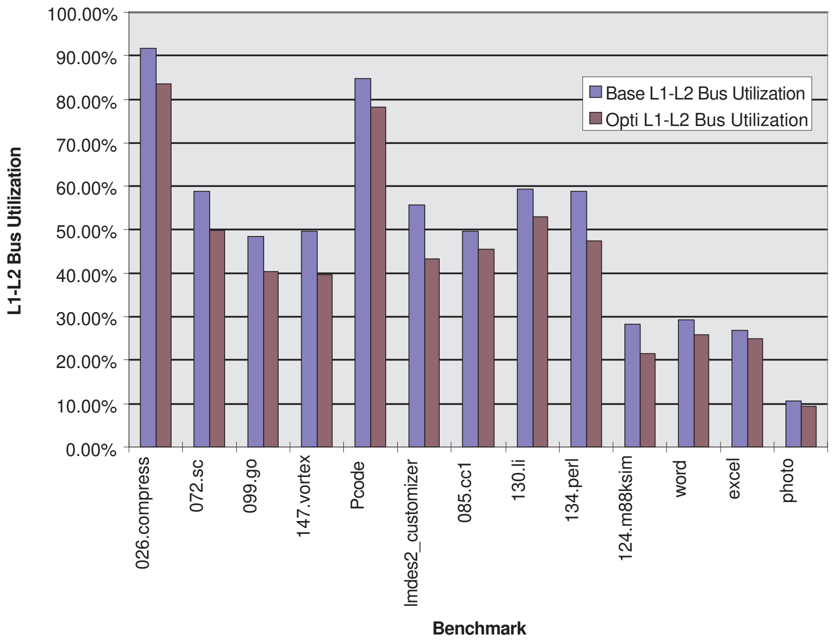
Figures 5.17(a) and (b) show the utilization of the L1-L2 bus and the system bus (connecting the L2 cache with the main memory), respectively, for both the base (unoptimized) configuration and the optimized configuration using infinite-entry MATs with 1-KB macroblocks and 4-bit *sctrs*. The direct-mapped L1 and L2 SLDTs contain 32 entries. The reductions in bus traffic are due both to improvements in the hit ratio and to reductions in the amount of data requested for data with little spatial locality.

Figure 5.18(a) shows the L1 read hit ratio for both configurations. In most cases the hit ratio improves; however, in many cases the hit ratio is actually reduced. The reason why these benchmarks obtain overall performance improvements is that the bus traffic is reduced, and therefore, the average latency incurred per access is also reduced. This reduction in average miss penalty is demonstrated in Figure 5.18(b).

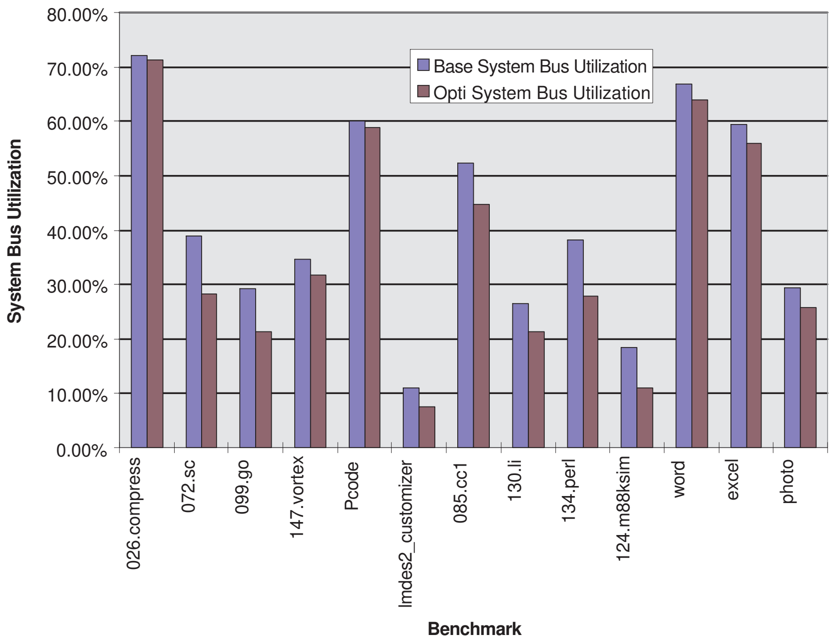
5.3.2.4 Set-associative data caches

Increasing the set associativity of the data caches can reduce the number of conflict misses, which may in turn reduce the advantage offered by this optimization. However, the reductions in capacity and cold start stall cycles that the optimization achieves should remain. To investigate these effects, the data cache configuration was modified to have a two-way set-associative L1 data cache and a two-way set-associative L2 data cache.

Figure 5.19 shows the new performance improvements for the spatial locality optimization, as well as the improvements shown earlier for direct-mapped caches. The improvements have reduced significantly for some benchmarks over the direct-mapped improvements. However,

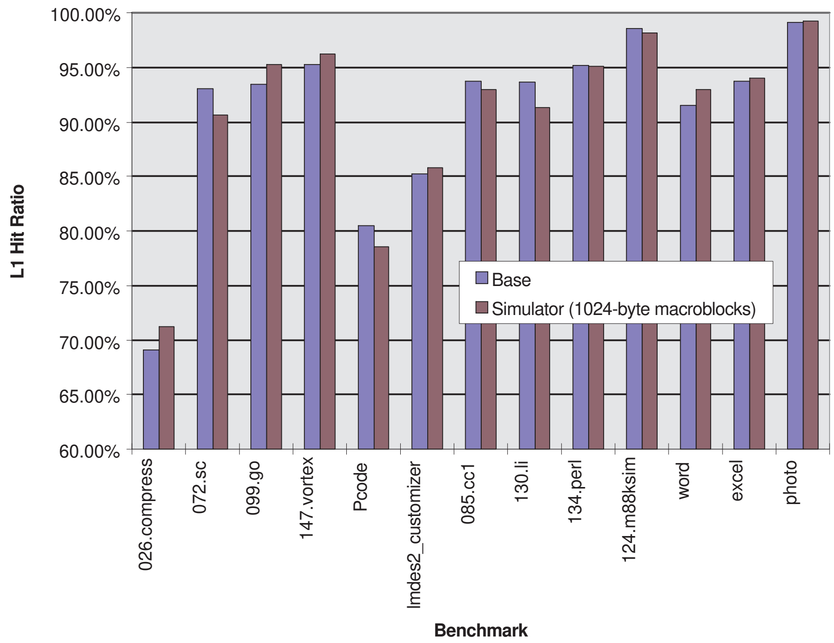


(a) L1-L2 bus utilization

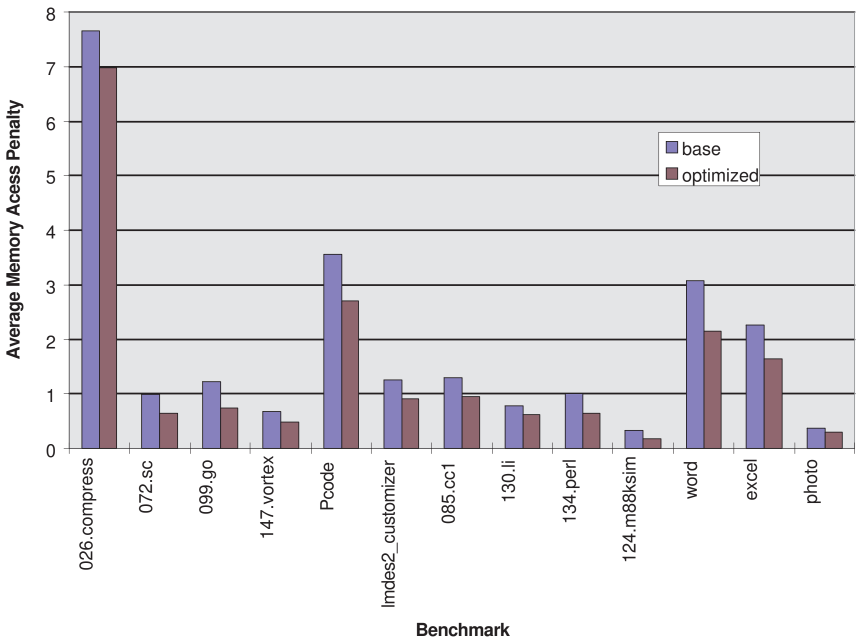


(b) system bus utilization

Figure 5.17 Bus utilization for the spatial locality optimization (infinite-entry MAT).



(a) L1 Read Hit Ratio



(b) average miss penalty per load access

Figure 5.18 L1 hit ratios and average miss penalties for the spatial locality optimization (infinite-entry MAT).

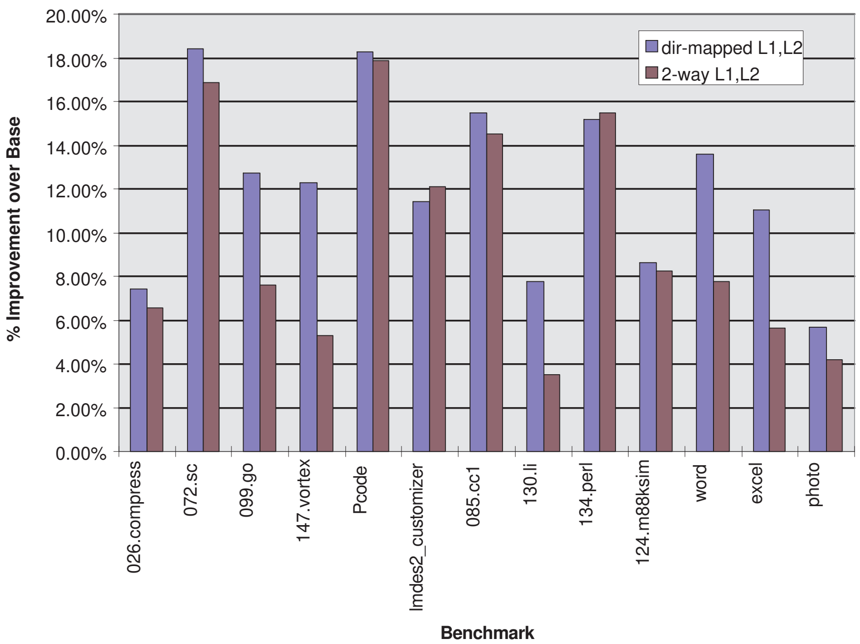


Figure 5.19 Performance for the spatial locality optimizations with two-way set-associative L1 and L2 data caches.

large improvements are still achieved for some benchmarks, particularly due to the reductions in L2 cold start stall cycles for data with spatial locality.

5.3.2.5 Growing memory latency effects

As discussed in Section 1, memory latencies are increasing, and this trend is expected to continue. Figure 5.20 shows the improvements achieved by the spatial locality optimization when applied to direct-mapped caches for the 100-cycle latency used earlier, as well as for 200 and 300-cycle latencies, each relative to a base configuration with the same memory latency. Most of the benchmarks see much larger improvements from the spatial locality optimization as the latency grows, with the exception of *026.compress*. Because *026.compress* has very little spatial locality to exploit, the longer latency cannot be hidden as effectively. Although the raw

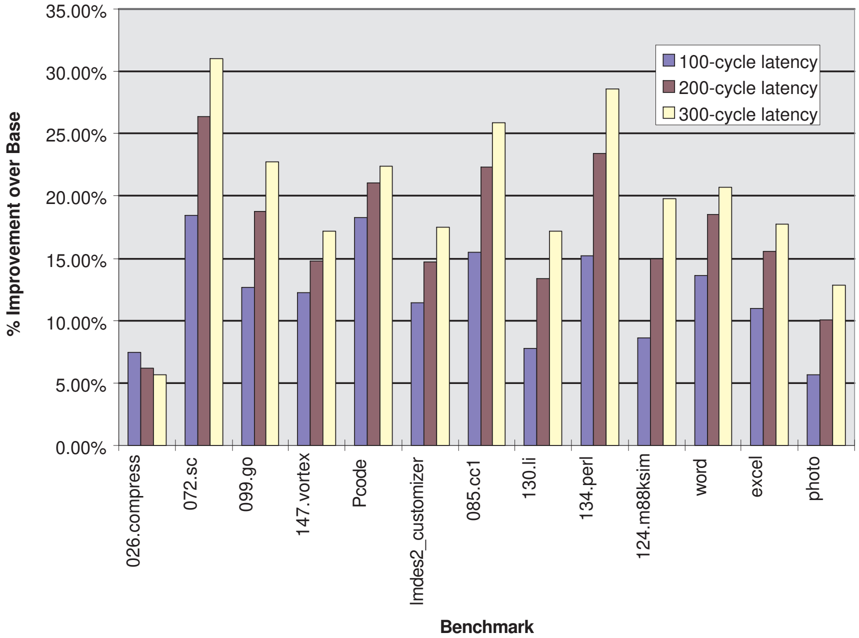


Figure 5.20 Performance for the spatial locality optimizations with growing memory latencies.

number of cycles eliminated grows, as a percentage of the associated base execution cycle count, it becomes smaller.

5.3.3 Comparison to upper-bound decisions

Just as the performance of the bypassing optimization was compared to upper-bound bypassing decisions in Chapter 4, it would be useful to compare the fetch size decisions made by the implementation discussed in this chapter to upper-bound fetch size decisions. However, the problem of determining optimal fetch size decisions is intractable; therefore, the algorithm described in this section only determines those fetch size decisions that are clearly optimal. It will be shown later in this section that these simple decisions generally cover most of the cache misses.

The simple optimal nonspatial fetches detected are those accesses for which there is no possibility of spatial reuse if the large fetch size is employed, given the reuse windows for the corresponding cache lines, even if no other spatial fetches are made. If an access does not have a simple optimal nonspatial fetch, because there is a potential for at least one spatial reuse, then several criteria must be met in order for that access to have a simple optimal spatial fetch. First, the extra cache blocks brought into the cache as a result of the spatial fetch must not displace any other regular (accessed) data from the cache, resulting in new misses. The extra cache blocks will certainly displace other data; however, if that displaced data has no reuse within the reuse window (ignoring all extra spatially fetched blocks), then there is no degradation in hit ratio from this spatial fetch. Also, if there are a large number of spatial fetches, the extra blocks brought into the cache may displace each other before their respective spatial reuses. Therefore, the second criteria for a simple optimal spatial fetch is that, given all the potential spatial fetches in each corresponding reuse window, the access in question must still obtain at least one spatial reuse.

Figure 5.21 shows the main algorithm for determining the simple optimal fetch size decisions. Algorithm `upper_bound_sploc` is organized similar to the bypassing algorithms presented in Section 4.1.2, where the trace is processed in reverse chronological order, and reuse windows for each cache line (of the smaller block size) are maintained. Now for every trace address examined, there are $N = (BLOCK\_SIZE2/BLOCK\_SIZE)$ lines that must be examined, one for each physical cache line in the larger block size that is being considered. Therefore, the nested `for` loop examines each of these cache line addresses, and the corresponding reuse windows, in turn. In order to facilitate the spatial locality detection, a single *spinfo* structure is created for each address read from the trace, that is referenced by each of the N reuse window

```
/* define */
CA_ASSOC = cache associativity;
CA_ENTRIES = number of sets in cache;
CACHE_INDEX(address) = physical line size set index corresponding to address;
CACHE_INDEX2(address) = large block size set index corresponding to address;
ADDR(address) = small block address;
ADDR2(address) = large block address;
BLOCK_SIZE = small (physical) block size;
BLOCK_SIZE2 = large block size;

algorithm upper_bound_sploc
{
    cache_queues[0..CA_ENTRIES] = NULL;

    for each address (reverse order) do:
    {
        ca_entry = CACHE_INDEX(address);
        ca_entry2 = CACHE_INDEX2(address);
        malloc spinfo;
        spinfo->hit_count = spinfo->miss_count = spinfo->valid_count = 0;
        spinfo->Sref_overflow = FALSE;
        spinfo->fetch_type = unknown;

        for (addr=ADDR2(address);addr<=ADDR2(address)+BLOCK_SIZE2;addr+=BLOCK_SIZE)
        {
            malloc new_entry;
            new_entry->spinfo = spinfo;
            spinfo->valid_count++;

            if (addr == ADDR(address) new_entry->type = NS-ref;
            else new_entry->type = S-ref;

            check_reuse(addr, new_entry);
            enqueue new_entry in cache_queues[ca_index];
        }
        if (!spinfo->hit_count)
        {
            spinfo->fetch_type = non-spatial;
            dequeue all S-ref type new_entries just created for address;
        }
        else if (spinfo->Sref_overflow)
            dequeue all S-ref type new_entries just created for address;
    }
    dequeue all remaining entries in cache_queues[0..CA_ENTRIES];
}
```

Figure 5.21 Algorithm to determine simple optimal fetch size decisions.

entries created. The *spinfo* fetch size is initially set to *unknown*. Each reuse window entry is flagged as one of two types:

- $NS{-}ref$: The block that is actually accessed by the trace address.
- $S{-}ref$: A block that would be created due to a spatial fetch by the $NS{-}ref$.

Therefore, one entry is an $NS{-}ref$, and $N-1$ entries are $S{-}ref$s.

For each cache line considered in the inner loop of algorithm `upper_bound_sploc`, a call is made to routine `check_reuse`, shown in Figure 5.22. This routine performs a number of duties, while iterating through the corresponding reuse window entries. As in the bypassing algorithms described in Section 4.1.2, the reuse window for the corresponding line is maintained and updated. However, only $NS-ref$s are counted for the purpose of updating the reuse window, which is performed by placing all $NS{-}ref$s in the temporary cache set *set_NS*. Also, only the $NS-ref$s are examined when determining if there is reuse with the given address. This is because a spatial reuse is useful only if the data is reused by a referenced address, not when it is reused by a cache block brought in as a result of another spatial fetch.

If a reuse is found, different actions are taken, depending on the *new_entry* type. If it is an $S{-}ref$, then this is potentially an optimal spatial fetch, and the corresponding *spinfo* hit counter is incremented to reflect the potential spatial reuse. Also, a check is performed to determine if the second criteria for a simple optimal spatial fetch is met for this spatial reuse. Another temporary cache set, *set_all*, is updated with all references (both $NS{-}ref$s and $S{-}ref$s), to ensure that this spatial reuse will still occur if all other potential spatial fetches in the reuse window also occur. If not, because there are too many other potential spatial fetches between this reference and the reuse opportunity, then the corresponding *spinfo* $S{-}ref$ overflow flag is set.

```
algorithm check_reuse(addr, new_entry)
{
    new_entry->ref = N-ref;
    set_NS = NULL;
    set_all = NULL;
    for each entry in cache_queues[ca_entry] do:
    {
        if (!done && entry->addr not in set_all)
            add entry to set_all;
        if (!done && entry->type == NS-ref)
        {
            if (new_entry->ref == N-ref && addr == ADDR(entry->address))
            {
                new_entry->ref = R-ref;
                if (new_entry->type == S-ref)
                {
                    new_entry->spinfo->hit_count++;
                    if (|set_all| > CA_ASSOC)
                        new_entry->spinfo->Sref_overflow = TRUE;
                }
                else if (|set_all| > CA_ASSOC)
                    mark\_new\_misses(address);
            }

            if (entry->addr not in set_NS)
            {
                add entry to set_NS;
                if (|set_NS| == CA_ASSOC) done = TRUE;
            }
        }
        else dequeue entry;
    }
}
```

Figure 5.22 Routine to check for spatial reuse opportunities.

On the other hand, if this is an $NS\text{–}ref$, then the reuse is a temporal, not a spatial, reuse. A check is then performed to ensure the first criteria for a simple optimal spatial fetch is still met for all potential spatial fetches between this reference and its temporal reuse. In this case, the size of the temporary set containing both reference types, *set_all*, is again checked to determine if the potential spatial reuses in the reuse window will result in the displacement of this reference before its temporal reuse. If the set size is larger than the associativity, then all tentative spatial fetches between this reference and its temporal reuse are conservatively marked so that they

```
algorithm mark_new_misses(address)
{
    ca_entry = CACHE_INDEX(address);

    for each entry in cache_queues[ca_entry] do:
    {
        if (entry->type == NS-ref && ADDR(address) == ADDR(entry->address))
            break;
        else if (entry->type == S-ref)
            entry->spinfo->miss_count++;
    }
}
```

Figure 5.23 Routine to detect new misses due to spatial fetches.

are not counted as simple optimal spatial fetches. This marking is performed by the routine `mark_new_misses`, shown in Figure 5.23. The entries in the associated reuse window are again traversed, until the temporal reuse is again found. For each $S-ref$ found, its corresponding *spinfo* miss counter is incremented.

Finally, after each cache block within the potential large block has been examined in the main algorithm shown in Figure 5.21 and has been added to the corresponding reuse windows, some of the *spinfo* information is examined to determine if a fetch size decision can be made for the reference. If no potential spatial reuses were detected, then the *spinfo* hit counter will be zero. In this case the reference can be marked as a simple optimal nonspatial fetch, and the *spinfo* fetch size is set to *nonspatial*. In addition, all $S-ref$ entries created for the reference can be removed from their corresponding reuse windows. If the hit counter is nonzero, but an $S-ref$ overflow was detected, then the reference is neither a simple optimal nonspatial nor spatial fetch. Therefore, the fetch size is left as *unknown*, but the $S-ref$ entries created for the reference are removed from their corresponding reuse windows. For other references, which are still potentially simple optimal spatial fetches, the $S-ref$ entries are left in the corresponding reuse windows for consideration. When the last of the N cache blocks created for a reference is

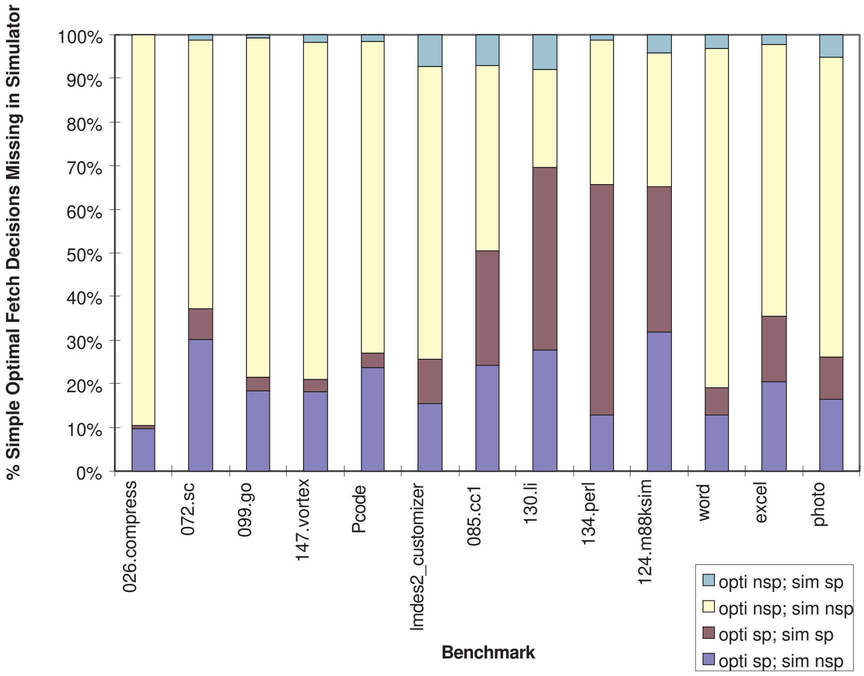
```
algorithm dequeue(entry)
{
    entry->spinfo->valid_count--;

    if (entry->spinfo->valid_count == 0)
    {
        if (entry->spinfo->fetch_type == unknown && entry->spinfo->miss_count == 0 &&
                  spinfo->hit_count > 0 && spinfo->Sref_overflow == FALSE)
            entry->spinfo->fetch_type == spatial;

        note fetch type;
    }
}
```

Figure 5.24 Routine to make simple fetch size decisions upon dequeues.

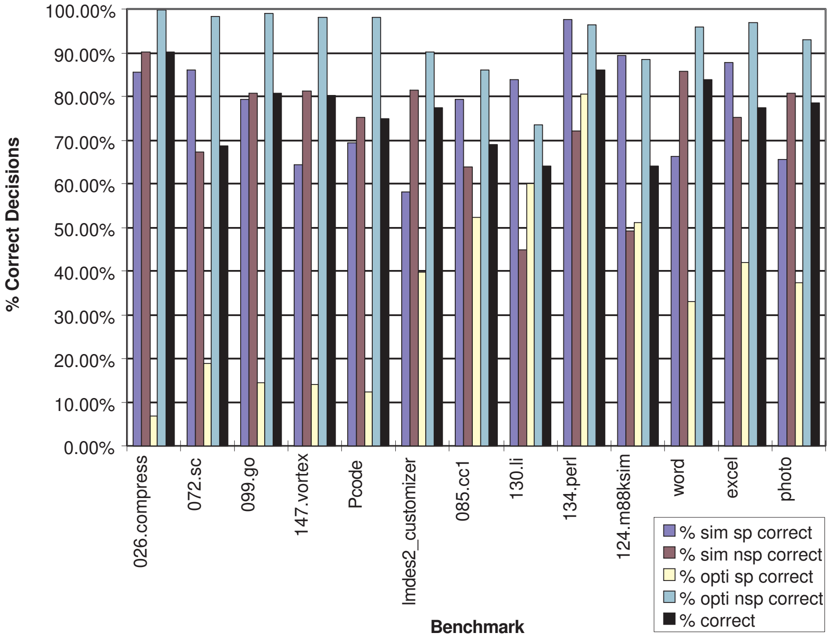
dequeued from its reuse window, a check is made to determine if it is a simple optimal spatial fetch.

The dequeuing checks are shown in Figure 5.24. If the fetch type is still *unknown*, and the miss counter is zero, meaning that no extra cache misses would be created by a spatial fetch for this reference, and the hit counter and $S{-}ref$ overflow flag are set appropriately, the reference is marked *spatial*. Finally, the fetch type is noted, and if the reference missed in the simulator, the fetch size decisions are compared.

Figure 5.25(a) shows a breakdown of the simple optimal fetch size decisions identified by algorithm `upper_bound_sploc`, that missed in the simulator. The missing references are broken down into four sections based on the upper-bound and simulator decisions. The bottom two sections correspond to those references for which the upper-bound analysis detected a simple optimal spatial fetch, and the upper two sections correspond to those that the upper-bound analysis detected as simple optimal nonspatial fetches. Ideally, the middle two sections should be largest, as they correspond to the accesses for which the upper bound and simulator decisions matched. The outer two sections should ideally be of zero size. The breakdown shows that



(a) breakdown of fetch size decisions



(b) correctness of fetch size decisions

Figure 5.25 Breakdown of the simple optimal fetch size decisons made by the upper-bound algorithm that missed in the simulator (*nsp* and *sp* denote nonspatial and spatial fetches, respectively). Also shown is the percentage of fetch size decisions that were correct.

in general, the simulated implementation was much better at correctly identifying nonspatial fetches than spatial fetches.

Figure 5.25(b) examines the correctness of the decisions in several ways. The leftmost two bars for each benchmark show the percentage of correct simulator spatial and nonspatial fetch size decisions, respectively. The correctness is reasonably good, and generally, between 60 and 90% of the spatial and nonspatial decisions were correct. Another way of looking at the correctness is to examine how many of the upper-bound decisions were correctly matched by the simulated implementation. In some ways, this shows the coverage of the simulated implementation. The second pair of bars for each benchmark shows the percentage of upper-bound decisions that were correctly matched in the simulator. As predicted by the breakdown of Figure 5.25(a), a large percentage of the simple optimal nonspatial fetches were correctly matched, but usually, a much smaller portion of the simple optimal spatial decisions were matched. The benchmarks that contained much more spatial locality, as indicated by the results of Figure 5.7(a), such as *130.li*, *134.perl*, and *124.m88ksim*, tended to match a much greater percentage of the simple optimal spatial decisions. The results show that, when the simulated implementation chose to make a spatial fetch, that decision was usually correct; however, many opportunities for spatial reuse were not identified and exploited. Finally, the rightmost bar for each benchmark shows the overall percentage of correct decisions that were made. On average, approximately 75% of the fetch size decisions were correctly made by the simulated implementation.

Table 5.2 shows the percentage of total accesses and simulator cache misses for which the upper-bound analysis found an optimal decision. On average, while only a third of the accesses

Table 5.2 Percentage of accesses and simulator misses for which the upper-bound analysis made an optimal decision.

Benchmark	% Accesses Covered	% Misses Covered
026.compress	56.23%	90.51%
072.sc	40.33%	76.86%
099.go	19.31%	31.79%
147.vortex	33.60%	44.02%
Pcode	39.34%	60.56%
lmdes2_customizer	32.89%	50.58%
085.cc1	32.52%	58.00%
130.li	35.03%	57.22%
134.perl	35.22%	67.83%
124.m88ksim	40.11%	55.36%
Word	22.91%	49.57%
Excel	26.26%	62.86%
Photo	31.94%	55.39%
Average	34.28%	58.50%

were covered by an optimal fetch size decision, over half of the accesses that actually missed in the cache (and that needed to make a fetch size choice) were covered.

5.4 Integrated Bypassing and Spatial Locality Optimizations

As the memory latencies increase, intelligent cache-management techniques will become increasingly important. This section examines the performance improvement achieved by integrating the spatial locality optimization described in this chapter with the intelligent cache bypassing optimization described in Chapter 4. This performance is compared to the performance obtained by simply doubling the caches at each level of the hierarchy, and the hardware overheads of the configurations are estimated and compared.

5.4.1 Performance improvements

Based on the performance improvements for bypassing demonstrated in Section 4.3.2 and in order to minimize the hardware overhead while obtaining high performance, the following

Table 5.3 Cycle times for doubled caches and bypass buffers, calculated by CACTI, and the resulting processor clocks for various processor speeds.

Cache Type	Cache Size (bytes)	Block Size (bytes)	Associativity	Cycle Time (ns)	200MHz clocks	300MHz clocks	500MHz clocks
Doubled L1	32K	32	direct-mapped	9.33	2	3	5
L1 Bypass Buffer	1K	32	4-way	10.48	3	4	6
	1K	32	2-way	9.52	2	3	5
	1K	32	1-way	6.34	2	2	4
	1K	8	1-way	6.73	2	3	4
Doubled L2	512K	64	direct-mapped	15.79	4	5	8
L2 Bypass Buffer	16K	64	4-way	12.79	3	4	7
	16K	64	2-way	11.60	3	4	7
	16K	64	1-way	7.95	2	3	6
	16K	32	4-way	12.00	3	4	6

parameters were selected: 2-KB macroblocks, 512-entry direct-mapped MATs, 4-bit access counters, and 2-bit *decr_ctrs*. Additionally, the cache access and cycle time model (CACTI) [61] was used to examine the cycle times required by various bypass buffer configurations, as well as by the doubled caches, in order to obtain an optimized configuration requiring no more than the cycle time of the doubled caches. Table 5.3 shows that a four-way set-associative 32-entry L1 bypass buffer requires a greater access time than a doubled L1 data cache. Since Figure 4.21(b) illustrated that decreasing the L1 bypass buffer set associativity had little effect on the performance, the cycle time of the L1 bypass buffer is lowered by reducing the L1 bypass buffer set associativity rather than the size. The cycle time for a direct-mapped 32-entry L1 bypass buffer, given in Table 5.3, is lower than that of a doubled L1 data cache. At the L2 level, Table 5.3 shows that a four-way set-associative 256-entry L2 bypass buffer has a cycle time less than that of a doubled L2 unified cache. Therefore, no reductions in the associativity are required. Also given are the cycle time for the direct-mapped L1 bypass buffer and four-way set-associative L2 bypass buffer with smaller lines, since the spatial locality optimization will be used to adjust the fetch sizes of bypassed data as well. The resulting cycle times are still less than those of the doubled caches at each level.

Then, the SLDT and MAT at each cache level are used to detect spatial locality and control the fetch sizes for both the data cache and the bypass buffer at that level. From the results of Section 5.3.2, 32-entry direct-mapped SLDTs and 4-bit *sctrs* are chosen.

Figure 5.26 shows the improvements achieved by combining the bypassing and spatial locality optimizations at both cache levels for a 100-cycle memory latency, using the configuration described above. Also shown are the performance improvements achieved by doubling both the L1 and L2 data caches. Doubling the caches is a brute-force technique used to improve cache performance that Section 5.4.2 will show is much more expensive than the integrated intelligent cache-management optimizations. Figure 5.26 shows that performing the integrated optimizations at both cache levels often outperforms simply doubling both levels of cache. The only case where the doubled caches greatly outperform the integrated optimizations is for *026.compress*. This improvement comes mostly from doubling the L2 cache, because the hash tables in *026.compress* can fit into a 512-KB cache. Comparing Figure 5.26 to the bypassing improvements in Chapter 4 shows that significant improvements can often be achieved by intelligently controlling the fetch sizes into the caches and bypass buffers.

5.4.2 Hardware overhead

In this section, the hardware overhead of the integrated run-time optimization schemes examined in Section 5.4.1 is estimated, and compared to the cost of doubling the L1 data cache and the L2 unified cache. As discussed in Section 4.2.1, the cost of the MAT hardware is amortized when performing both spatial locality and bypassing optimizations. For this reason, the combined hardware cost for both of these optimizations is computed, just as their combined performance was compared to the performance of doubling the caches in Section 5.4.1.

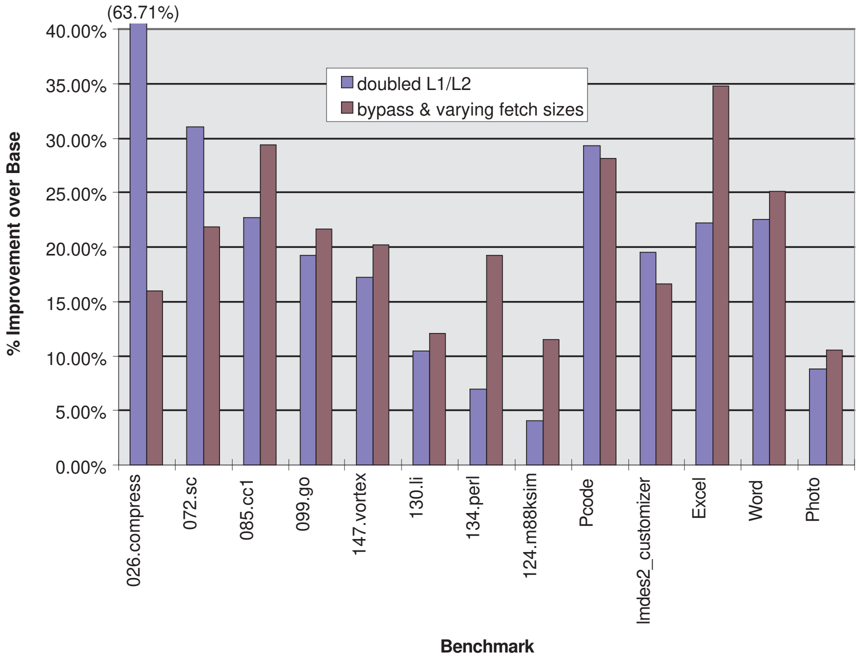


Figure 5.26 Comparison of doubled caches to integrated spatial locality and bypassing optimizations.

The additional hardware cost incurred by the run-time optimization scheme is small compared to doubling the cache sizes at each level, particularly for the L2 cache. For the 16-KB direct-mapped L1 cache used to generate the results of Section 5.4.1, 18 bits of tag are used per entry (assuming 32-bit addresses). Doubling this cache will result in 17-bit tags. Because the line size is 32 bytes, the total additional cost of the increased tag array will be $17 * 2^{10} - 18 * 2^{9}$, which is 1 KB.[1] In addition, an extra 16 KB of data is needed. Similar computations will show that the cost of doubling the 256-KB L2 cache is an extra 6144 bytes of tag and 256 KB of data. The total tag and data costs of the doubled L1 and L2 caches are shown in Table 5.4.

For a direct-mapped MAT with 4-bit access counters, 2-bit *decr_ctrs*, and 4-bit spatial counters, Table 5.5 gives the hardware cost of the data and tags for the 512-entry MATs utilized in Section 5.4.1. Since all addresses within a macroblock map to the same MAT

[1]These computations ignore the valid bit and other state for simplicity.

Table 5.4 Hardware cost of doubled caches.

Cache Level	Data Cost (bytes)	Block Size (bytes)	Sets	Tag Size (bits)	Tag Cost (bytes)
L1 Data	32K	32	1024	17	2176
L2 Unified	512K	256	2048	13	13312

Table 5.5 Hardware cost of 512-entry MAT with 2-KB macroblocks. Cost is same for both L1 And L2 cache levels.

MAT Entries	Data Cost (bytes)	Size of MAT Address (bits)	Tag Size (bits)	Tag Cost (bytes)
512	640	21	12	768

counter, a number of lower address bits are discarded when accessing the MAT. The size of the resulting *MAT address* for 2-KB macroblocks is shown in column 3 of Table 5.5.

Table 5.6 shows the data and tag array costs for the direct-mapped data caches used by the spatial locality optimization. The data cost remains the same as the base configuration cost, but the tag array cost is increased due to the decreased line sizes and additional hardware support for the scheme, which requires a 1-bit *fetch initiator bit* per tag entry.

The cost for the L1 buffer, which is a direct-mapped cache with 8-byte lines is shown in Table 5.7. As with the optimized data caches, the bypass buffers require a 1-bit *fetch initiator bit*, in addition to the address tag. The cost for the four-way set-associative L2 bypass buffer is computed similarly in Table 5.7.

Table 5.6 Hardware cost of optimized data caches.

Cache Level	Data Cost (bytes)	Fetch Size (bytes)	Block Size (bytes)	Sets	Tag Size (bits)	Tag Cost (bytes)
L1	16K	8/32	8	2048	18+1=19	4864
L2	256K	32/256	32	8192	14+1=15	15360

Table 5.7 Hardware cost of bypass buffers.

Cache Level	Entries	Fetch Size (bytes)	Block Size (bytes)	Data Cost (bytes)	Tag Size (bits)	Tag Cost (bytes)
L1	128	8/32	8	1K	22+1=23	368
L2	512	32/256	32	16K	20+1=21	1344

Table 5.8 Hardware cost of SLDTs.

Cache Level	SLDT Entries	Tag Size (bits)	Tag Cost (bytes)
L1	128	22+4=26	104
L2	128	19+5=24	96

The final component of the spatial locality optimization is the 32-entry SLDT, which can be organized as a direct-mapped tag array, with the *vc*, 1-bit *sz*, and 1-bit *sr* fields included in each tag entry. The L1 SLDT requires a 2-bit *vc* because there are four 8-byte lines per 32-byte maximum fetch, and the L2 SLDT requires a 3-bit *vc* due to the eight 32-byte lines per 256-byte maximum fetch. A bit mask could be used to implement the *vc*, rather than the counter design, to reduce the operational complexity. However, for large maximum-to-minimum fetch-size ratios, such as the 8-to-1 ratio for the L2 cache, a bit mask will result in larger entries. Table 5.8 shows the total tag array costs of the L1 and L2 SLDTs.

Finally, combining the costs of the MAT, the optimized data cache, the bypass buffer, and the SLDT results in a total L1 cost of 24 152 bytes with a 512-entry MAT. Therefore, the savings over doubling the L1 data cache is over 10 KB for the 512-entry MATs. Similar calculations show that the L2 optimizations save over 235 KB for the 512-entry MATs, more than doubling the L2 data cache. This translates into 31% and 45% less tags and data than the doubled caches at the L1 and L2 levels, respectively, for the 512-entry MAT used to generate the results of Section 5.4.1. Comparing the performance of the spatial locality and bypassing optimizations to the performance obtained by doubling the data caches at both levels, as shown in Figure 5.26, illustrates that for much smaller hardware costs, the integrated run-time cache optimizations usually outperform simply doubling the caches.

To reduce the hardware cost, the L1 MAT could potentially be integrated with the translation lookaside buffer (TLB) and page tables. For a macroblock size larger than or equal to the page size, each TLB entry will need to hold only one 8-bit counter value. For a macroblock size less than the page size, each TLB entry needs to hold several counters, one for each of the macroblocks within the corresponding page. In this case a small amount of additional hardware is necessary to select between the counter values. However, further study is needed to determine the full effects of TLB integration.

CHAPTER 6

CACHE REMAPPING

Cache conflict misses often account for the majority of misses incurred by an application. For example, Figure 6.1 shows the distribution of processor execution cycles spent stalled while waiting for a cache read miss to be serviced, using a direct-mapped 16-KB L1 data cache, a direct-mapped 32-KB L1 instruction cache, and a direct-mapped 256-KB L2 unified cache, for a number of integer programs. The L1 stall cycles are due to misses attributed to both the L1 data and instruction caches, and the L2 stall cycles are due to misses attributed to the L2 unified (both instruction and data) cache. The percentage of stall cycles incurred by conflict misses varies across the applications, but on average, conflict misses account for the most significant portion of the stall cycles. Also, the conflict miss stall cycles are dominated by those at the L2 cache level. Therefore, the reduction of conflict misses has been the focus of much work aimed at improving cache performance. The simplest conceptual way to reduce cache conflicts is to increase the associativity of the cache, allowing more data to map into the same cache set simultaneously. However, as the associativity increases, so does the time needed to perform associative searches and multiplexing [62], [63]. Combined with the shrinking processor cycle times, increasing the cache associativity may require more processor cycles per cache access. For example, Table 6.1 shows the cycle times calculated by CACTI [61], a model of estimating cache access and cycle times, for L2 caches of varying associativity. As the clock rate grows, the processor cycle times required by the different caches diverge.

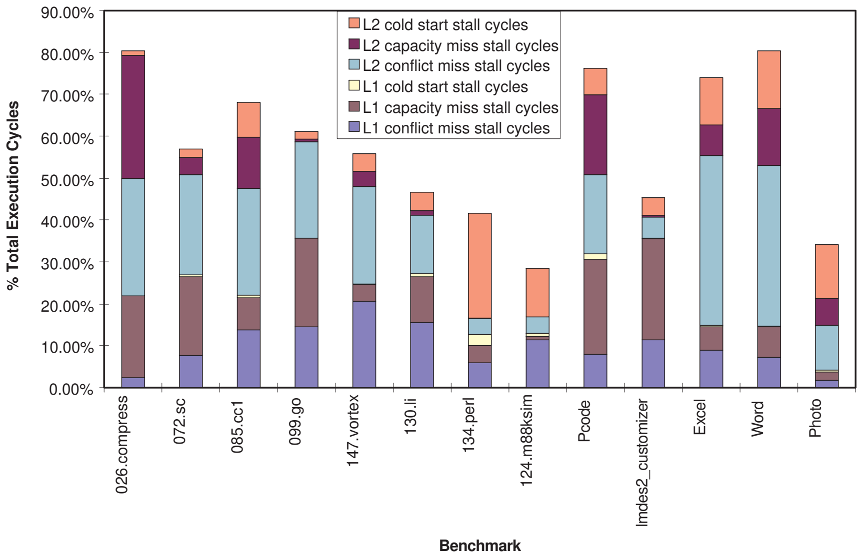


Figure 6.1 Cache stall cycle breakdown for a direct-mapped 16-KB L1 data cache, a direct-mapped 32-KB L1 instruction cache, and a direct-mapped 256-KB L2 unified cache. A stall cycle is attributed to the L2 cache if the requested address has had an L2 miss, otherwise it is attributed to the L1 cache.

Table 6.1 Cycle times for L2 caches of varying associativity, calculated by CACTI, and the resulting processor clocks for various processor speeds.

Cache Size (bytes)	Block Size (bytes)	Associativity	Cycle Time (ns)	200MHz clocks	300MHz clocks	500MHz clocks
256K	64	direct-mapped	13.31	3	4	7
256K	64	2-way	17.28	4	6	9
256K	64	4-way	18.60	4	6	10

As a result, some work has been aimed at improving the performance of direct-mapped caches. Several papers have examined improving data cache performance for numeric codes via static analysis and compiler optimizations [59], [64], [65]. However, these techniques have not been as applicable in integer codes. Other work examined a *virtual cache-unit technique*, whereby arrays are allocated to different, nonconflicting units of the cache [53]. The techniques were evaluated for a number of livermore loops and matrix multiply, and reduced the miss ratio in a direct-mapped cache. However, the allocation to the different units was performed by hand, and no compiler methods were proposed. Compiler optimization of code layout has been examined to improve instruction cache performance [48], [66], [67], [68].

Victim caches attempt to reduce conflict miss effects by saving data replaced from the data cache in a small fully associative cache [9]. The data is swapped back into the data cache on a hit in the victim cache. Victim caches tend to be small, 4 to 16 entries, due to the extra area necessary to implement the additional cache, and due to the time to perform a fully associative search and maintain LRU replacements. The mechanism presented in this chapter effectively allows portions of the L2 cache itself to assume the role of a victim cache. Selective allocation of victim cache entries was shown to give some improvement over traditional victim caches [57]. Bounce-back caches send data about to be replaced from the victim cache back to the main cache if its temporal bit is set [69]. However, the temporal bits are set by the load instructions, which must be marked as temporal through static analysis geared towards array accesses in numeric codes.

The skewed-associative cache [70] improves upon a conventional two-way set-associative cache when the associativity is implemented with multiple banks. Rather than use the same hash function $f(A)$ to compute the index into both banks for address A, the skewed-associative cache uses distinct functions $f_1(A)$ and $f_2(A)$ to compute two different indexes into the two banks. Functions $f_1(A)$ and $f_2(A)$ are chosen such that data conflicting in one bank will not be as likely to conflict in the other. Hit ratios close to those of a four-way set-associative cache are reported for a two-way skewed-associative cache, which has the access time of a two-way set-associative cache. This chapter utilizes the family of bitwise exclusive-or (XOR)-mapping functions introduced for the skewed-associative cache. However, they are augmented here with knowledge of under-utilized regions of the cache, and as a result, the performance of a four-way set-associative cache can be achieved with the access time of a direct-mapped cache.

Several papers [71], [72] examine the application of both the bitwise XOR-mapping functions and a polynomial XOR-mapping [73] function to several different cache organizations, including direct-mapped caches, two-way set-associative caches, two-way skewed-associative caches, column-associative caches, and victim caches. They show that both types of XOR mapping functions, particularly the polynomial mapping, greatly reduce the miss ratios in floating-point benchmarks by reducing conflicts. However, the miss ratios for integer benchmarks did not improve as significantly, and in half of the integer benchmarks the miss ratios increased. The polynomial mapping could be used in conjunction with the techniques presented in this chapter, rather than the mapping function that is currently employed.

Another paper examined the usage of bit-permutation mappings to help reduce conflict misses for regular numeric applications with constant-stride array accesses [33]. They explored three methods for performing the bit-permutation mappings: an entirely software scheme, where the compiler supplies information to the operating system which then remaps pages in memory; an entirely hardware scheme, where address mask generation and bit-permutation are done by the cache; and a hybrid software-hardware scheme. In the entirely hardware approach the cache is indexed twice, first with the standard bit-selection mapping, and second with the bit-permutation mapping. However, in order to perform remapping in their scheme, the cache must be able to detect constant-stride accesses. In integer codes, this approach will find fewer suitable remapping candidates.

Agarwal et al. proposed *pseudo-associative caches* called the *hash-rehash cache* [74] and the *column-associative cache* [32]. These caches are organized like a direct-mapped caches; however, each address can map into two locations: the normal line specified by traditional cache index computation (primary location), and an alternate location generally obtained by the application of a hash function to the original address (secondary location). Then, if there is a cache miss on the first direct-mapped access, the secondary location is accessed on the following cycle to check for a cache hit. Having two designated cache locations per address, a pseudo-associative cache is able to achieve hit ratios closer to that of a two-way set-associative cache. However, the tag comparison time is that of a direct-mapped cache. The disadvantage is that hits to the secondary location, as well as all cache misses, each require one extra cache access. If the overheads are low enough, meaning that the hit ratio is high and the majority of the hits are to the primary location, then the performance can be close to that of a two-way set-associative cache. If the two-way set-associative cache requires an extra cycle per access, due to the added complexity, then the pseudo-associative cache could perform better. Drach and Seznec proposed a *semi-unified cache*, where the alternate line for the direct-mapped L1 data cache line is the direct-mapped L2 instruction cache line, and vice versa [75]. The advantage is that the caches can remain direct-mapped with a higher overall hit ratio, however, a data path for swapping lines must be provided between the on-chip instruction and data caches.

The column-associative cache [32] requires the use of swapping between the primary and alternate locations, when a hit occurs in the alternate location. This maintains LRU ordering in order to reduce accessing overheads. Other multi-probed cache designs do not require swapping, but instead dynamically change the probing order. One design added an MRU bit to each cache tag, which was set on a hit to that line under bit-selection mapping (direct mapping), and cleared on a hit to the line specified by an alternate mapping function [76]. Then, the MRU bits were examined prior to accessing the cache to determine which set to probe first.

The disadvantage of this approach is that it will increase the access time, due to the need to examine the MRU bits before the cache access. The PSA-cache [77] alleviates this problem by using a steering bit table (SBT), indexed by information available earlier in the pipeline, to predict which pseudo-way to examine first. The information used to index the SBT included register contents, register number, and instruction address. Additionally, another table, called the MRU table, was used to implement LRU replacement among the pseudo-associative sets.

A number of designs have been proposed to reduce the data delay in set-associative designs [78], [79], [80], [81]. These designs optimistically return data from one line in the set, before the tag comparison is complete. A penalty is paid if the data is found in another line in the set. So and Rechtshaffen showed that the majority of the cache hits access the MRU lines in set-associative caches [79]. The System/370 organization for a multichip set-associative cache [78] provided the MRU block on the first cycle. If the tag comparison showed that the requested block was on another chip, it was provided on the following cycle. This design allowed most references to complete in one cycle and provided a reduced cycle time. Liu proposed instead to use a small portion of the address tags of each block to make the prediction, rather than the MRU information [80]. In the *DASC* cache [81], the data array is direct mapped, and the tag array is set associative. The primary direct-mapped location is optimistically returned on the first cycle, and if the set-associative tag entry indicates the data is elsewhere in the cache, it is supplied on a later cycle. To implement set-associativity in a direct-mapped data array, alternate locations via hashing functions similar to a column-associative cache are utilized. As in the column-associative design, data is swapped on hits to the alternate locations. The main advantage over the column-associative cache is the optimistic return of data, and the fact that greater than two-way set associativity can be implemented with less overhead, as the set-associative tag array indicates if and where data is located in the cache.

The half-and-half cache [82] divided the cache capacity between two smaller caches, one direct mapped and one set associative. Both are accessed simultaneously, and a hit in the direct-mapped cache results in a one cycle access, whereas a hit to the set-associative cache takes an extra cycle. The design presented in this chapter allows greater flexibility, because as much of the cache capacity can be direct-mapped as needed, and the amount of pseudo-associativity also varies based on the application behavior.

The cache miss lookaside (CML) buffer [83] monitors the conflicts occurring for each virtual memory page-size partition of a direct-mapped cache, called cache pages. When the CML buffer detects that a cache page is experiencing heavy conflicts, the operating system copies the contents of one or more of the conflicting pages to new physical pages such that the conflicts are avoided. Performance close to that of a two-way set-associative cache was reported.

A number of studies have examined cache-conscious page placement [84], [85], [86], [87], [88] by the operating system. These methods are orthogonal to the mechanism presented in this chapter, which optimizes cache placement at a finer-grain hardware level.

6.1 Dynamic Remapping Concept

While hit ratios for pseudo-associative caches have been reported to be close to those of two-way set-associative caches, these designs have a significant limitation, namely that the alternate cache location is determined through a static hash function determined at design time. For example, the column-associative cache [32] forms the alternate cache location index by flipping the most significant bit of the original index. The advantage of this type of remap location choice is that there is a given unique alternate location for each location in the cache, meaning that no additional information is necessary to find the alternate location. The disadvantage is that the alternate location could also be experiencing heavy conflicts, which would be exacerbated by remapping additional data into the location. Thus, static mapping functions have limited control over successfully increasing the cache performance.

The dynamic intracache remapping mechanism introduced in this dissertation also enables data to be remapped into alternate locations, which are called *remap locations*, when the original location specified by the normal cache mapping experiences conflict misses. As in the pseudo-associative caches, this mechanism first accesses the original location, followed by the remapped locations on successive accesses, in order to find the requested data. However, the mechanisms differ in terms of when data is chosen for remapping and on the choice of the remap locations. Minimal performance statistics are monitored and recorded in the cache tags, including information on the conflicts to the corresponding cache lines. Then data is remapped into alternate cache locations only when the original cache location has experienced recent conflicts. Also, the choice of alternate location is determined at run time, adapting dynamically to the usage patterns of the executing application. This is performed with the help of a small LRU stack of *cache segments*, which are uniform-sized contiguous regions of the cache. The LRU stack directs the remapped data into the less utilized portions of the cache. Then, the collected statistics are used to ensure that data is not remapped into heavily-utilized cache locations within the LRU segment. These techniques together provide a clear advantage over purely static remapping functions.

This chapter focuses on improving L2 cache performance. As shown in Figure 6.1, conflict misses at the L2 level are much more significant than those at the L1 level, which makes sense given that the L1 cache is smaller, and therefore more likely to be dominated by capacity misses. It is also critical to improve L2 cache performance in general, because the system bus latencies are large and growing. Also, given that the L2 cache is larger, it is more likely to have underutilized portions that can be identified and exploited by caching remapped data. Finally, because the methods proposed in this dissertation are more complex, and therefore can incur more overhead, the L2 cache is more likely to gain performance without suffering from the overheads introduced by remapping.

Previously published work on pseudo-associative caches has only examined one alternate location per cache line, meaning that they will achieve at most two-way pseudo-associativity. Here, cache locations experiencing heavy conflicts can obtain larger degrees of pseudo-associativity as needed. While this is theoretically possible with the column-associative cache, it was not studied because the researchers believed that the overheads would eliminate the benefit of any increase in hit ratio. The algorithm proposed here can help avoid unnecessary overhead. Results presented later in this chapter show that, when applied to a direct-mapped cache, this remapping technique can sometimes achieve the performance of a four-way set-associative cache.

6.2 Dynamic Remapping Techniques

Several issues need to be addressed in order to make cache remapping policies more adaptive and intelligent. These include when to remap data, how to select a remap location efficiently and intelligently, and how to efficiently maintain the remap locations. In general, the objective is to form *remap chains*, stemming from each heavily contended direct-mapped location in the cache, as illustrated in Figure 6.2. These chains have variable length and are traversed serially on an access, starting from the original line that an address maps to under direct mapping. The

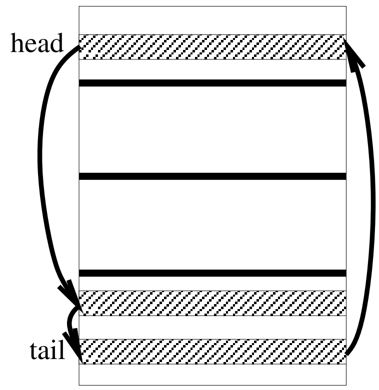


Figure 6.2 Illustration of a circular singly-linked remap chain containing three entries. All three entries map to the head location under direct mapping.

chains are traversed until a hit is obtained, or until a miss is verified after checking all elements. The run-time overhead of accessing the remap chains will be low if the hit ratio is high, and if the majority of the hits are to the head of the remap chain. This chapter will show that the cost in time and chip area of creating, maintaining and traversing these chains is minimal in comparison to the achieved performance.

Figure 6.3 shows the augmented tag entries used to support the remapping technique. The remap chain is a circular, singly linked list of cache locations, linked via the *nloc* (next location) fields. The size of the pointer is minimized using the cache segmenting techniques described in Section 6.2.2. The *rflags* field contains two bits of status, namely, whether this entry is a remap source or a remap destination or both. Throughout this dissertation it is assumed that $rflags = < rsrc, rdest >$. Section 6.2.1 discusses the *conf_ctr* field, and Section 6.2.3 discusses the *hit_ctr* field.

The rest of this section describes the mechanisms for determining which lines to remap, how the remap locations are chosen, the steps performed by the cache controller to remap data, the steps performed to unremap data, and finally, the steps taken to access remapped data requested by the processor.

1	c	2	s+x	1	1	32−l−b
hit_ctr	*conf_ctr*	*rflags*	*nloc*	valid	dirty	tag

Figure 6.3 Tag fields of the tag array entries augmented with support for dynamic cache remapping. The italicized fields are the new fields, and the number of bits per field is listed above each corresponding field, where c is the number of conflict counter bits, s is $\log_2$(number of cache segments), x is the number of bits computed by the XOR mapping described in Section 6.2.2, l is $\log_2$(number of lines in the cache), and b is $\log_2$(blocksize).

The hardware overhead of the dynamic remapping technique will be discussed in Section 6.4.

6.2.1 Remap decisions

As discussed in Section 1, data should only be remapped from a particular line in the cache when that line has experienced recent conflicts. Therefore, minimal performance statistics are monitored and recorded in the cache tags, including information on the conflicts to the corresponding cache lines. A small conflict counter, *conf_ctr*, is added to each cache tag, as shown in Figure 6.3. This saturating counter is incremented on each cache miss to that line, if that line held valid data. However, if this line has already been remapped, it is not known if there is a cache miss until all remap locations have also been checked. In order to avoid reaccessing the original tags after this search, the *conf_ctr* is incremented when the missing data is returned from main memory. At that point the *conf_ctr* is also examined to determine whether or not to remap the data. The comparison performed is:

$$conf\_ctr \geq t_1 \tag{6.1}$$

where t_1 is some integer value indicating the remap conflict threshold. In Section 6.3.1, the effects of different t_1 values will be examined. As will be described later in Section 6.2.3, the *conf_ctr* has another use when the corresponding line is chosen as a potential remap location for another line's data.

Various methods for decrementing the *conf_ctr* values have been investigated. The simplest method which performed well was to simply clear the corresponding *conf_ctr* when a line is remapped. Therefore, the actions taken by the cache controller on a cache line refill include performing the comparison of Equation (6.1), checking for a valid remap operation (to be explained in Section 6.2.3), and then either clearing or incrementing the corresponding *conf_ctr*, depending on whether the remap operation was valid or not, respectively. The action of accessing the original line to update *conf_ctr* is detailed in Section 6.2.4, and is an overhead modeled in the simulations discussed in Section 4.3.

6.2.2 Remap location selection

Data should be remapped into underutilized locations in the cache in order to reduce the possibility of displacing useful data and to increase the time before the remapped data is displaced by a primary access (original direct-mapped line access) to that location. However, finding the most underutilized location is expensive, so an intermediate solution is employed. The cache is divided into 2^s uniform-sized contiguous regions called *cache segments*. Then, if 2^s is small, a small LRU stack can be maintained, to indicate the LRU region of the cache. Because only the segment number needs to be stored in the LRU stack, the LRU operations can be implemented cheaply using a selective shifting technique, where the LRU "stack" shifts segment numbers, so that the LRU segment number can be easily located at the "LRU" end of the buffer. For example, the hardware must read the buffer contents, find the accessed segment, shift the more MRU segment numbers once towards the LRU end, and replace the MRU end with the accessed segment number. The buffer will require $s * 2^s$ bits, which would be 64 bits for 16 segments. The operation should be feasible to complete in one L2 cache cycle for small s. The LRU segment number then becomes the s most significant bits of the remap cache index. Not only can this technique reduce the number of bits needed in the *nloc* field of the tag entries, but as the results presented in Section 6.3.2 will show, there is little performance lost over a true fully associative LRU selection.

There are several options for choosing a location within the LRU segment. The first is to simply map to the same relative location within the new segment, leaving the $l - s$ least significant bits of the cache index intact, where l is $\log_2$(number of lines in the cache). This scheme is labeled *Direct*. In this case, *nloc* only needs to hold an s-bit segment number (x in Figure 6.3 is zero). However, all data conflicting for a single location in the cache using the normal bit selection mapping will vie for the same locations in all of the segments. For example, if the cache attempts to remap line l_1 from segment s_1, and if segment s_1 happens to be the LRU segment, Direct will attempt to remap to line l_1, and the remap will fail as described later.

Another possibility is to use an XOR-mapping [70], [71]. In this case an address $A = < a_n, ..., a_0 >$ can be viewed as $(A_4, A_3, A_2, A_1, A_0)$, where $A_4 = < a_n, ..., a_{b+2l-s} >$, $A_3 = < a_{b+2l-s-1}, ..., a_{b+l} >$, $A_2 = < a_{b+l-1}, ..., a_{b+l-s} >$, $A_1 = < a_{b+l-s-1}, ..., a_b >$, $A_0 = < a_{b-1}, ..., a_0 >$, and b is $\log_2$(blocksize). In other words, A_0 is the portion of the address used for word selection within a cache line, $A_2|A_1$ form the index of the direct-mapped cache line, A_2 alone is the segment number of the cache line, and $A_4|A_3$ is the tag. In the *Simple* scheme, the remap location is $A' = A_4|A_3|S|C|A_0$, where S is the LRU segment number and $C = A_3 \ xor \ A_1$. Therefore, the new offset within the segment is the XOR of the original offset A_1 and the least significant bits of the tag. This XOR operation disperses the accesses conflicting for the original location to different locations across each segment. However, because the $x = l - s$ least significant bits of the index are now also determined by the tag of the remapped data, the segment number is not enough information to uniquely identify the next location in the remap chain. Now *nloc* must hold the full $s + x = l$ bits of index. For a 256-KB cache with 64-byte lines and 16 segments, Simple will require a 12-bit *nloc* versus a 4-bit *nloc* for Direct. Section 6.3.2 will show that for 16 segments Simple does not give significant performance advantages over Direct, and therefore the cheaper Direct method is sufficient.

An intermediate possibility is to use only $x < l - s$ bits of the tag to XOR with x bits of the original index. This will allow data conflicting into the same direct-mapped location to map into at most 2^x distinct locations within each segment.

It should be noted that because these methods are only applied to the L2 cache, the addresses will always be physical addresses, and address translation is not an issue when the Simple scheme is utilized (in contrast to address hashing at the L1 cache level).

6.2.3 Remap operation validity

Once a remap location is chosen, the algorithm does not blindly remap data there. It first checks for the validity and desirability of remapping the conflicting data to that location. The necessary checks for validity are the following:

1. The remap location must not be the same as the original location.

2. The remap location must not be in any remap chain, meaning that $rflags = 00_B$.

The second check could be relaxed to verify only that the location is not the head of any remap chain. However, it has been found that the complexity of supporting the takeover of non-head remap chain entries for use in another chain does not provide substantial improvements.

Additionally, in order to ascertain whether it will be beneficial to remap the conflicting data into the selected remap location, the *conf_ctr* fields of the original location and the remap location are compared. If the *conf_ctr* of the original conflicting location is less than half the *conf_ctr* of the candidate remap location, the remap operation is aborted. Section 6.3.1 will examine the performance of different *conf_ctr* sizes.

However, the *conf_ctr* will not be large if a location is heavily utilized by accesses to one address which hits almost every time. In order to detect this situation, the one-bit *hit_ctr* is checked, and if its value is equal to one, the remap is aborted. The *hit_ctr* is set on a hit to that line, and cleared on every miss and every remap attempt to that line, giving an indication of whether there has been any recent temporal locality that might be destroyed by replacing that data with remap data.

6.2.4 Remapping steps

Using the mechanisms described in Sections 6.2.1-6.2.3, the steps taken when missing data is returned from main memory to the L2 cache are:

1. Access the normal direct-mapped cache line. If this line contains valid data and if Equation (6.1) is true, a remap operation is attempted. Otherwise, abort the remap operation. Simultaneously, determine the LRU segment number and form the remap location index. The direct-mapped cache tag and data must be saved in a buffer.

2. Access the candidate remap location tags and perform the checks described in Section 6.2.3. If any of the checks fail, abort the remap operation.

3. Now, the last location in the current remap chain must be found in order to add a link to the new remap location. The *nloc* fields are used to traverse the singly-linked list. If

the *rsrc* bit of the *rflags* field of the direct-mapped cache tag saved in the first step is set, use the saved *nloc* to access the tags of the next entry in the remap chain.

4. Repeat the last step with the newly accessed tag entry to find the next entry in the remap chain. Repeat this step until a tag entry with a zero *rsrc* bit of the *rflags* field is found. This is the last location in the current remap chain. Set the *rsrc* bit to one and the *nloc* field to the LRU segment number.

5. Set the remap location *nloc* field to the segment number of the accessed data (and therefore, of the direct-mapped location and head of the remap chain) and set the *rdest* bit. Also fill the tag and data from the buffer from the first step.

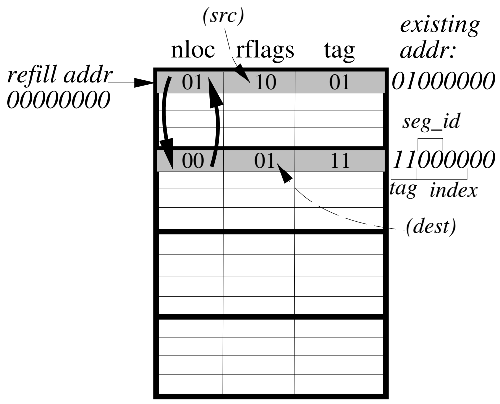
6. Finally, fill in the direct-mapped cache line with the missing data. Also clear the *conf_ctr* as described in Section 6.2.1.

Note that the L2 cache is assumed to contain a dedicated port for refills and the above operations. This is not unrealistic since dual-ported SSRAM has recently been introduced [89]. Also note that pipelined SRAM is assumed, with a tag array access time of T processor cycles and a total access time of D ($\geq T$). If $|L|$ is the length of the remap chain prior to performing this remap operation, then step 1 takes T cycles, step 2 takes T cycles, step 3 takes T cycles, step 4 takes $T * (|L| - 2)$ cycles, and steps 5 and 6 each take one cycle.[1] Therefore, if the remap operation is successful, the above steps consume roughly $T * (1 + |L|) + 2$ processor cycles. If the system bus latency is long and L2 cache misses are sufficiently apart in time, then the above operations will incur little noticeable overhead. If the remap operation is aborted, the missing data is simply written into the direct-mapped location, replacing its contents, and its *conf_ctr*

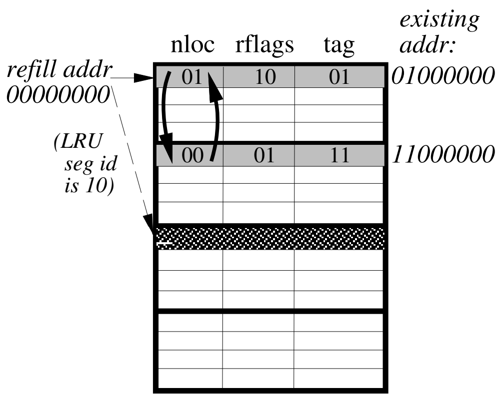
[1] This is somewhat aggressive as the tag writes are assumed to be pipelined at the processor clock rate. However, it is not unreasonable to assume that the tag writes could be buffered.

is incremented. The remap operation can be aborted after either step 1 or 2, in which case the operation takes $T + 1$ or $2 * T + 1$ processor cycles, respectively. Since the remap chain must have been previously traversed when the cache miss was first identified, it would seem possible to save the *nloc* of the last entry in the chain somewhere for use during the cache refill, in order to avoid traversing the entire chain again (steps 3 and 4). However, in a nonblocking cache it is possible that other data returning from main memory between the initial cache miss and the corresponding cache refill could modify the remap chain, resulting in a different tail location.

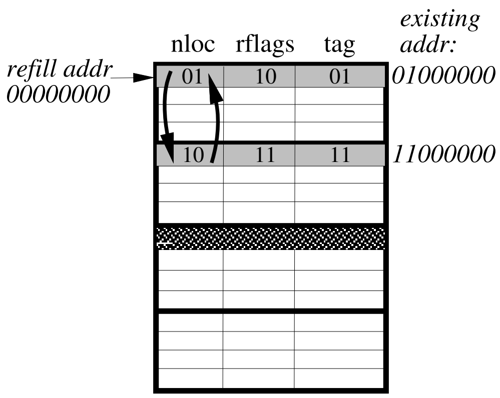
Figure 6.4 shows the steps outlined above, for an example cache refill using the Direct scheme of Section 6.2.2. Figure 6.4(a) shows the state of the remap chain originating at the direct-mapped location for the refill data. On the next cache access, shown in Figure 6.4(b), an access to the potential remap location is initiated, using the LRU segment number of 10_B, which results in an access to line 1000_B under the Direct scheme. Assuming that the conditions of steps 1 and 2 above are met, an access is made to the second location in the remap chain, using the saved *nloc* field of the direct-mapped access. Because that location (0100_B) is determined to be the current tail, due to $rflags = 01_B$, its *nloc* is set to the LRU segment number (10_B), and its *rsrc* bit is set, as shown in Figure 6.4(c). Finally, after the fill of the remap location with the data buffered from the direct-mapped read, and the fill of the direct-mapped location with the missing data, the resulting remap chain is shown in Figure 6.4(d). Notice that because of the desire to minimize swapping, only the direct-mapped location is moved, and it is moved to the new tail. Therefore, the algorithm does not quite maintain LRU replacement on the remap chain, however, the results presented in Section 6.3.2 will show that this algorithm reduces the overhead and sustains performance to a large extent, in comparison to true LRU.



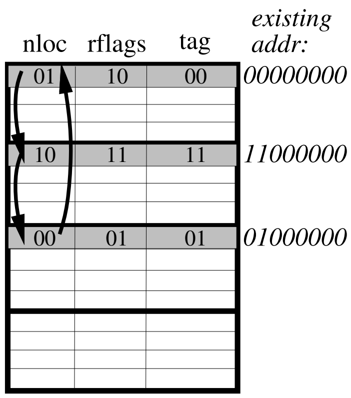
(a) initial direct-mapped access



(b) access of remap location using LRU segment number



(c) access of second entry in remap chain, setting *nseg* to LRU segment number



(d) state of cache when refill is complete

Figure 6.4 Remapping steps for refill of missing data at address 00000000_B, assuming a direct-mapped 16-entry cache with four segments, and 8-bit addresses (for simplicity) and the Direct scheme.

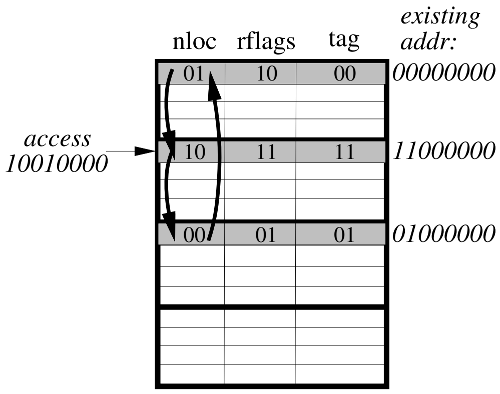
6.2.5 Unremapping data

On a cache refill, if *rdest* bit of the *rflags* field is set for the line accessed in step 1 of Section 6.2.4, then data from another cache location has been remapped into this location. Note that this access must normally be performed for writeback caches in order to handle dirty data displacement. In this situation the tag entry must be saved to a buffer, as in Section 6.2.4, and an unremap operation is performed before refilling this location with the missing data. In the first step, the cache controller traverses the circular list via the *nloc* fields, to find the location prior to the accessed location. Then, the *nloc* and *rsrc* bit of that location are set to the *nloc* and *rsrc* bits of the accessed location, which were buffered as described above. This effectively unlinks the accessed data from the remap chain.

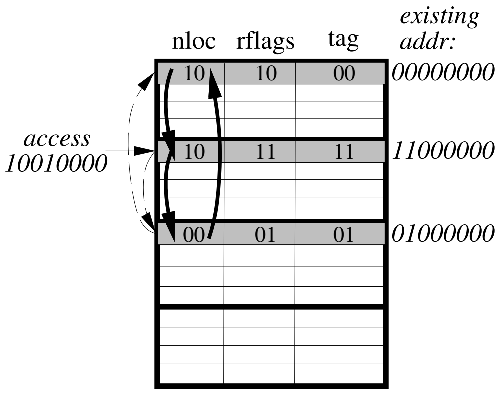
If the *dirty* bit of the location being unremapped is set, the *nloc* of the remap chain tail (i.e., the segment number of the remap chain head) is used to form the writeback address. The tail location is accessed during the above traversal, and is identified by an *rsrc* bit of zero.

The unremap operation will add $T * (|L| - 1)$ cycles to the normal refill operation, because a normal refill requires T cycles to check the tag and ensure that the displaced data is not dirty. Notice that the remap chain head will never be unremapped, because any line accessing that location on a cache refill will be linked into the remap chain stemming from that location, under the steps described in Section 6.2.4.

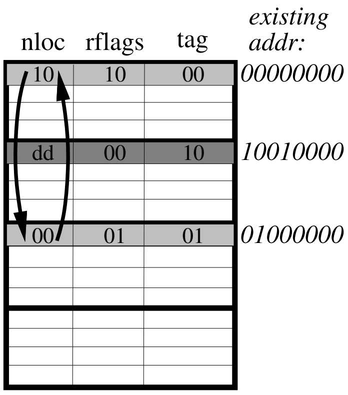
Figure 6.5 illustrates the steps taken on a cache refill to a line currently employed by a remap chain stemming from a different cache location. Assume that the data at that location is not dirty. Figure 6.5(a) shows that line 0100_B is accessed on the direct-mapped access for the cache refill data. Because the *rflags* indicate that this element is currently employed by another remap chain, the cache controller then traverses the chain, indicated by the dashed lines in Figure 6.5(b), until the entry with an *nloc* equal to the segment number of the refill address, in this case 01_B, is found. The *nloc* and *rsrc* bit of that entry are set to the corresponding values saved in a buffer from the direct-mapped access, in this case 10_B and 1_B, respectively. Finally, after the cache refill to the direct-mapped access is complete, the new cache state is shown in Figure 6.5(c), where the *nloc* value of 'dd' indicates that the value does not matter, since the *rflags* is 00_B.



(a) initial direct-mapped access



(b) search for previous entry in chain, setting its *nseg* to that of the direct-mapped access



(c) state of cache after unremap, showing two shorter remap chains

Figure 6.5 Unremapping steps for refill of missing data at address 10010000_B, assuming a direct-mapped 16-entry cache with four segments, and 8-bit addresses (for simplicity) and the Direct scheme.

No remap operation is attempted for the new data when an unremap operation is needed, because at that point the new data will be the only data in the cache mapping to that direct-mapped line.

6.2.6 Read and write operations

When servicing a read or write request from the L1 cache, the cache controller must check all entries in the remap chain stemming from the direct-mapped location, until either a hit is found or all entries in the chain have been checked. It is again assumed that each tag access takes T cycles, using the *nloc* and tag fields to form the address for the next access. If a miss is verified after checking all entries in the remap chain, a miss request is sent to main memory for reads, and the write data is sent to the write buffer for writes. If a hit is found in one of the remapped locations, a single swap is performed between the hitting remap location and the direct-mapped location. If the data and tags accessed in the original direct-mapped access are saved in some buffer, then the swap will take two additional cycles, one to write each location with new data, in a pipelined SRAM. Therefore, if the access hits at depth d in the remap chain, the access will require $T * (d - 1) + D + 2$ cycles, where D is the cycles to perform both tag and data accesses for the line that hits (i.e., D is the L1-to-L2 read latency). Note that this does not include the time to send the requested data back to the L1 cache, as that has not changed over a conventional cache, but only the time to read the requested data out of the data array at the L2 cache. If the access misses, the miss identification will occur in $T * |L|$ cycles.

As noted in Section 6.2.4, in order to minimize the required overhead, at most only one swap will be performed. While this does not maintain perfect LRU ordering in the remap chains, it is of less cost and performs almost as well, as will be shown in Section 6.3.2.

6.3 Experimental Evaluation

6.3.1 Remapping simulation

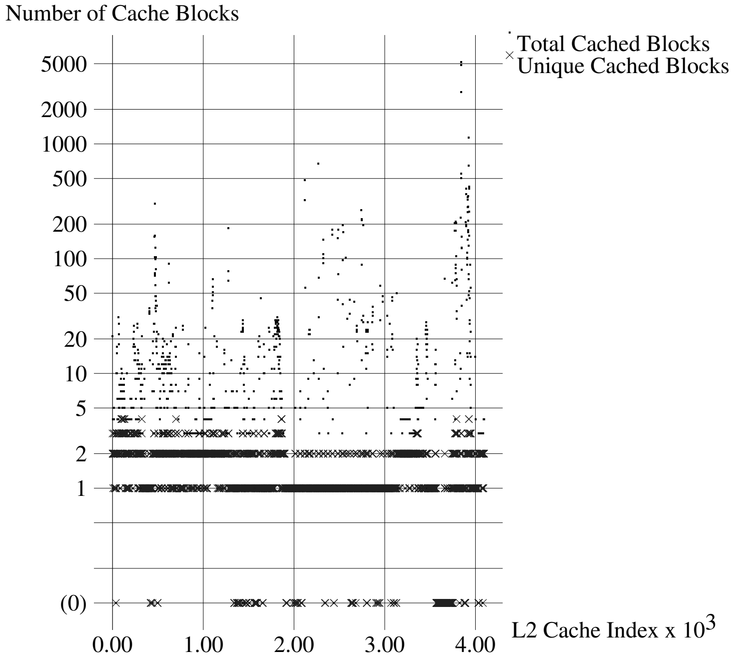
A detailed modeling of the remapping algorithm described in Section 6.2, including the noted overheads, was added to the simulator. It is assumed that the L2 cache is implemented with a pipelined synchronous SRAM. In general, the tag array can be implemented to require less access time than the data array. Therefore, since the L2 latency in processor cycles (D) is four, as shown in Table 3.3, T is assumed to be three cycles. Thus, the time between accesses when traversing the remap chain is modeled to be $T = 3$ processor cycles, and the L2 cache is blocked throughout each remap chain traversal described in Section 6.2.6.

The maximum length of a remap chain is limited to four in most of the experiments. However, it will be shown that the performance when allowing unlimited-length remap chains is roughly the same.

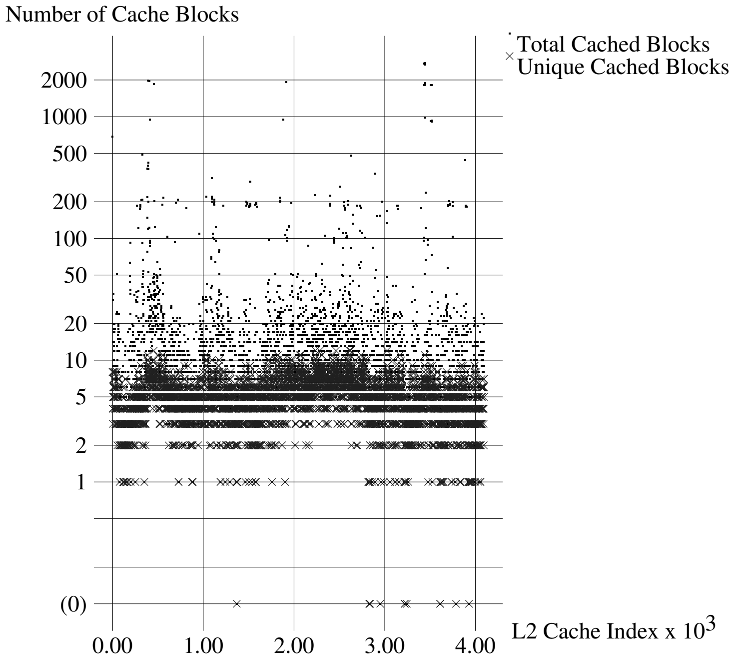
6.3.2 Results

Figure 6.6 shows the usage of each L2 unified cache set across the execution of two programs for the base L2 cache, in terms of both the total number of blocks cached in each set, and the number of unique blocks cached in each set. Figure 6.6(a) shows that *099.go* has at most four unique blocks that map to any one set across program execution. However, for most sets the total blocks cached (i.e., the number of misses) is much larger than the unique blocks cached, illustrating that the few blocks mapping to each set are conflicting heavily. There are also sets that did not cache any blocks, suggesting that the the scheme should find underutilized locations in which to remap the conflicting data. Figure 6.6(b) shows that Excel caches up to around 10 unique blocks per set. There are also fewer unutilized sets. However, when the cache usage was examined for each single 500 000-instruction execution phase, the number of unique blocks decreased, and the number of unutilized sets was large. The dynamic scheme presented in this chapter will adapt to this varying cache utilization.

Figures 6.7-6.12 show the performance improvement of the dynamic L2 remapping scheme using 16 cache segments and the Direct selection method, over a conventional direct-mapped cache, for one to six-bit *conf_ctrs*. The graphs also examine the performance effects of different



(a) 099.go



(b) excel

Figure 6.6 The total number of blocks and the number of unique blocks cached at each line in the direct-mapped L2 cache across the execution of two programs. The y-axis is plotted on a log scale, with values of zero added at the bottom.

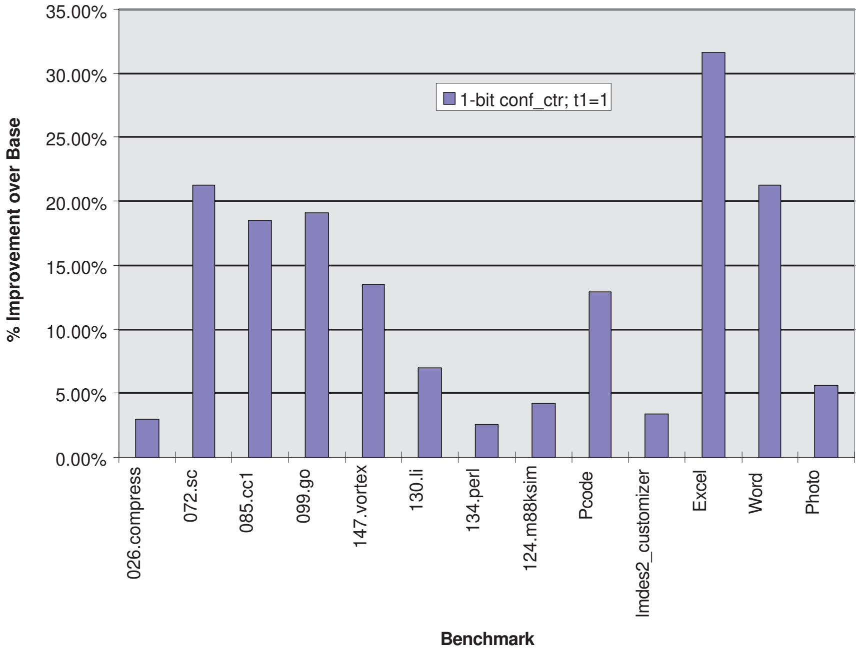


Figure 6.7 Performance for the dynamic L2 remapping scheme using 16 cache segments and the Direct selection method, for various t_1 thresholds and a 1-bit *conf_ctr*. Performance is shown as the percent improvement over a direct-mapped L2 cache.

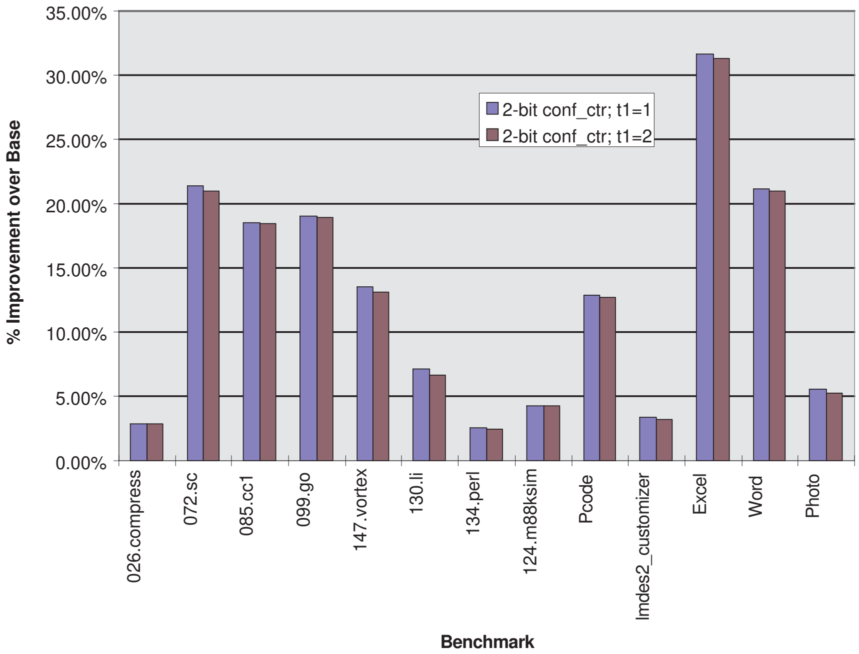


Figure 6.8 Performance for the dynamic L2 remapping scheme using 16 cache segments and the Direct selection method, for various t_1 thresholds and a 2-bit *conf_ctr*. Performance is shown as the percent improvement over a direct-mapped L2 cache.

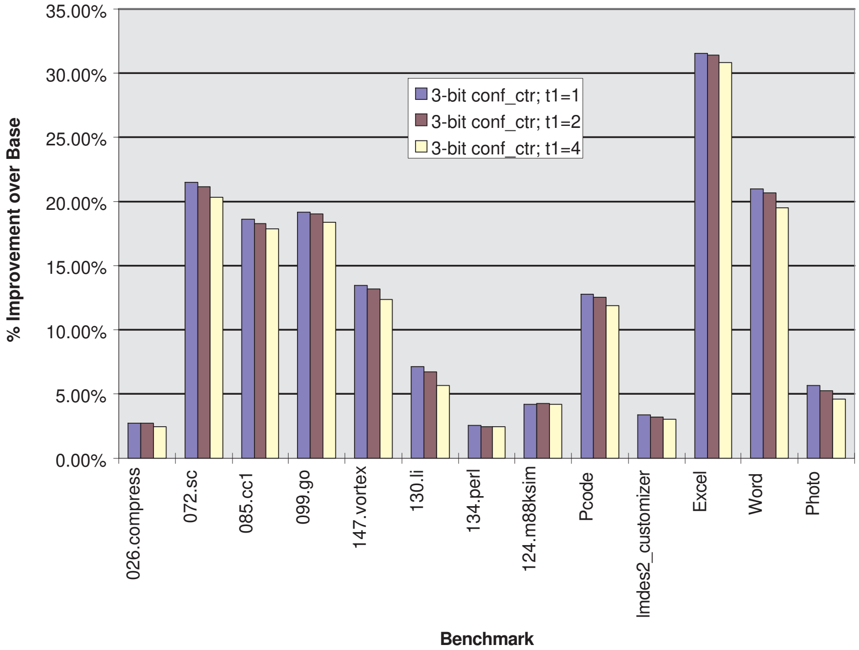


Figure 6.9 Performance for the dynamic L2 remapping scheme using 16 cache segments and the Direct selection method, for various t_1 thresholds and a 3-bit *conf_ctr*. Performance is shown as the percent improvement over a direct-mapped L2 cache.

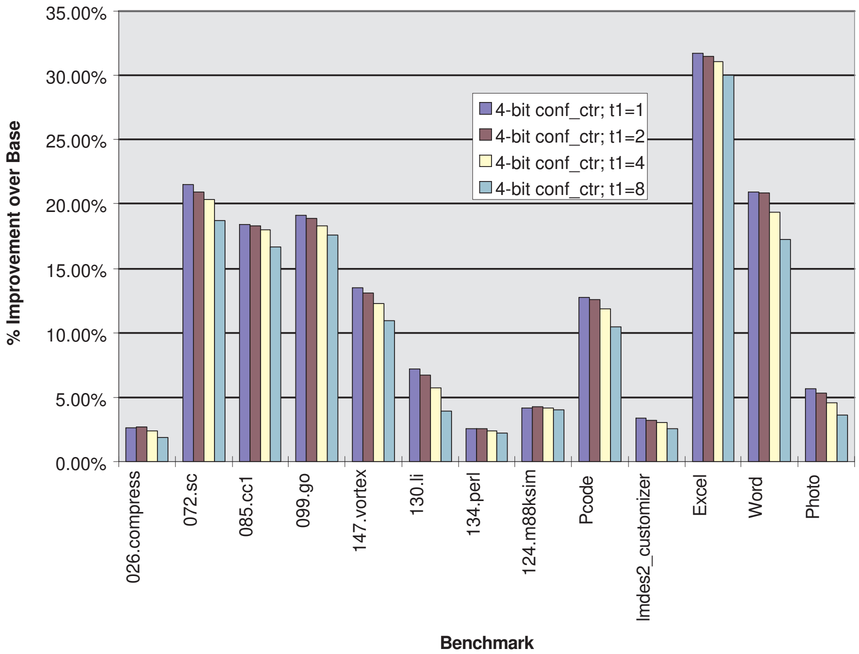


Figure 6.10 Performance for the dynamic L2 remapping scheme using 16 cache segments and the Direct selection method, for various t_1 thresholds and a 4-bit *conf_ctr*. Performance is shown as the percent improvement over a direct-mapped L2 cache.

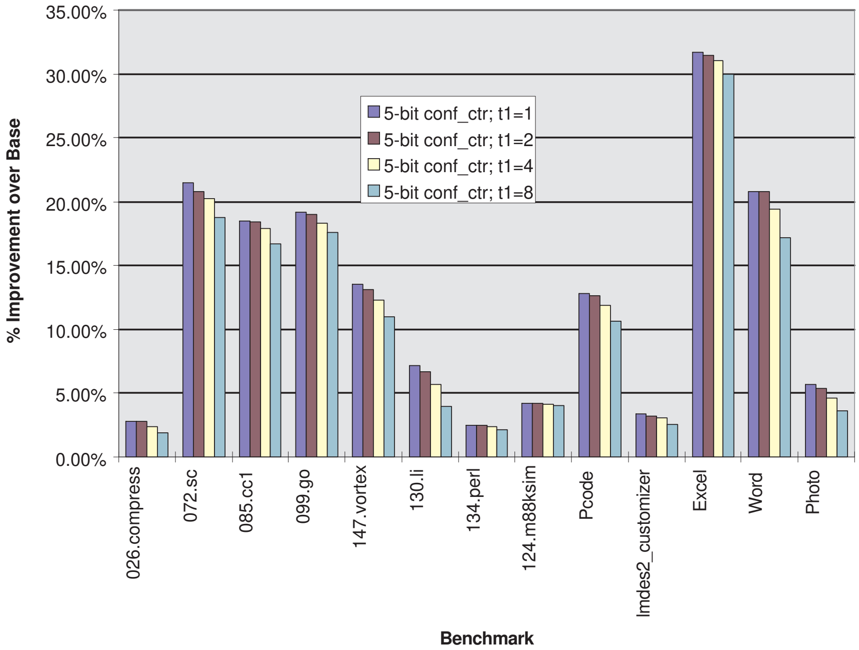


Figure 6.11 Performance for the dynamic L2 remapping scheme using 16 cache segments and the Direct selection method, for various t_1 thresholds and a 5-bit *conf_ctr*. Performance is shown as the percent improvement over a direct-mapped L2 cache.

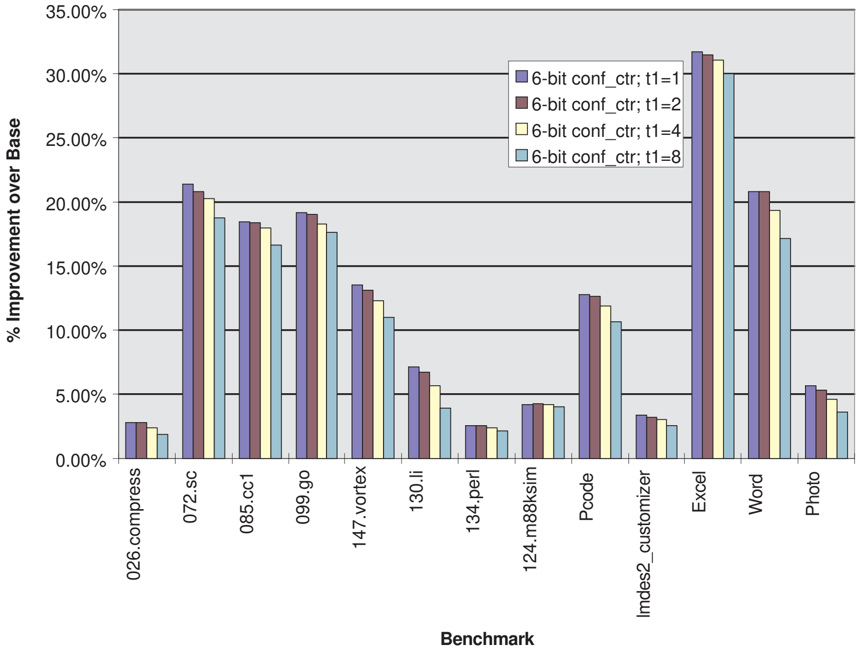
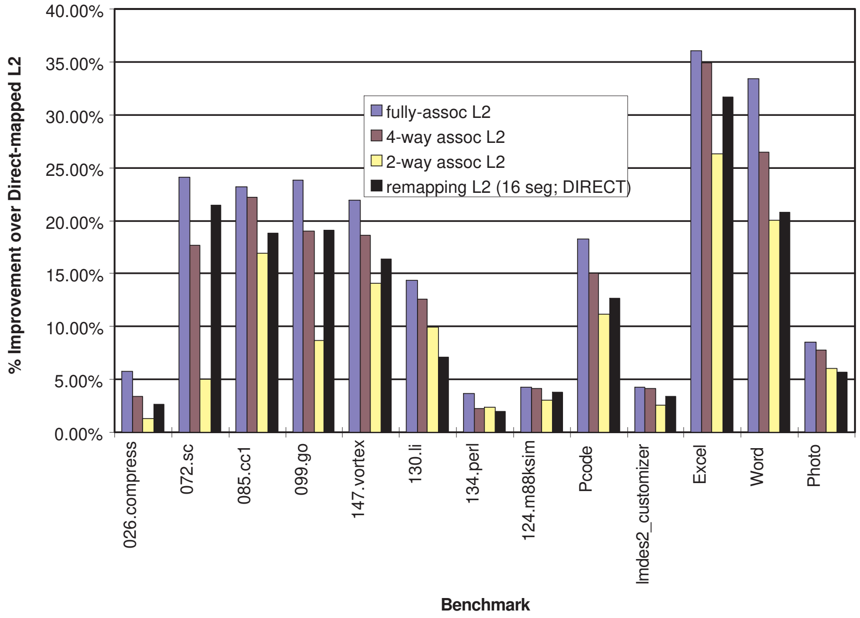


Figure 6.12 Performance for the dynamic L2 remapping scheme using 16 cache segments and the Direct selection method, for various t_1 thresholds and a 6-bit *conf_ctr*. Performance is shown as the percent improvement over a direct-mapped L2 cache.

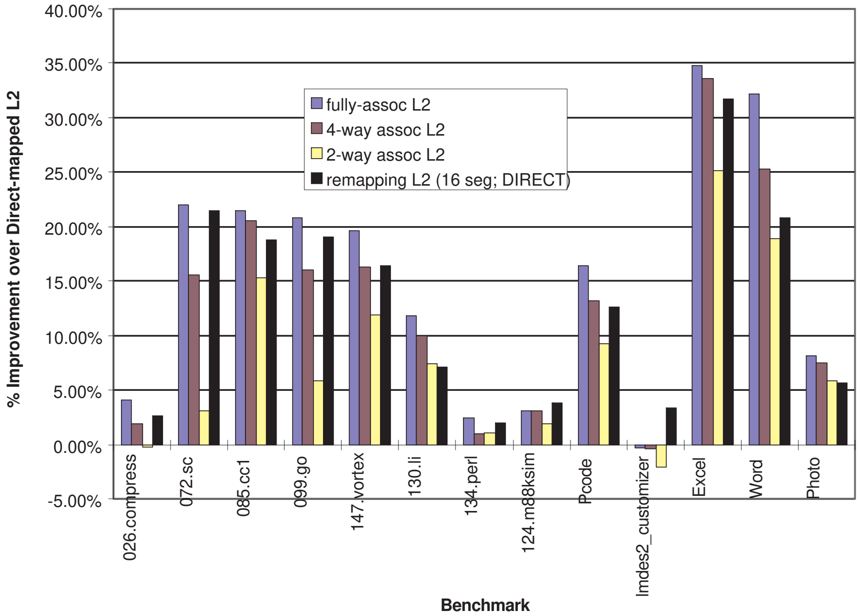
values of t_1 from Equation (6.1). The t_1 values examined are one, two, four, and eight. Some graphs do not show results for all four t_1 values because the corresponding *conf_ctr* size is too small for one or more of the larger t_1 values. The results show that the performance almost always decreases when increasing the t_1 value, regardless of the *conf_ctr* size. This means that aggressive remapping is desired, only preventing those lines with no recent conflicts from being remapped.

Although a t_1 value of one is best, larger *conf_ctr* values may be preferred to give increased resolution in the counter comparisons described in Section 6.2.3. The comparisons are made on remap attempts to ensure that data is not remapped to a more contended location. However, a comparison of the improvements obtained by the different *conf_ctr* sizes shows that the performance is generally unchanged with t_1 values of one. A reason for this trend will be demonstrated later in this section. Therefore, to minimize hardware overhead a one-bit *conf_ctr* is sufficient.

Figure 6.13(a) shows the performance improvement over a direct-mapped 256-KB unified L2 cache achieved by the dynamic remapping scheme, using 16 cache segments and the Direct remap location selection scheme described in Section 6.2.2. Also, 5-bit *conf_ctrs* and $t_1 = 1$ are utilized. Also shown for comparison are the performance improvements achieved by fully, four-way, and two-way set-associative caches of the same size. In all cases, the L1-to-L2 latency was the same: four cycles. As shown in Figure 6.13(a), the dynamic remapping techniques outperform a 2-way set-associative design in all but a few cases. In some cases, the dynamic remapping scheme performs close to a 4-way set-associative design, and even exceeds it in the case of *072.sc*.



(a) same L2 latencies



(b) set-associative 1-cycle longer L2 latency

Figure 6.13 Performance for the dynamic L2 remapping scheme using 16 cache segments, the Direct selection method, 5-bit *conf_ctrs* and $t_1 = 1$, in comparison to set-associative L2 caches. Performance is shown as the percent improvement over a direct-mapped L2 cache. Results are shown for scenarios where the set-associative caches have the same access time as direct-mapped, and where they incur one extra processor cycle per access due to slightly longer access times.

Table 6.2 Average remap chain length using 16 cache segments and the Direct selection method. Also, 5-bit *conf_ctrs* and $t_1 = 1$ are utilized.

Benchmark	Ave Length
026.compress	2.00
072.sc	2.05
085.cc1	2.11
099.go	2.06
147.vortex	2.22
130.li	2.03
134.perl	2.08
124.m88ksim	2.01
Pcode	2.06
lmdes2_customizer	2.01
Excel	2.18
Word	2.22
Photo	2.24

Figure 6.13(b) shows the performance of the same configurations, with a slight modification to the set-associative caches. In this case, the effects of a slightly longer access time incurred by the set-associative designs is shown, resulting in one-cycle longer L1-to-L2 latency for those caches. As demonstrated in Table 6.1, more processor cycles are required to access the set-associative caches, even at current processor speeds. Because the remapping scheme uses a basic direct-mapped design, and adds the overheads only when required by the remapping algorithm, it avoids the need for adding associative searches and LRU replacements to every access. Therefore, the base L2 latency should not increase for remapping. As shown in Figure 6.13(b), the advantage of remapping over set-associative caches grows when the longer set-associative latencies are taken into account. Remapping even performs better than a fully associative cache in the case of *124.m88ksim* and *lmdes2_customizer*.

Table 6.2 shows the average length of the remap chains formed in the dynamic remapping scheme used for the results in Figure 6.13. The average was calculated only for those chains of length greater than one, meaning that there was some data remapped. The average was calculated as follows. If the number of blocks remapped to depths two through four are N_2, N_3

and N_4, respectively, then the average length is:

$$Ave = ((N_2) * 2 + (N_3) * 3 + (N_4) * 4)/num\_chains \tag{6.2}$$

where

$$num\_chains = N_2 + N_3 + N_4 \tag{6.3}$$

The number of chains to any one depth is equal to the number of blocks remapped to that depth, even if that chain eventually grows longer. The reason for counting the intermediate subchains as separate chains is that it avoids complications in the face of unremapping. For example, if a block is remapped to depth three, and an unremap operation occurs, backing the chain to length two, followed by another remap to depth three, the above average will count two chains of length three from this sequence. Even though the chains have average lengths of just over two, the results of Figure 6.13a show that performance often exceeds that of a two-way set-associative cache. For example, the average length of the remap chains formed for *099.go* is only 2.06 (and this average does not count those locations from which no remap chains were formed, meaning that their lengths were one). However, even though the access time of the set-associative caches in Figure 6.13(a) are those of the direct-mapped cache with remapping, and the remap chain traversals incur penalties, remapping is able to achieve the performance of a four-way set-associative cache. The reason for this performance is twofold. First, the dynamic remapping technique utilizes pseudo-associativity only when the gathered conflict statistics indicate a need, thereby allowing more of the cache capacity to hold other data. Second, through the use of the LRU segments, the dynamic remapping technique is able to use less-utilized cache regions for the pseudo-associativity. It is for the above reasons that the dynamic remapping will also outperform column-associative caches, which at most can obtain two-way set-associative performance.

Figure 6.14 shows a breakdown of the failed remapping attempts by the reason for the failure, shown as a percentage of total remapping attempts. The most frequently occurring reason for aborting a remap operation is that the candidate remap location had a *hit_ctr* of one, meaning that that location had experienced a potentially recent hit, indicating temporal locality that might be destroyed by replacing the cached data with remap data. The second most common reason for aborting a remap operation is that the selected candidate remap location was already utilized in another remap chain. Figure 6.14 demonstrates that few remaps were prevented by candidate remap locations with larger five-bit *conf_ctr* values. This is one reason why the performance varies little across different *conf_ctr* sizes, as noted earlier from the comparisons of Figures 6.7-6.12.

A large percentage of the remapping attempts failed. However, if more aggressive remapping is allowed, a larger number of new cache misses might be introduced at the target remap locations. Figure 6.15 shows the percentage of cache hits in the base configuration that become misses with remapping. For most of the benchmarks, only 0.5% of the cache hits become misses with remapping. Although over 2% of the cache hits become misses with remapping for *026.compress* and *Pcode*, later results will demonstrate that a net gain in cache hits is achieved.

Figure 6.16 shows the effects of varying the number of cache segments from 4 to 64. In general, the performance increases with increasing numbers of segments. This result occurs because with more segments, and therefore smaller segments, the granularity at which under-utilized segments are detected grows. Past approaches, such as the column-associative cache, effectively have only 2 segments. In most cases, 16 segments achieve performance very close

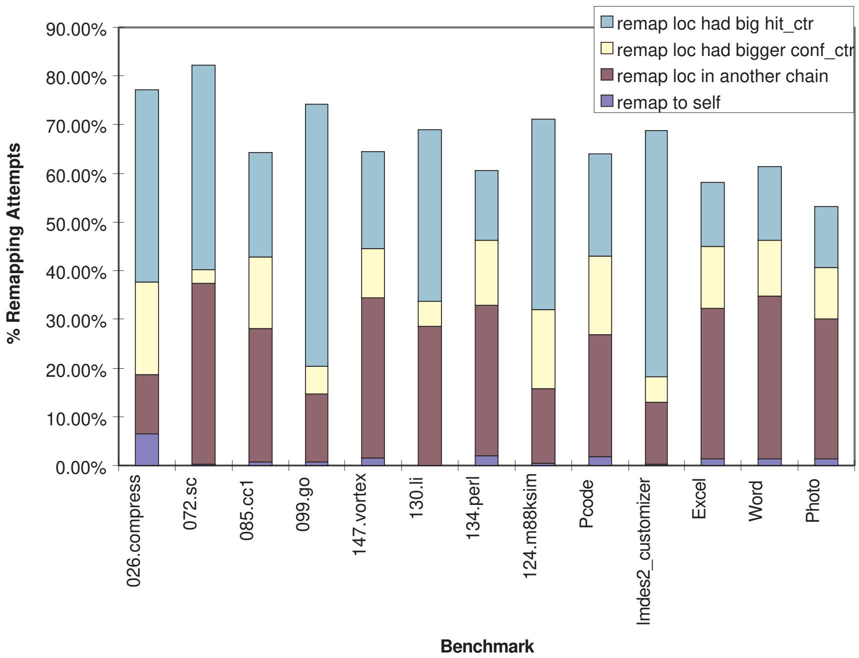


Figure 6.14 Breakdown of failed remapping attempts by the reason causing the remap operation to abort, shown as a percentage of total remapping attempts, using 16 cache segments and the Direct selection method. Also, 5-bit *conf_ctrs* and $t_1 = 1$ are utilized.

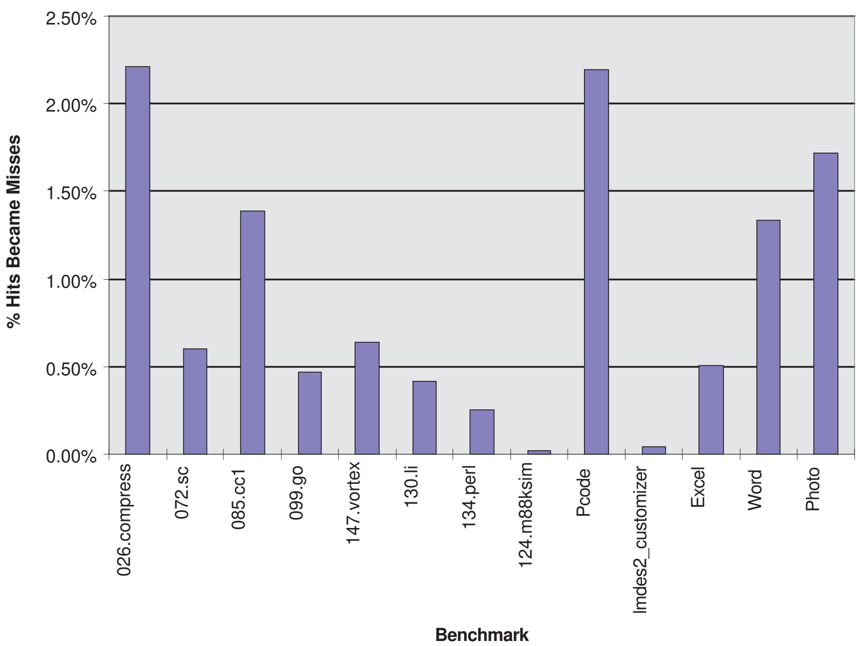


Figure 6.15 Percentage of base configuration cache hits that become misses with remapping, using 16 cache segments and the Direct selection method. Also, 5-bit *conf_ctrs* and $t_1 = 1$ are utilized.

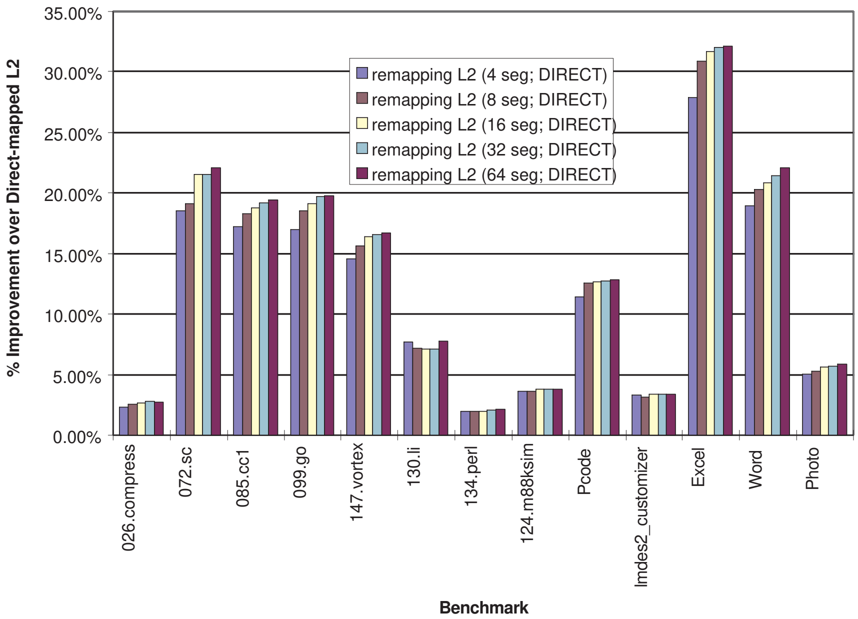


Figure 6.16 Performance of the dynamic L2 remapping scheme for different numbers of cache segments. Performance is shown as the percent improvement over a direct-mapped L2 cache.

to that of 64 segments, but at a lower cost in both tag array area and LRU segment detection complexity.

Figure 6.17 shows the effects of several variations in the remapping scheme. The leftmost data points are the performance improvements shown in the earlier graphs, using 16 segments and Direct remap location selection. Also, five-bit *conf_ctrs* and $t_1 = 1$ are utilized. The second set of data shows the performance improvement when ideal LRU ordering is maintained in the remap chains. In this case no penalties are modeled for maintenance of the LRU ordering (i.e., no swapping penalties are incurred). These results show that the ideal LRU ordering with no penalties does not perform much better than the pseudo-LRU ordering described in Section 6.2. Therefore, the pseudo-LRU ordering maintained with single swaps is sufficient.

The third set of data shows the performance achieved by selecting a remap location via a full LRU search of the cache, rather than by the techniques described in Section 6.2.2. For

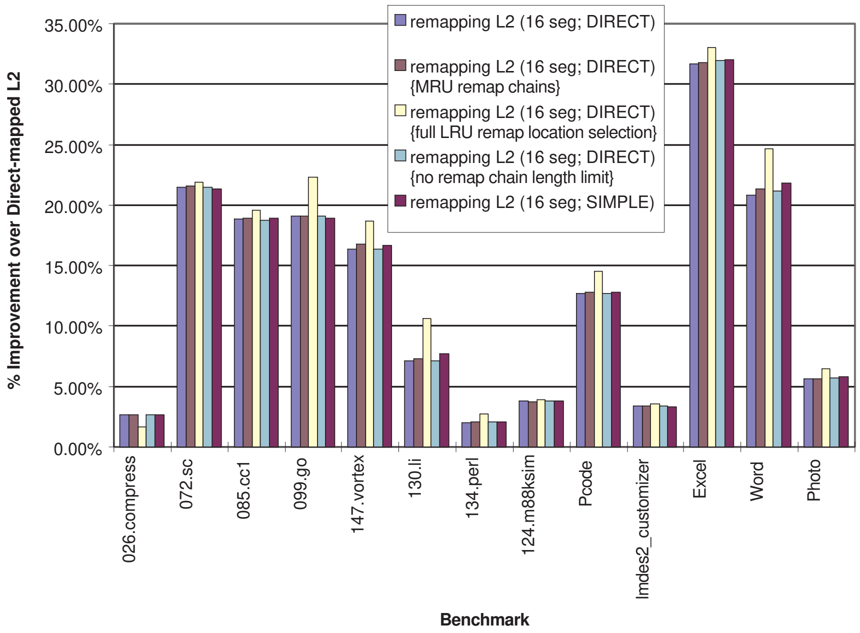


Figure 6.17 Performance of the dynamic L2 remapping scheme when various remapping parameters are changed. Shown are the effects of maintaining MRU ordered remap chains with no overhead, performing a full LRU search for a candidate remap destination, allowing unlimited remap chain lengths, and using the Simple remap location selection strategy. Performance is shown as the percent improvement over a direct-mapped L2 cache.

some benchmarks, such as *099.go*, *130.li*, and Word, a full LRU search (if even possible) would provide significantly better performance. However, for most benchmarks, the segment/Direct remap location selection algorithms perform almost as well, and even do better for *026.compress*.

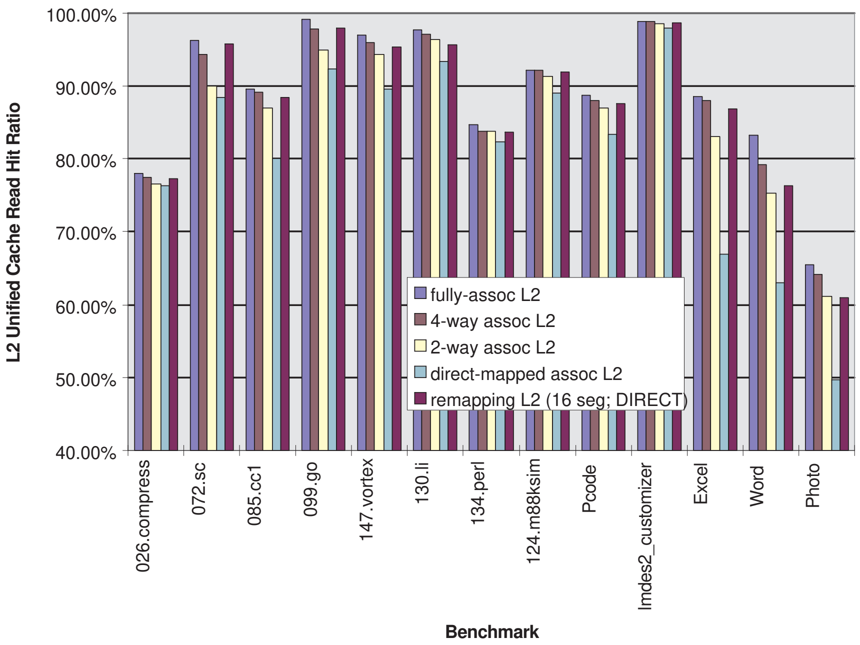
The fourth set of data shows the performance when there is no limit placed on the length of the remap chains. As described in Section 6.3.1, for the other remapping results a maximum length of four was enforced. The results show that there is little noticeable difference in performance between the results when a length of four is enforced, versus when no maximum length is enforced. In fact, Table 6.3 shows that the maximum length of a remap chain created in the latter case is less than or equal to four for many benchmarks, and not much more than four for the others.

Table 6.3 Maximum remap chain length.

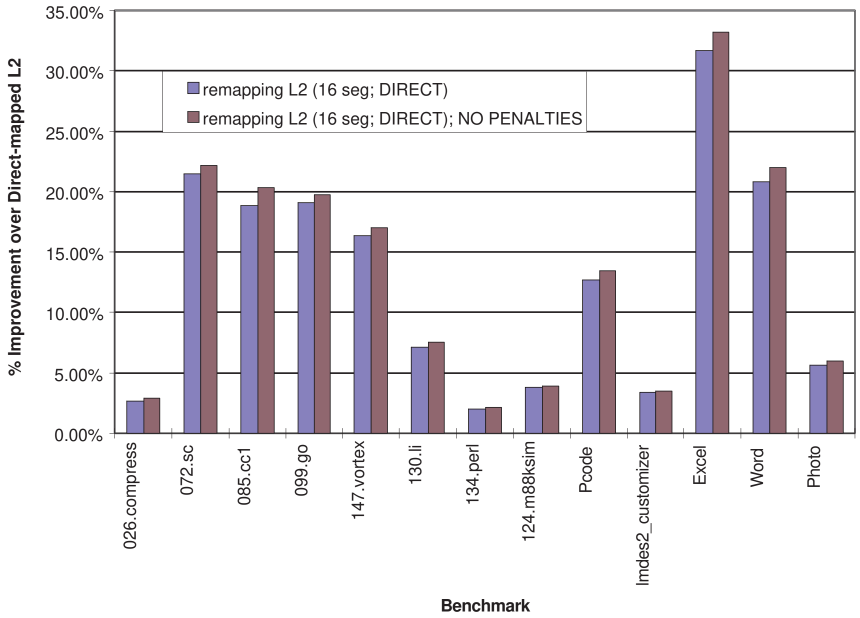
Benchmark	Max Length
026.compress	3
072.sc	3
085.cc1	5
099.go	3
147.vortex	6
130.li	4
134.perl	6
124.m88ksim	3
Pcode	5
lmdes2_customizer	3
Excel	5
Word	6
Photo	6

Finally, the last set of data shows the performance achieved by the Simple remap location selection strategy described in Section 6.2.2. Although this scheme would allow multiple conflicting locations to remap into one segment, it does not achieve better performance. Therefore, the less costly Direct method is preferred. As the breakdown shown in Figure 6.14 illustrates, few remaps were prevented by the Direct method selecting the remap source as the remap destination, which the Simple method would have alleviated. The Simple method could also potentially reduce the amount of remap attempts that failed due to selecting a candidate destination already utilized in another remap chain. In most cases, however, no reduction occurred.

Figure 6.18(a) shows the read hit ratios for the L2 unified cache achieved by the dynamic remapping scheme, with 16 segments, Direct remap location selection, five-bit *conf_ctrs* and $t_1 = 1$, as well by fully associative, four-way set-associative, two-way set-associative, and direct-mapped designs. The dynamic remapping scheme achieves hit ratios the same as or higher than the two-way set-associative cache in all cases except *130.li*. In many cases, remapping achieves hit ratios the same as or higher than the four-way set-associative cache, and even achieves roughly the same hit ratio as fully associative in several cases. However, the run-time



(a) L2 unified cache read hit ratio



(b) examination of remapping overheads

Figure 6.18 Comparison of remapping hit ratios to those of L2 caches with varying set-associativities for the remapping scheme with 16 cache segments and the Direct method. Also, 5-bit *conf_ctrs* and $t_1 = 1$ are utilized. Also shown is a comparison of the performance of the same remapping scheme both with and without remapping overheads.

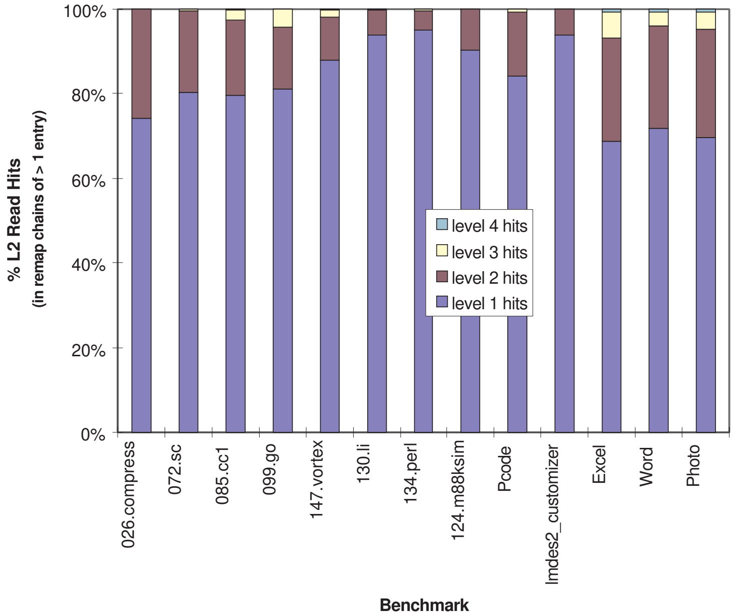


Figure 6.19 Breakdown of the remap chain depth at which L2 read hits occurred in chains with more than one entry.

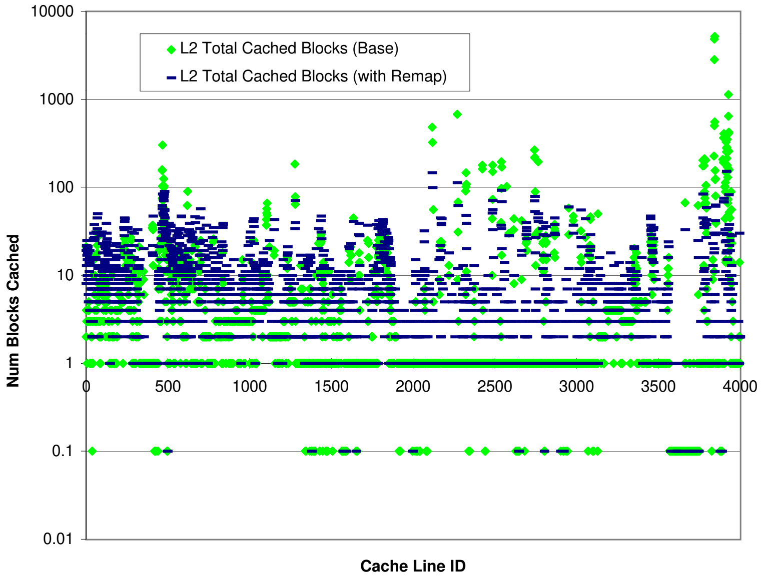
overheads incurred by remapping result in the somewhat lower relative performance shown in Figure 6.13(a). The effect of this overhead can be seen in Figure 6.18(b), which shows the performance improvements obtained by the remapping scheme both with and without simulating the run-time remapping overheads.

Figure 6.19 shows the depth at which read hits occurred in remap chains of lengths greater than one. The results show that the majority of the hits are to the first entry (the original direct-mapped, or primary, entry). The next largest amount of hits occurs at the second entry in the chain. This suggests that the pseudo-LRU ordering maintained in the remap chains by the use of single swaps is sufficient, a conclusion already suggested by the results for perfect LRU ordering shown in Figure 6.17. Another explanation is that many of the chains formed are a length of two, which is also suggested by the relatively short maximum lengths reported in Table 6.3.

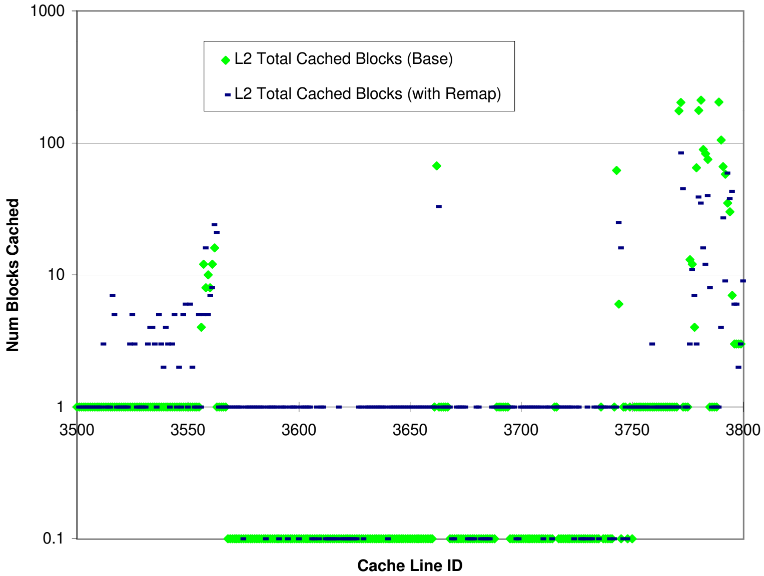
Figure 6.20 shows an illustration of the improvements in cache utilization, for *099.go*. Figure Figure 6.20(a) shows the number of blocks cached, or the number of cache misses, at each line in the L2 cache, for lines 0-4000 of the 4096 lines. Results are shown for both the base configuration and remapping, with 16 cache segments and Direct location selection. The y-axis is a $\log_{10}$ scale of the number of blocks cached over the entire program execution. Therefore, any line with zero blocks cached was forced to a y value of 0.1 so that the utilization could be plotted on the log scale. The base configuration often results in significant hot spots, or contention, at a number of lines in the cache, resulting in the number of blocks cached exceeding 100 and even 1000. With remapping, the utilization is evened, and only a few lines cache slightly over 100 blocks.

Examining the unutilized lines in the base configuration, the y values of 0.1 in Figure 6.20(a), shows that often these lines were found and exploited with remapping. This is illustrated by the fact that fewer lines are unutilized with remapping. However, the range of lines from rougly 3500-3800, which were unutilized in the base configuration, still look unutilized with remapping. Therefore, Figure 6.20(b) shows a closer view of those cache lines. The results clearly show that a large number of the unutilized lines in that range were indeed found and exploited with remapping.

Figure 6.21 shows a comparison of the performance improvements obtained for the remapping scheme with 16 cache segments, the Direct method, five-bit *conf_ctrs*, and $t_1 = 1$, when utilizing the uniform sampling methods described in Chapter 3 and when simulating the benchmarks completely. The results show that in most cases the performance improvement obtained with uniform sampling, utilized to reduce simulation time, is very close to that obtained by full simulation. In a few cases the difference is more significant; however, the sampled results



(a) cache lines 0-4000



(b) cache lines 3500-3800

Figure 6.20 Cache utilization for *099.go*, for both the base configuration and the dynamic L2 remapping scheme using 16 cache segments, the Direct selection method, 5-bit *conf_ctrs* and $t_1 = 1$.

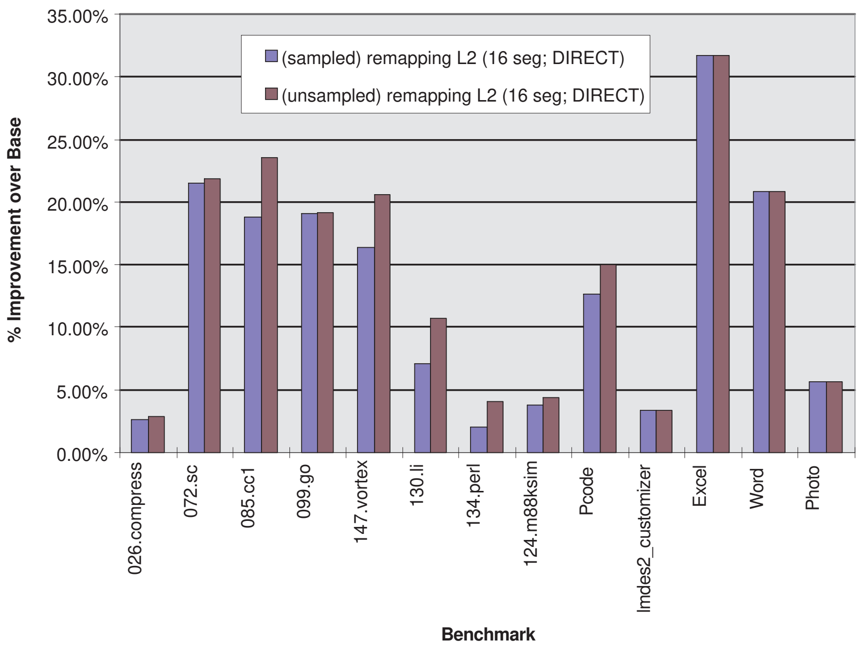


Figure 6.21 Comparison of performance improvements achieved using the sampling methodology described in Chapter 3 to that achieved when completely simulating the benchmarks.

always underestimate the true performance gain obtained by the remapping optimization. As with the same comparison for the cache bypassing optimization (shown in Figure 4.26), the underestimation is due to the fact that the statistics gathered in the cache tags reflect recent accessing behavior. When jumping to a new sample, these statistics are less likely to be correct, and may generate less optimal remapping decisions. Additionally, the remap chains that were present at the end of the previous sample may not reflect the accessing behavior of the new sample, and excess penalties may be paid by traversing these stale remap chains. Again, however, the performance results were not so disparate as to warrant taking corrective measures on sample boundaries.

6.3.3 Comparison to upper bound

Figure 6.18(a) showed that the remapping scheme obtains hit ratios of conventional two-way or four-way set-associative caches. However, as the remapping scheme does not attempt to faithfully mimic set-associative cache operation and does not maintain LRU ordering within the remap chains, it would be interesting to compare the hit ratios to that of set-associative caches with optimal replacement. Several algorithms to determine optimal replacement decisions in set-associative caches have been proposed in the past [48], [49], [90], [91]. However, with only simple modifications, the algorithm for determining optimal bypass decisions without a bypass buffer, presented in Figure 4.5, can be transformed into an algorithm for determining optimal replacement decisions. The resulting algorithm is shown in Figure 6.22.

Now all references must be placed in the temporary cache set. With optimal replacement, the algorithm can preferentially replace an $N\!-\!ref$, which by definition does not have any later reuse. Therefore, if the reuse window maintained in the *cache_queue* does not already contain an address, an $N-ref$ is removed from the set before adding the new reference. Therefore, when the set size equals the associativity, the set must either contain all $R\!-\!ref$s except at most one $N-ref$, which must have been the last reference added to the set. If a reference matches an address already in the set, the matching set element's flag must be updated to that of the reference. The reason is that, as described in Section 4.1.2, the identified reuse chains are of the form $(R\!-\!ref \rightarrow)^{*} N\!-\!ref$. Therefore, when the last element in a reuse chain is being examined in the inner loop of algorithm `upper_bound_replace`, it will be an $N\!-\!ref$, and the reuse chain can be safely purged from the cache set. Marking the corresponding set element as an $N-ref$ will result in its displacement. The proof of the optimality of this algorithm is similar to that

```
/* define */
CA_ASSOC = cache associativity;
CA_ENTRIES = number of sets in cache;
CACHE_INDEX(address) = set index corresponding to address;

algorithm upper_bound_replace
{
    /* Init */
    cache_queues[0..CA_ENTRIES] = NULL;

    for each address (reverse order) do:
    {
        new_entry->flag = N-ref;
        set = NULL;
        ca_entry = CACHE_INDEX(address);
        done = FALSE;

        for each entry in cache_queues[ca_entry] do:
        {
            if (!done)
            {
                if (new_entry->flag == N-ref && address == entry->address)
                    new_entry->flag = R-ref;

                if (entry->addr is in set)
                    replace matching set element's flag with entry->flag;
                else
                {
                    if (set contains any N-ref elements) remove one N-ref element;

                    add entry to set;

                    if (|set| == CA_ASSOC) done = 1;
                }
            }
            else dequeue entry;
        }
        enqueue new_entry in cache_queues[ca_index];
    }
    dequeue all remaining entries in cache_queues[0..CA_ENTRIES];
}
```

Figure 6.22 Algorithm to determine optimal replacement decisions.

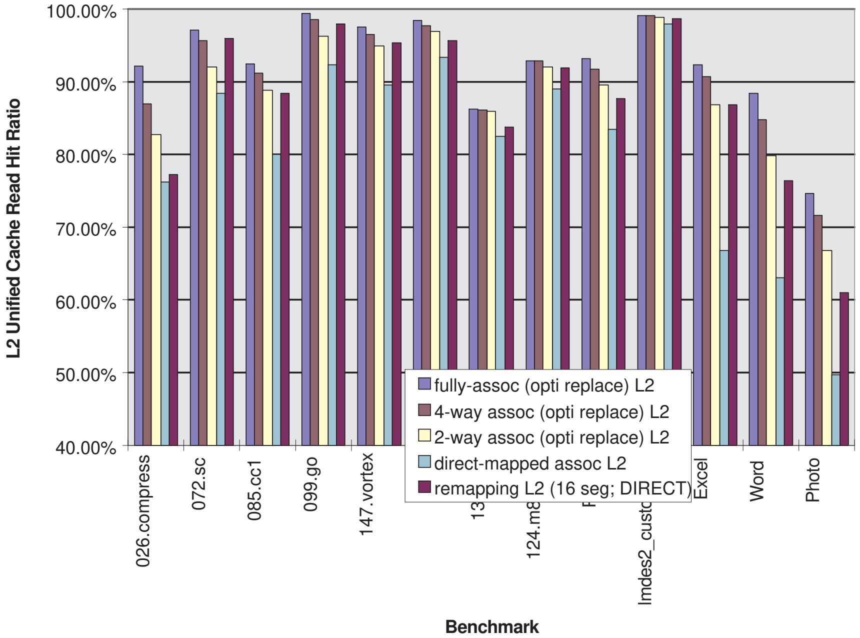


Figure 6.23 Comparison of remapping hit ratios to those of L2 caches with varying set-associativities and optimal replacement decisions. The remapping configuration uses 16 cache segments and the Direct method. Also, 5-bit *conf_ctrs* and $t_1 = 1$ are utilized.

given in Section 4.1.2 for the algorithm to determine optimal bypassing decisions when no bypass buffer exists.

Figure 6.23 again shows the hit ratio obtained by the remapping scheme with 16 cache segments, the Direct selection method, five-bit *conf_ctrs* and $t_1 = 1$. Also shown are the hit ratios obtained using the optimal replacement decisions from the above algorithm. In many cases, the remapping algorithm is able to obtain the hit ratio of a two-way set-associative cache using optimal replacement, and in a few cases, that of a four-way set-associative cache with optimal replacement. The results for a fully associative cache with optimal replacement represent the best usage of the cache capacity, given the line size, and therefore the highest hit ratio that the remapping scheme could possibly obtain.

6.4 Hardware Overhead of Remapping Optimization

As discussed in Section 6.3.2, 16 cache segments and a one-bit *conf_ctr* provide good performance for the dynamic remapping mechanism. Therefore, assuming 16 cache segments, a one-bit *conf_ctr*, a one-bit *hit_ctr* and the two-bit *rflags*, the added cost per tag is one byte. For a 256-KB L2 cache with 64-byte lines, the added tag cost will be 4 KB. Remapping will also require several buffers, as described in Section 6.2. Two-way and four-way set-associative caches require 512 bytes and 1 KB, respectively, for the larger tags. They will also require additional tag state to maintain LRU replacements and extra logic to implement the additional tag comparisons. While remapping requires a larger tag array area, the performance results in Figure 6.13 show that the remapping can outperform set-associative caches, even with the same latency.

6.5 Integrated Adaptive Cache Management Optimizations

Just as Chapter 5 examined the performance of the integrated bypassing and spatial locality optimizations, this section examines the performance of integrating all three optimizations presented in this dissertation.

6.5.1 Performance improvements

The same configuration studied in Section 5.4.1 is examined here; however, remapping is performed in the L2 unified cache. The remapping configuration utilizes 16 cache segments, a one-bit *conf_ctr*, a one-bit *hit_ctr*, and the two-bit *rflags*.

However, since bypassing and remapping often target the same load misses, a simple prioritization is implemented. When a cache miss occurs in the L2 cache and the line has a high

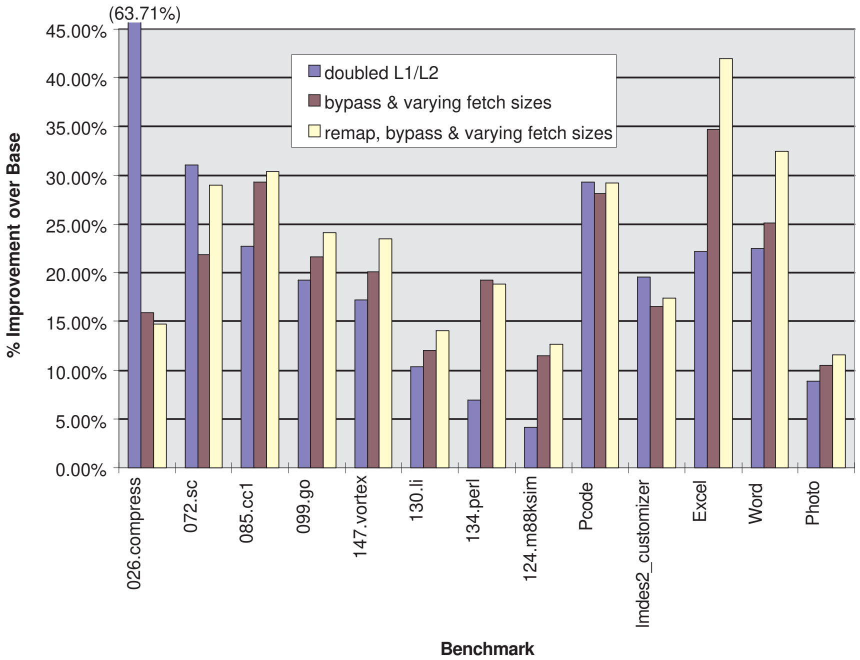


Figure 6.24 Comparison of doubled caches to integrated remapping, spatial locality, and bypassing optimizations.

conf_ctr, remapping is attempted; otherwise, bypassing is attempted. If the remapping attempt fails, due to choosing an unsuitable remapping location, bypassing is not attempted. It is possible that bypassing could be attempted at that point, but the logic and delay may become prohibitive.

As in Chapter 5, a 32-byte L1 instruction cache line size is utilized in this section, due to the fact that the spatial locality optimization employs 32-byte L2 unified cache lines.

Figure 6.24 shows the performance improvements obtained by integrating only the bypassing and spatial locality optimizations, which was also shown in Section 5.4.1, as well as the performance improvement obtained by integrating all three optimizations. Again, the performance improvement obtained by simply doubling the caches at both levels is also shown. In all but two cases, the performance improvement increases when remapping is also employed, sometimes significantly. For example, for *072.sc* the performance of the integrated optimizations with remapping is much closer to that of doubling the caches. In two cases, *026.compress* and *134.perl*, the performance improvement decreases somewhat when utilizing remapping. However, for *134.perl* the performance is still much better than that of doubling the caches.

Figure 6.25 shows the breakdown of processor execution cycles into the portions spent stalled waiting for cache misses to be serviced. The L1 portions include stall cycles from both the L1 instruction and data caches, and the L2 portions include stall cycles from both the instructions and data in the L2 unified cache. At the L1 cache level, the performance improvements tend to come from reductions in the conflict miss stall cycles. In the case of both *Pcode* and *lmdes2_customizer*, there is also a significant reduction in L1 capacity miss stall cycles. At the L2 cache level, the performance improvements tend to come from large reductions in the L2 conflict miss stall cycles, as well as from reductions in L2 cold start miss stall cycles.

6.5.2 Hardware overhead

The hardware overhead estimated in Section 6.4 will increase when combined with the spatial locality optimization, because of the increase in tag array entries from the smaller line sizes. For 256-KB L2 unified cache with 32-byte lines, the added cost will be 8K bytes. Therefore, the savings in hardware overhead at the L2 level, computed in Section 5.4.2 for the integrated bypassing and spatial locality optimizations, will decrease to 43% of the doubled L2 cache cost when remapping is also performed. This is still a large savings, and with the performance for the integrated optimizations shown in Figure 6.24, provides an even clearer advantage over simply doubling the caches.

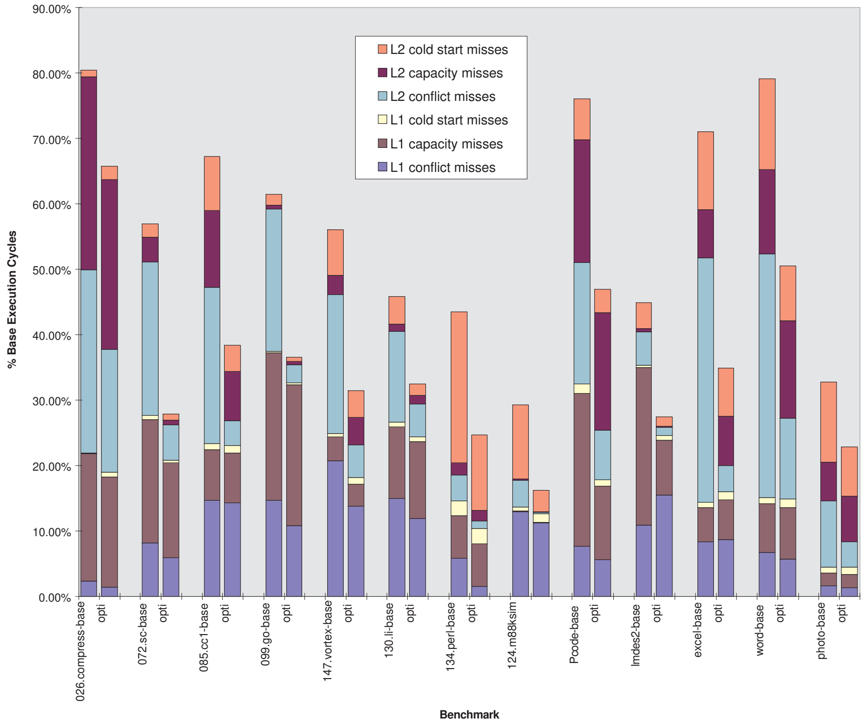


Figure 6.25 Breakdown of stall cycles incurred by each level of the memory hierarchy for both the base configuration and the integrated optimizations.

CHAPTER 7

CONCLUSIONS

7.1 Summary

As memory latencies increase, the importance of cache performance improvements at each level of the memory hierarchy will continue to grow. Also, as the available chip area grows, it makes sense to spend more resources to allow intelligent control over the cache management in order to adapt the caching decisions to the dynamic accessing behavior. This dissertation examined run-time adaptive cache optimizations, which perform hardware monitoring of dynamic accessing behavior and cache utilization, and then use the gathered information to make adaptive caching decisions.

The concept of a macroblock was introduced, which allows larger portions of the accessed memory locations to be feasibly and effectively monitored. Also introduced was the memory address table (MAT), which is a table in hardware that contains the monitors corresponding to each accessed memory macroblock. Three run-time adaptive caching optimizations were then introduced, two of which utilize the MAT technology.

The first optimization is cache bypassing, which can improve the efficiency of the caches by not caching data that is expected to have little reuse in cache. This allows more frequently accessed data to remain cached longer, and therefore have a larger chance of reuse. An upper-bound analysis was first performed, which suggested that cache bypassing could indeed provide significant improvements in hit ratio; however, in order for uniformly significant gains across all benchmarks examined in this dissertation, bypassed data should be cached in a small buffer. These small bypass buffers hold bypassed data near the main cache for a short time, allowing exploitation of small amounts of temporal locality which may exist in the bypassed data. Detailed trace-driven simulations were performed for a number of integer programs, including benchmarks from SPEC92 and SPEC95, modules taken from a research compiler, and several Windows NT applications. The results show that significant speedups over conventional caches can be achieved by this technique. The speedups are due to the improved miss ratios and reduced bus traffic, which also result in a reduction in the average miss penalty per load. The techniques were also shown to achieve roughly half of the upper-bound hit ratio improvements, versus conventional caches, for most benchmarks.

Spatial locality optimizations must be able to detect and adapt to the varying amount of spatial locality both within and across applications in order to be effective. The second optimization presented in this dissertation meets these objectives by detecting the amount of spatial locality in different portions of memory, and making dynamic decisions on the appropriate number of blocks to fetch on a memory access. A Spatial Locality Detection Table (SLDT) was introduced to facilitate spatial locality detection for data while it is cached. This information is later recorded in the MAT for long-term tracking at the macroblock granularity, and is then used to tune the fetch sizes for each missing access. Detailed simulations showed that significant speedups over conventional caches can be achieved by this technique. The improvements are due to the reduction of conflict and capacity misses by utilizing small blocks and small fetch sizes when spatial locality is absent, and utilizing the prefetching effect of large fetch sizes when spatial locality exists. An upper-bound analysis was used to detect those cases in which the optimal fetch size could be clearly determined, and it was shown that the implementation described in this dissertation chose the optimal fetch size in those cases roughly 75% of the time.

The third optimization presented in this dissertation dynamically remaps conflicting data to less-utilized portions of the cache. This scheme uses small amounts of statistics gathered in the cache tags to guide the remapping algorithm. The cache is divided into a small number of segments, which provides a cost-effective mechanism for identifying under-utilized cache lines at run time. Remap chains of varying lengths are formed at run time, which are then traversed on accesses to the original direct-mapped location in order to search for the accessed data. Although this traversal can add run-time overhead to the cache accesses, it was shown that the chains can be maintained in such a way that the net overhead is small. Detailed simulations show that the remapping optimization, when applied to a direct-mapped L2 unified cache, can achieve hit ratios and execution time reductions exceeding those achieved by a two-way set-associative cache, and sometimes exceeding a four-way set-associative cache. The benefit of remapping is magnified if the set-associative designs incur longer access times due to their added complexity.

Finally, this dissertation examined integration of the three run-time optimizations. The integrated intelligent cache management techniques were shown to usually outperform simply doubling the sizes of the L1 and L2 data caches, for a much smaller hardware cost.

7.2 Future Work

There are a number of interesting lines of future research stemming from this dissertation. First, the mechanisms examined in this dissertation could be further refined. For example, more sophisticated hardware monitors could be employed. The upper-bound analyses performed in this dissertation provided invaluable insight into the behavior of the proposed optimizations. In Chapter 4, the insight provided by this analysis was utilized to improve the access counter update mechanism. Other improvements may be possible with further study of the insights provided by this analysis.

A second area of future work is to extend the applicability of the adaptive cache management optimizations presented in this dissertation. For example, the techniques were applied to instructions only in the unified L2 cache. It is important to understand how these techniques apply to the L1 instruction cache. Another important extension is to apply the remapping technique to set-associative caches. This will allow two-way set-associative caches to achieve greater degrees of pseudo-associativity. Also, ways to reduce the run-time overhead should be examined, so that the remapping technique is more applicable to L1 caches.

This dissertation performed monitoring of memory addresses at the macroblock granularity. A third area of future research is to examine alternate methods for performing the hardware monitoring. For example, it is possible that some combination of the memory address (or macroblock address) and the instruction program counter could be used as an index into the MAT, which may provide better resolution for the caching decisions. Another approach is to utilize compiler support to mark the accessing behavior of either the memory references or the data structures when it can be determined through static analysis or profiling. Then the MAT capacity can be used to monitor only those addresses or references with truly dynamic accessing patterns. This will enable either reduced MAT sizes, or reduced macroblock sizes.

Finally, this dissertation developed a hardware monitoring framework that enabled significant performance improvements from several adaptive cache management optimizations. This framework could be utilized when exploring other types of run-time adaptive cache management optimizations.

REFERENCES

[1] K. Boland and A. Dollas, "Predicting and precluding problems with memory latency," *IEEE Micro*, vol. 14, pp. 59–66, August 1994.

[2] R. A. Sugumar, "Multi-configuration simulation algorithms for the evaluation of computer architecture designs," Ph.D. dissertation, University of Michigan, Ann Arbor, MI, 1993.

[3] S. G. Abraham and B. R. Rau, "Predicting load latencies using cache profiling," Hewlett-Packard Laboratories, Palo Alto, CA, Tech. Rep. HPL-94-110, November 1994.

[4] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Hewlett-Packard Laboratories, Palo Alto, CA, Tech. Rep. HPL-93-80, February 1994.

[5] K. K. Chan, C. C. Hay, J. R. Keller, G. P. Kurpanek, F. X. Schumacher, and J. Zheng, "Design of the HP PA 7200 CPU," *Hewlett-Packard Journal*, vol. 47, pp. 25–33, Feb. 1996.

[6] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th International Symposium on Computer Architecture*, May 1981, pp. 81–87.

[7] G. S. Sohi and M. Franklin, "High-bandwidth data memory systems for superscalar processors," in *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 53–62.

[8] A. J. Smith, "Cache memories," *Computing Surveys*, vol. 14, pp. 473–530, September 1982.

[9] 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*, June 1990, pp. 364–373.

[10] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceeding of Supercomputing '91*, November 1991, pp. 176–186.

[11] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," Department of Computer Science and Engineering, University of Washington, Seattle, WA, Tech. Rep. 92-06-03, June 1992.

[12] I. Sklenar, "Prefetch unit for vector operations on scalar computers," *Computers Architecture News*, vol. 20, pp. 31–37, September 1992.

[13] S. Mehrotra and L. Harrison, "Quantifying the performance potential of a data prefetch mechanism for pointer-intensive and numeric programs," Center for Supercomputing Research and Development, University of Illinois, Tech. Rep. 1458, November 1995.

[14] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *Proceedings of the 25th Annual Conference on Microprogramming and Microarchitectures*, December 1992, pp. 102–110.

[15] F. Dahlgren, M. Dubois, and P. Strenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *Proceedings of the 1993 International Conference on Parallel Processing*, August 1993, pp. 56–63.

[16] A. K. Porterfield, "Software methods for improvement of cache performance on supercomputer applications," Ph.D. dissertation, Rice University, Houston, TX, 1989.

[17] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991, pp. 43–53.

[18] T. C. 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*, Oct. 1992, pp. 62–73.

[19] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, November 1991, pp. 69–73.

[20] W. Y. Chen, S. A. Mahlke, W. W. Hwu, T. Kiyohara, and P. P. Chang, "Tolerating data access latency with register preloading," in *Proceedings of the 6th International Conference on Supercomputing*, July 1992, pp. 104–113.

[21] L. K. John, V. Reddy, P. T. Hulina, and L. D. Coraor, "A comparative evaluation of software techniques to hide memory latency," in *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, January 1995, pp. 229–238.

[22] C.-H. Chi, C.-S. Ho, and S.-C. Lau, "Reducing memory latency using a small software driven array cache," in *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, January 1995, pp. 202–210.

[23] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, September 1996, pp. 222–233.

[24] M. H. Lipasti, W. J. Schmidth, S. R. Kunkel, and R. R. Roediger, "SPAID: Software prefetching in pointer- and call-intensive environments," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 231–236.

[25] T. Ozawa, Y. Kimura, and S. Nishizaki, "Cache miss heuristics and preloading techniques for general-purpose programs," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 243–248.

[26] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta, "Predictability of load/store instruction latencies," in *Proceedings of the 26th International Symposium on Microarchitecture*, December 1993, pp. 139–152.

[27] J. L. Baer and G. R. Sager, "Dynamic improvements of locality in virtual memory systems," *IEEE Transactions on Software Engineering*, vol. 2, pp. 54–62, March 1976.

[28] J. Pomerane, et al., "Prefetching system for a cache having a second directory for sequentially accessed blocks," U.S. Patent Office, Tech. Rep. 4807110, February 1989.

[29] M. J. Charney and A. P. Reeves, "Generalized correlation based hardware prefetching," Department of Electrical Engineering, Cornell University, Tech. Rep. EE-CEG-95-1, February 1995.

[30] V. Phalke and B. Gopinath, "A miss history-based architecture for cache prefetching," in *International Workshop IWMM 95*, September 1995, pp. 381–398.

[31] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997, pp. 252–263.

[32] 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 Annual International Symposium on Computer Architecture*, May 1993, pp. 179–190.

[33] S. G. Abraham and H. Agusleo, "Reduction of cache interference misses through selective bit-permutation mapping," University of Michigan, Tech. Rep. CSE-TR-205-94, 1994.

[34] C.-H. Chi and H. Dietz, "Unified management of registers and cache using liveness and cache bypass," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, June 1989, pp. 344–355.

[35] S. McFarling, "Cache replacement with dynamic exclusion," in *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 191–200.

[36] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 93–103.

[37] A. González, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *Proceedings of the 1995 International Conference on Supercomputing*, July 1995, pp. 338–347.

[38] J. A. Rivers and E. S. Davidson, "Reducing conflicts in direct-mapped caches with a temporality-based design," in *Proceedings of the 1996 International Conference on Parallel Processing*, August 1996, pp. 151–162.

[39] T. L. Johnson and W. W. Hwu, "Run-time adaptive cache hierarchy management via reference analysis," in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997, pp. 315–326.

[40] D. F. Thiebaut and H. S. Stone, "Footprints in the cache," *ACM Transactions on Computer Systems*, vol. 5, pp. 305–329, November 1987.

[41] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991, pp. 266–275.

[42] AMD Corporation, *AMD-K6 Processor Data Sheet.* AMD Corporation, Sunnyvale, CA, Order Number 20695, 1997.

[43] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[44] S. Laha, J. H. Patel, and R. K. Iyer, "Accurate low-cost methods for performance evaluation of cache memory systems," *IEEE Transactions on Computers*, vol. 37, pp. 1325–1336, November 1988.

[45] J. W. C. Fu and J. H. Patel, "How to simulate 100 billion references cheaply," Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, Tech. Rep. CRHC-91-30, 1991.

[46] Intel, *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Guide.* Intel Corporation, Santa Clara, CA, 1995.

[47] "SPEC newsletter." SPEC, Fremont, CA, 1991.

[48] S. McFarling, "Program analysis and optimization for machines with instruction cache," Ph.D. dissertation, Stanford University, 1988.

[49] R. A. Sugumar and S. G. Abraham, "Efficient simulation of caches under optimal replacement with applications to miss characterization," in *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993, pp. 24–35.

[50] S. Przybylski, "The performance impact of block sizes and fetch strategies," in *Proceedings of the 18th International Symposium on Computer Architecture*, June 1990, pp. 160–169.

[51] A. J. Smith, "Line (block) size choice for cpu cache memories," *IEEE Transaction on Computers*, vol. C-36, pp. 1063–1075, September 1987.

[52] C. Dubnicki and T. J. LeBlanc, "Adjustable block size coherent caches," in *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 170–180.

[53] F. Dahlgren and P. Strenstrom, "On reconfigurable on-chip data caches," in *Proceedings of the 24th International Conference on Microarchitecture*, November 1991, pp. 189–198.

[54] F. C. Chow, K. I. Farkas, and N. P. Jouppi, "How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processor?" in *Proceedings of the 1st High Performance Computer Architecture Symposium*, January 1995, pp. 78–89.

[55] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "Memory-system design considerations for dynamically-scheduled processors," in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997, pp. 133–143.

[56] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994, pp. 24–33.

[57] J. E. Bennett and M. J. Flynn, "Prediction caches for superscalar processors," in *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997, pp. 81–90.

[58] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay, "The split temporal/spatial cache: Initial performance analysis," in *Proceedings of the SCIzzL-5*, March 1996, pp. 63–70.

[59] K. S. McKinley and O. Temam, "A quantitative analysis of loop nest locality," in *Proceedings of the 7th International Conference on Architecture Support for Programming Languages and Operating Systems*, October 1996, pp. 94–104.

[60] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, June 1991, pp. 54–63.

[61] S. J. E. Wilton and N. P. Jouppi, "CACTI: An enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 677–688, May 1996.

[62] M. D. Hill, "A case for direct-mapped caches," *IEEE Computer*, vol. 21, pp. 25–40, December 1988.

[63] S. Przybylski, M. Horowitz, and J. Hennessy, "Performance tradeoffs in cache design," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, May 1988, pp. 290–298.

[64] M. Wolf and M. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 30–44.

[65] J. Sánchez, A. González, and M. Valero, "Static locality analysis for cache management," in *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, November 1997, pp. 261–271.

[66] J. E. Smith and J. R. Goodman, "Instruction cache replacement policies and organizations," *IEEE Transaction on Computers*, vol. C-34, pp. 234–241, March 1985.

[67] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th International Symposium on Computer Architecture*, May 1989, pp. 242–251.

[68] K. Gosmann, C. Hafer, H. Lindmeier, J. Plankl, and K. Westerholz, "Code reorganization for instruction caches," in *Proceedings of the 26rd Hawaii International Conference on System Sciences*, January 1993, pp. 214–223.

[69] O. Temam and N. Drach, "Software assistance for data caches," in *Proceedings of the 1st High Performance Computer Architecture Symposium*, January 1995, pp. 154–163.

[70] A. Seznec, "A case for two-way skewed-associative caches," in *Proceedings of the 20th Annuual International Symposium on Computer Architecture*, May 1993, pp. 169–178.

[71] A. González, M. Valero, N. Topham, and J. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," in *Proceedings of the 1997 International Conference on Supercomputing*, July 1997, pp. 76–83.

[72] N. Topham, A. González, and J. González, "The design and performance of a conflict-avoiding cache," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 71–80.

[73] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proceedings of 18th Internation Symposium on Computer Architecture*, May 1991, pp. 74–83.

[74] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating systems and multiprogramming," *ACM Transactions on Computer Systems*, vol. 6, pp. 393–431, 1988.

[75] N. Drach and A. Seznec, "Semi-unified caches," in *Proceedings of the 1993 International Conference on Parallel Processing*, August 1993, pp. 25–28.

[76] R. R. Kessler, R. Jooss, A. Lebeck, and M. D. Hill, "Inexpensive implementations of set-associativity," in *Proceedings of the 16th International Symposium on Computer Architecture*, May 1989, pp. 131–139.

[77] B. Calder, D. Grunwald, and J. Emer, "Predictive sequential associative cache," in *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, February 1996, pp. 244–253.

[78] J. H. Chang, H. Chao, and K. So, "Cache design of a sub-micron CMOS System/370," in *Proceedings of the 14th Annual Symposium on Computer Architecture*, June 1987, pp. 208–212.

[79] K. So and R. Rechtshaffen, "Cache operation by MRU change," *IEEE Transactions on Computers*, vol. 37, pp. 700–709, June 1988.

[80] L. Liu, "Partial address directory for cache access," IBM Research Report, March 1993.

[81] A. Seznec, "DASC cache," in *Proceedings of the 1st High Performance Computer Architecture Symposium*, January 1995, pp. 134–141.

[82] K. B. Theobald, H. H. J. Hum, and G. R. Gao, "A design framework for hybrid-access caches," in *Proceedings of the 1st High Performance Computer Architecture Symposium*, January 1995, pp. 360–369.

[83] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen, "Avoiding conflict misses dynamically in large direct-mapped caches," in *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1994, pp. 158–170.

[84] B. K. Bray, W. L. Lynch, and M. J. Flynn, "Page allocation to reduce access time of physical caches," Stanford University, CA, Tech. Rep. CSL-TR-90-454, November 1990.

[85] W. L. Lynch and M. J. Flynn, "Paging performance with page coloring," Stanford University, CA, Tech. Rep. CSL-TR-91-493, September 1991.

[86] R. E. Kessler and M. D. Hill, "Miss reduction in large, real-indexed caches," Department of Computer Science, University of Wisconsin-Madison, Tech. Rep. 940, June 1990.

[87] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Transactions on Computer Systems*, vol. 10, pp. 338–359, November 1992.

[88] K. Westerholz, S. Honal, J. Plankl, and C. Hafer, "Improving performance by cache driven memory management," in *Proceedings of the 1st High Performance Computer Architecture Symposium*, January 1995, pp. 234–242.

[89] Integrated Device Technology, Inc., "IDT Announces New Category of SRAM." Integrated Device Technology, Inc., Santa Clara, CA, http://www.idt.com/corpinfo/PR/1997/Apr97/syncdpt.html, April 1997.

[90] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[91] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

VITA

Teresa Louise Johnson was born in Rantoul, Illinois, in 1970. She grew up in St. Charles, Illinois, and attended St. Charles High School. She pursued her undergraduate studies at the University of Illinois at Urbana-Champaign, where she received the bachelor of science degree in electrical engineering in 1993. In 1993, she began her graduate studies at the University of Illinois at Urbana-Champaign and received the master of science degree in electrical engineering in 1995. During her graduate tenure, she was a member of the Center for Reliable and High-Performance Computing and the IMPACT project directed by Professor Wen-mei Hwu. She spent the summer of 1995 at Silicon Graphics, Inc., in Mountain View, California, and the summer of 1996 at Hewlett-Packard Laboratories in Palo Alto, California.