Using Virtual Memory to Improve Cache and TLB Performance

by

Theodore Haynes Romer

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1998

Approved by____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree____

Date____

University of Washington

Abstract

Using Virtual Memory to Improve Cache and TLB Performance

by Theodore Haynes Romer

Chairperson of Supervisory Committee
Professor Brian N. Bershad
Computer Science and Engineering

This thesis introduces new operating system policies that use virtual memory to dynamically improve memory system performance. Overall application execution time is increasingly dependent on memory system performance, motivating the development of new techniques for reducing cache and translation lookaside buffer (TLB) miss rates.

My thesis is that the operating system can effectively manage cache and TLB resources at runtime on behalf of applications. I develop and evaluate two examples of operating system memory system optimizations. First, I show how the operating system can optimize TLB performance by dynamically constructing superpages. I introduce a policy that analyzes the cause of each TLB miss, and uses this information to selectively create large pages, reducing the TLB miss rate without the increase in memory consumption that would accompany a global increase in the page size. Second, I show how the operating system can optimize cache performance by dynamically remapping pages. The operating system controls which virtual pages potentially conflict in a physically indexed cache, because it controls the mapping from virtual to physical pages. I propose two techniques that inform the operating system when virtual pages are in fact resulting in conflict misses, allowing the operating system to remap conflicting pages. By exploiting the associativity inherent in the mapping from virtual pages to cache pages the operating system can eliminate cache conflict misses without the hardware cost of an associative cache.

I use both simulation and implementation to show that these techniques can improve application performance. I also show that the two techniques can be combined, resulting in a new policy that outperforms either of the separate techniques. Since these policies are

implemented in the operating system, they benefit arbitrary applications, without modifying existing processors and without rewriting or recompiling existing software. As long as hardware memory system implementations provide adequate feedback and control mechanisms, the operating system can automatically improve performance by better managing available hardware resources to meet application resource requirements.

Table of Contents

List of Figures

List of Tables

Acknowledgments

In writing this thesis I enjoyed tremendous support from a wonderful group of friends and colleagues in Seattle and in the Computer Science and Engineering Department at the University of Washington. It is impossible to name them all here, or to even properly express my gratitude to those I do mention here. My advisor Brian Bershad helped me see the research I did in graduate school as a single body of work. Melody Kadenko-Ludwa helped me manage Brian. My other reading committee members—Anna Karlin and Hank Levy—helped improve the thesis significantly. My remaining thesis committee members—Craig Chambers, Susan Eggers, and Eve Riskin—also offered valuable advice. Several friends read and commented on drafts of the thesis, including Jonathan Shade, Sean Sandys, Dennis Lee, and Geoff Voelker. My research is at its best when it was collaborative: Brad Chen, Wayne Wong, Jean-Loup Baer, Alec Wolman, and Wayne Ohlrich, as well as Dennis, Geoff, Brian, Hank, and Anna were all co-authors on one or more papers. My late undergraduate advisor Louis E. Rosier was my first co-author and shared the joy of my first result. David Becker and Warren Jessop were always able to find me more disk space. No graduate student in the department prospers without the help of Frankye Jones, our academic advisor. John Wilkes at HP Labs waited, good-naturedly, while I finished my thesis.

My officemates over the past seven years offered many kinds of assistance, most of all in tolerating and even enjoying my "tedness" and in letting me "romer" their food: Neal Lesh is the best coffee drinker I've ever known; Jonathan let push him around; Sean changed the way I play Ultimate; Tashana Landray and Jake Cockrell provided the office Christmas tree; Wayne O. met me for breakfast every other Thursday; Dave Grove taught me how to deal with kegs of beer; Anthony LaMarca decorated the office with posters and plants and a dartboard and a velvet John Wayne; Mike Salisbury made the montage of childhood photos of the students in the office; and Jeff Dean shared his chips and salsa and Coke and technical insights.

Sung-eun Choi and E Chris Lewis and Ben Dugan and Jim Fix always tolerated my visits to their office. Sung surprised me with birthday and graduation gifts. Ruth Anderson helped me explore the parameters of the ski-rental problem in the real world. Nick Kushmerick and Marilyn McCune let me stay with them in Prague. Eric Anderson was always available to discuss matters technical and otherwise over beer. Brad Chamberlain was forgiving when I left school with his housekeys.

Finally, Shannon Creger provided support beyond anything I could have imagined as I finished writing, revising, and defending my thesis. My parents, Bob and Betty Romer, have always encouraged me throughout my diverse professional and personal pursuits. My late mother, Diana Romer, reminded me to trust my instincts.

Parts of this thesis have previously been published in somewhat different form. Chapter 3 is based on the paper "Reducing TLB and Memory Overhead Using Online Superpage Promotion" published in the *Proceedings of the 22nd Annual Symposium on Computer Architecture* [Romer et al. 95]. Chapter 4 is based on the paper "Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware" published in the *Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation* [Bershad et al. 94]. Chapter 5 is based on the paper "Avoiding Cache Misses Dynamically in Large Direct-Mapped Caches" published in the *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* [Romer et al. 94].

In memory of my mother, Diana Haynes Romer.

Chapter 1

Introduction

In this thesis I present operating system virtual memory policies that automatically, transparently, and dynamically improve the memory system performance of arbitrary user-level programs. The policies are automatic in that they do not require any hints from the program about its memory reference patterns. The policies are transparent in that they do not require any change to the program's abstract model of storage, virtual memory. The policies are dynamic in that they observe and respond to changing application behavior as the program runs. The effect of these policies is to improve performance by matching application resource requirements and the available hardware resources.

The effectiveness of the memory hierarchy is a key element in the performance of computer programs. It has almost become an axiom of computer science that the time to access memory is increasing relative to the time to process instructions. For example, Hennessy and Patterson report that the gap between processor and memory performance is growing at 50% a year [Hennessy & Patterson 95]. To address the high cost of accessing memory, modern memory hierarchy implementations are becoming increasingly complex.

The focus this thesis is on the effect of caches and translation lookaside buffers (TLBs) on performance. Caches hold a subset of the contents of main memory in high speed storage, and TLBs hold a subset of operating system virtual to physical translations (page tables) in high speed storage. Application memory references that are satisfied by the cache avoid the penalty of accessing relatively slow main memory, and virtual addresses that can be translated by the TLB avoid a relatively slow page table lookup. Memory references that do result in cache or TLB misses, however, cost tens of processor cycles. Thus two key predictors of overall application execution time are the cache and TLB miss rates.

One indication of the importance of memory system performance is the increasing investment of both money and engineering effort to reduce the impact of high memory latency. An example of monetary investment in memory system technology is provided by Rambus, a corporation that produces and licenses technology for high bandwidth memory systems. The market capitalization of Rambus was $1.8 billion as of August 22, 1997, reflecting a 580% increase since its initial public offering in May, 1997. An example of engineering investment in memory systems is provided by the Digital Alpha 21264 microprocessor [Keller 96]. This state-of-the art processor includes not just on-chip caches, but next line and set predictors (to reduce the average time to access first-level instruction caches), miss status holding registers (to allow the processor to continue to execute while satisifying cache misses), and prefetch instructions (to allow the compiler to hide the latency of memory references). Thus the stock market is willing to invest capital, and Digital (among other microprocessor manufacturers) is willing to invest engineering resources, in order to address the processor-memory performance gap. Like the solutions offered by Digital and Rambus, the policies introduced in this thesis can transparently improve the performance of arbitrary applications. Unlike those solutions, however, the policies in this thesis require only minimal changes to hardware processor and memory system designs.

Despite the importance of the memory system implementation to program performance, the details of cache and TLB configurations are hidden from user-level applications. While this aids portability, it can also hinder performance. Applications can make memory references that result in unnecessary cache or TLB misses. As a trivial example, a program might repeatedly generate references to two memory addresses that map to the same cache line. In a direct-mapped cache, only one of these two addresses can be in the cache, so the alternating references will result in cache misses. Even though the cache is large enough to hold both data items, the reference stream results in a large number of cache misses. This state of affairs is only revealed indirectly to the program or to the user in the form of long execution time. Only if the program is aware of the cache configuration will it be able to attribute the poor performance to the conflicting memory references.

This pathological example reveals both a problem and an opportunity. Because the user-level program is unaware of the details of the hardware, small problems such as this conflict may persist and result in significant performance bottlenecks. At the same time, the persistence of poor behavior means (1) that it is possible to detect problems and improve future performance by correcting the causes of the problems and (2) that it is acceptable for the detection and correction mechanisms themselves to have a high cost, as long as they have an even larger benefit.

The policies described in this thesis are targeted at memory system performance problems that result from the interaction of the software memory reference stream and the hardware implementation of the memory hierarchy, and that:

- have locality – that is, they persist as the application runs;
- can be detected using feedback mechanisms; and
- can be corrected in the operating system by making more effective use of the cache and TLB.

The policies I present to solve these problems:

- rely on a combination of simple hardware support and operating system modifications to monitor the dynamic behavior of applications;
- use the information collected to identify sources of memory system delays such as cache misses and TLB misses;
- identify and resolve these bottlenecks, and as a result not only pay for the overhead of the monitoring mechanisms, but also significantly improve overall system performance; and
- require no changes to existing software or to existing CPU implementations.

The importance of such techniques will grow as the cost of accessing memory continues to increase relative to processor cycle time.

1.1 Thesis statement and contributions

My thesis is that (1) effective management of hardware memory system resources requires dynamic detection, analysis, and correction of memory system bottlenecks, and that (2) the operating system is ideally situated to carry out these tasks. Static compiler-based techniques for improving memory system performance rely on programs with predictable reference patterns, cannot address resource conflicts that arise among multiple programs, require recompilation, and may be tailored to a specific memory system configuration. Purely hardware-based schemes are inflexible, in that the policies for analyzing and eliminating bottlenecks cannot be modified once the hardware is designed and built. In contrast, given appropriate feedback and control mechanisms, the operating system can implement software policies that are tailored to the characteristics of both the workloads and the hardware, and that can adapt dynamically as program behavior changes.

The contributions of this thesis are to concretely demonstrate the opportunity to use the operating system to improve memory system performance.

- First, I introduce a policy that improves TLB performance by monitoring the TLB miss stream, analyzing the implied memory reference pattern of the workload, and then selectively constructing superpages. A superpage increases the granularity of virtual-to-physical translation, allowing a fixed-size TLB to map a larger portion of the virtual address space without a corresponding increase in physical memory requirements. This solution has three components: a feedback mechanism, namely notification of the location of TLB misses; a control mechanism, namely the ability to specify and construct superpages; and a superpage construction policy implemented in software.

- Second, I introduce operating system policies that eliminate cache conflicts in direct-mapped physically indexed caches by detecting pages that are suffering from cache conflicts and changing virtual to physical mappings to resolve the conflict. The goal of these policies is to arrange that active virtual pages are mapped to physical pages that do not conflict in the cache. The primary difficulty in implementing these policies is in designing appropriate feedback mechanisms. I introduce and evaluate two approaches to cache conflict resolution. One uses detection mechanisms based on existing hardware (TLB misses and cache miss performance counters), while the other uses proposed new hardware that precisely identifies the pages suffering from conflict misses. The details of the policies that interpret this feedback depend on the accuracy of the feedback mechanism. The control mechanism, in this instance, is the ability to transparently change virtual to physical mappings.

- Third, I analyze the interaction of these two techniques. The first technique relies on selectively increasing the page size in order to increase the coverage of the TLB. The second technique relies on a high degree of associativity in choosing the mapping from virtual to physical pages, but as the page size increases, this flexibility decreases. I show that despite this apparent conflict, the two techniques in fact complement one another. Using the two techniques together results in better performance than when using either technique alone.

Together, the contributions of this thesis illustrate the importance of using hardware feedback and control mechanisms and software operating system policies to analyze and eliminate memory system bottlenecks.

1.2 Thesis overview

The rest of this thesis is organized as follows:

- Chapter 2 provides background information on the interaction of the virtual memory system, caches, and TLBs, and describes related work in this area.

- Chapter 3 describes a policy for dynamically constructing superpages in order to eliminate TLB misses without increasing the hardware TLB size or significantly increasing physical memory requirements.

- Chapter 4 describes how to detect and eliminate cache conflicts using standard hardware. Once conflicting pages are detected, the operating system can be remap them to physical pages that do not contend for the same location in a physically indexed cache.

- Chapter 5 describes how the approach described in Chapter 4 could be made more effective with the addition of a proposed simple hardware device, the Cache Miss Lookaside (CML) Buffer, that summarizes per-page miss counts.

- Chapter 6 considers the performance impact of a combined policy for superpage construction and cache conflict elimination, and also considers how the policies proposed in this thesis would affect the performance of Windows applications compiled for the x86 architecture.

- Finally, Chapter 7 summarizes and concludes.

THE PRINCIPLE OF LOCALITY

"She's made mistakes," he said. "So have I and so have you."

"At least I ain't made the same ones over and over agin," I said.

"Why not? You might as well make them you're used to as to make new ones all the time. It don't do no more damage."

Larry McMurtry, *Leaving Cheyenne*

Chapter 2

Background and Related Work

The memory system optimizations introduced in this thesis draw upon and extend the efforts of many researchers to improve the design, implementation, and performance of the memory hierarchy. In this chapter I describe the basic operation of virtual memory, caches, and TLBs. I then discuss how prior research has sought to improve performance by exploiting the interactions between virtual memory and caches, and between virtual memory and TLBs. The work described in this thesis moves beyond the earlier research by showing how to manage cache and TLB resources dynamically rather than statically, thus allowing the system to adapt to changing application resource requirements.

As discussed in the introduction, the focus of this thesis is on policies that can be implemented in the operating system, without requiring changes either to on-chip hardware or to applications. Memory system optimizations that require specialized on-chip functionality are expensive and time consuming to implement. Optimizations that rely on algorithmic changes, program hints, or compiler support require varying degrees of programmer intervention, and may be limited to improving performance for specific hardware configurations. In this chapter, therefore, I describe background and prior work for cache and TLB performance optimizations that require little or no hardware support, that are program-independent, and that are automatic.

2.1 Background

Before presenting more detailed background, I first give some intuition about how virtual-to-physical mappings can affect cache and TLB performance.

Figure 2-1 shows a simplified representation of the functionality of the TLB: it holds a subset of the mapping from virtual pages to physical pages, where a page is just a contiguous, aligned region of the address space. In the left half of the figure, five TLB entries are needed to map the application's working set. Four of the pages are contiguous and aligned in the virtual address space, but not in the physical address space. In the right half of the figure, the same virtual pages are in use, but now four of the pages are also aligned and con-

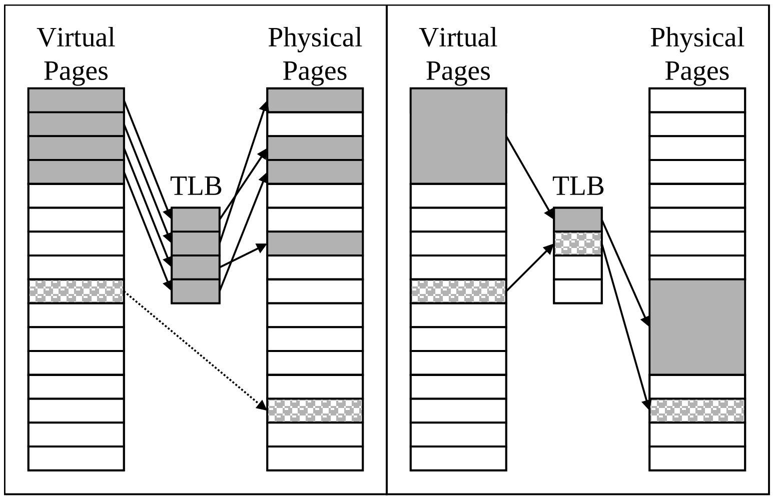


Figure 2-1 Effect of page mappings on TLB performance. With appropriate virtual-to-physical mappings, the page size can increase, so that fewer TLB entries are needed to cover the same part of the virtual address space. In the mapping on the left, the four-entry TLB cannot cover the shaded virtual addresses. In the mapping on the right, the same virtual addresses are covered by just two TLB entries.

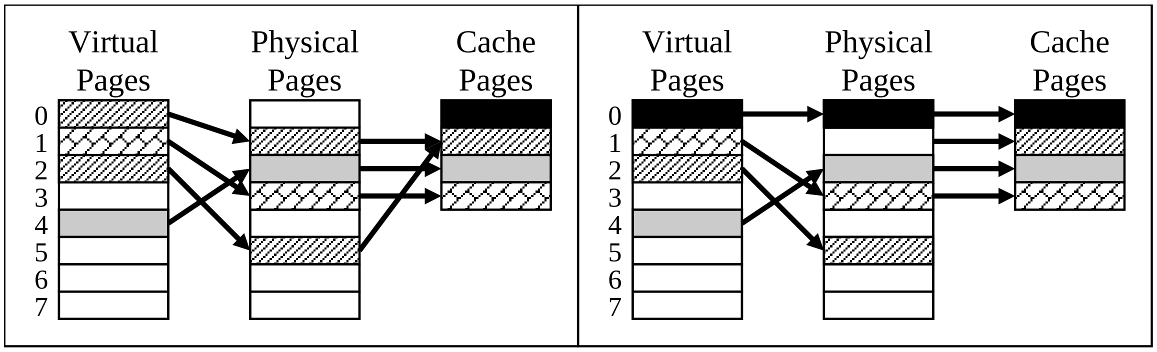


Figure 2-2 Effect of virtual-to-physical mappings on cache performance. In the mapping on the left, two active virtual pages are mapped to the same cache page, so cache conflict misses can occur. In the mapping on the right, the set of active virtual pages is mapped to disjoint cache pages, so cache conflict misses cannot occur.

tiguous in the physical address space as well. If the TLB supports multiple page sizes, then a larger page can be used to map the group of four pages. The working set of the application can then be mapped with just two TLB entries, resulting in fewer TLB misses. In Chapter 3 I introduce dynamic virtual-physical mapping policies that transparently create large pages as needed. In Section 2.1.1 I provide more detailed background on the role of the virtual memory system in TLB performance and the opportunity to use superpages.

Figure 2-2 illustrates how virtual-to-physical mappings can affect cache performance. Again, the figure shows mappings from virtual pages to physical pages. In addition, the figure shows the mapping from physical pages to *cache* pages, page-sized regions of a physically indexed cache. Each cache page is assigned a different color, and each virtual page is given the color of the cache page to which it is ultimately mapped. Pages of different colors will never conflict in the cache. In the left half of the figure, pages 0 and 2 may conflict; but in the right half of the figure, with exactly the same set of virtual pages, no cache conflicts are possible. In Chapters 4 and 5 I will describe techniques for dynamically creating virtual-to-physical mappings like that in the right half of Figure 2-2 in order to reduce the number of cache conflicts. In Section 2.1.2 I provide more detail on the operation of caches and their interaction with the virtual memory system.

2.1.1 Virtual memory, translation lookaside buffers (TLBs), and superpages

In order to understand the virtual memory management policies that I present later in this thesis, it is first necessary to understand the basic operation of virtual memory, and in particular to understand the role of page size in performance. For a more complete treatment of virtual memory the reader can consult standard references such as [Silberschatz & Galvin 97]. The material that I present here is intended only to provide enough background to motivate the use of multiple page sizes, page remapping, and dynamic superpage construction.

Modern computer systems provide user-level programs with an abstraction of storage called *virtual memory*. The program has uniform and exclusive access to a large *virtual address space*, typically 2^{32} bytes (4 gigabytes) or more. In this thesis I am concerned with systems that implement *paged* virtual memory [Denning 70]. In a paged system, physical memory is used as a cache of virtual memory, and the size of the cache element is the virtual memory *page size*. Typically the page size is between 4 and 64 kilobytes (KB), and is fixed system wide. The status of each virtual page is tracked by operating system *page tables*. Each virtual page may be cached (*mapped*) by a page in physical memory, stored in secondary storage such as a paging file on disk, or in some special state (*e.g.* invalid). The operating system is responsible for maintaining the page tables and other data structures that track the status of each virtual and physical page in the system. As long as the program refers only to virtual pages that are mapped to physical pages, it will have the illusion of fast exclusive access to large amounts of storage, even on a system with a small physical address space. When the program refers to virtual pages that are *not* cached in physical memory and must be fetched from the paging file, it is *paging*, and its performance will suffer since secondary storage is many orders of magnitude slower than physical memory.

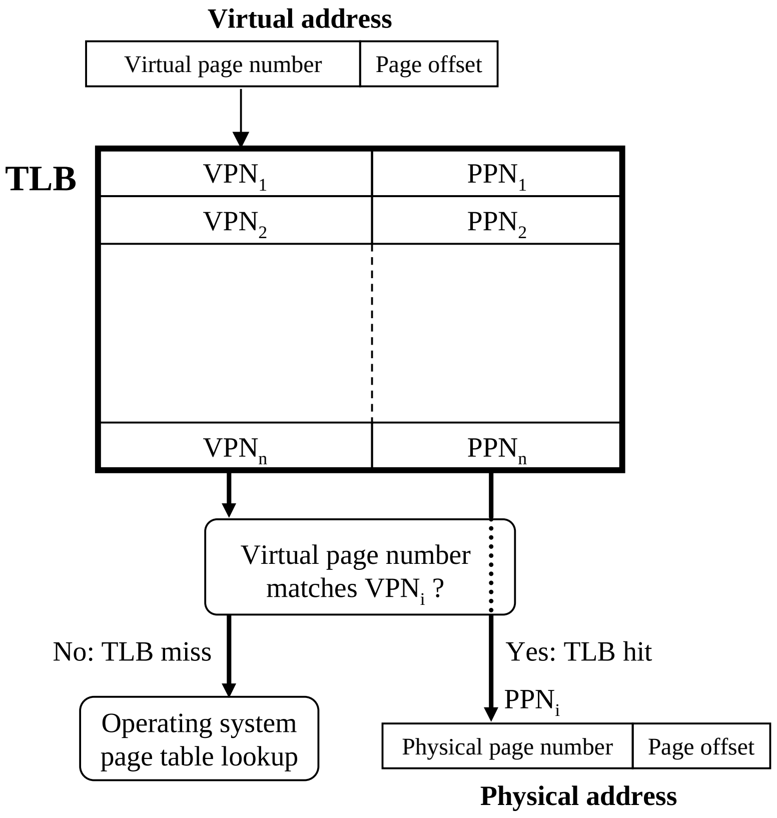


Figure 2-3 A typical fully-associative software-filled TLB, with a single page size. On a reference to virtual address v, the virtual page number is used to access the TLB. If there is a TLB entry with a matching virtual page number, the physical page number is read from the TLB entry and combined with the page offset to form the physical address. The page number is simply the high order $s - \log_2 p$ bits of the virtual address, where s is the number of bits in the address and p is the page size in bytes.

Because the implementation of virtual memory is transparent to applications, user-level programs are unaware of the status of any particular virtual page. In particular, the operating system can choose an arbitrary physical page to back a given virtual page, or even change the physical page backing a virtual page while the application runs. Changing the virtual to physical mapping for a page, or *remapping*, requires three steps: (1) copying the data on the page to a free physical page; (2) updating the page table entry for the affected page to reflect the new virtual-to-physical mapping; and (3) freeing the original physical page. The policies introduced in this thesis rely on the ability to transparently remap pages.

While virtual memory is a great convenience to the programmer, it requires a virtual-to-physical translation for each virtual address generated by the program. Conceptually this means consulting the page tables to find the physical page corresponding to the virtual page. To reduce the overhead of address translation, modern microprocessors include a hardware cache of recently translated pages called a *translation lookaside buffer (TLB)* (Figure 2-3). Addresses that hit in the TLB result in no delay, while addresses that miss in the TLB result in a page table lookup. The cost of a TLB miss is system-dependent, but as an example, on a DEC Alpha 3000/700 a TLB miss costs at least 31 processor cycles. Once the translation is found in the page tables it is inserted into the TLB, evicting some other translation according to the TLB's replacement policy. Some processors perform the page table lookup in hardware (for example, the Intel x86), while others rely on operating system software to perform the lookup (for example, the DEC Alpha, HP PA-RISC, and MIPS R4x00). The hardware approach can reduce the cost of each TLB miss, but constrains the operating system's implementation of page tables. In this thesis I will focus on software-filled TLBs, which notify the operating system on every TLB miss, and hence provide a straightforward feedback mechanism. The techniques described in this thesis could also be applied to systems with hardware-filled TLBs, although this would require a different feedback mechanism.

The *coverage* of the TLB is the product of the number of TLB entries and the operating system page size. When an application's working set is larger than the TLB's coverage, the resulting TLB misses can adversely affect application performance. For example, Figure 2-4 shows the percentage of execution time spent by various applications running on a DEC Alpha 3000/700 workstation. This system was running Digital Unix with fixed size 8 KB pages, and is based on a DEC Alpha 21064 processor, with a 32 entry fully associative TLB for data and an 8 entry fully associative TLB for instructions. The coverage of the data TLB is therefore just 512 KB. Because the data working set of most of these applications is much larger than 512 KB, the applications incur a large number of TLB misses as they run. Coupled with a TLB miss penalty of 31 cycles, the high TLB miss rates result in the

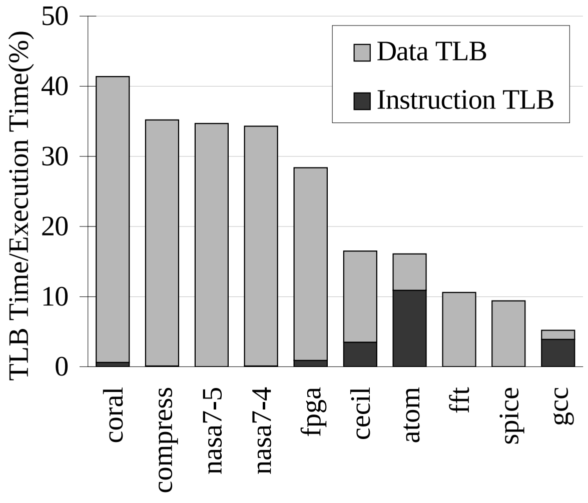


Figure 2-4 Effect of TLB misses on performance on a DEC Alpha 3000/700. This graph shows the percentage of execution time spent handling TLB misses. The data was collected by instrumenting the TLB miss handler with calls to the hardware cycle counter. For additional information on the system under test and the workloads, see Chapter 3.

significant performance degradation illustrated by the figure.

One way to increase the coverage of the TLB and deal with this performance problem is to devote additional hardware resources to the TLB. For example, the follow-on to the 21064, the DEC Alpha 21164, expanded the TLB to 64 entries for data and 48 entries for instructions, doubling the coverage of the data TLB (the primary source of TLB misses for these applications). However, such hardware solutions may increase the time to access the TLB for all references, not just misses; furthermore, merely doubling the TLB size is insufficient for many memory-intensive applications.

Another way to increase the TLB coverage is to increase the operating system page size. However, larger pages can result in inefficient use of physical memory. For example, if a program uses only two words that are widely separated in the virtual address space, two entire physical pages must be allocated to map the two corresponding virtual addresses. The larger the page size, the more memory will be wasted due to internal fragmentation. Inefficient use of memory, in turn, can result in paging – if the working set of the application no longer fits in main memory it must in part be stored on secondary storage.

In contrast, the operating system policies that I describe in this thesis can increase the coverage of a given sized TLB by much more than a factor of two, without requiring changes to existing TLB implementations. In particular, these policies take advantage of TLBs that support variable page sizes, or *superpages*. Variable page sizes offer an opportunity to in-

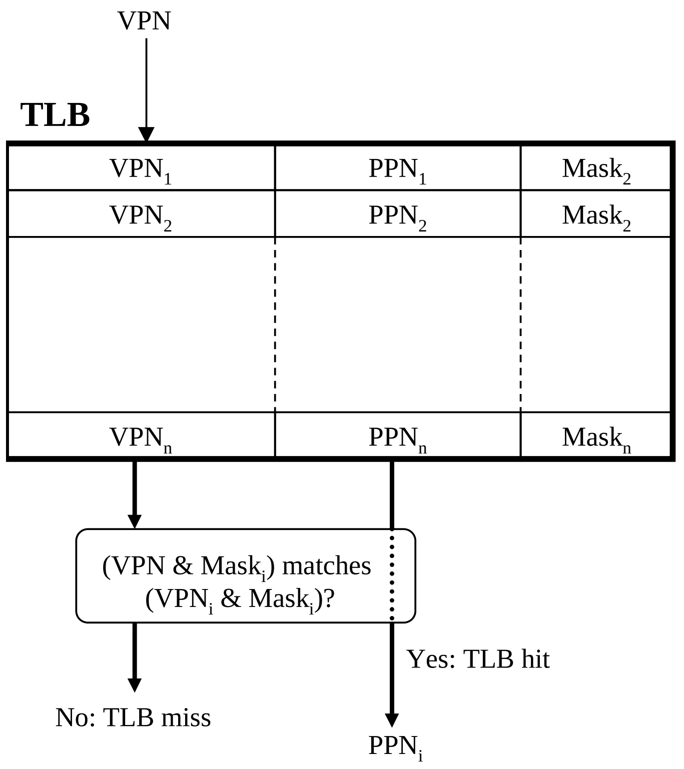


Figure 2-5 A typical TLB, with support for multiple page sizes (superpages). Each TLB entry includes an additional field corresponding to the page size. Conceptually the field is simply a mask m, which indicates which of the bits in the virtual page number are used to detect a match on a reference to the TLB. On a reference to virtual page v, a TLB hit occurs if there is a TLB entry for virtual page v' with mask m' such that $v \& m' = v' \& m'$, where & is the bit-wise AND operation.

crease TLB coverage without incurring the internal fragmentation that would accompany a global increase in the page size. A superpage is a page with a size that is a power of two multiple of the minimum page size, and that is contiguous and aligned in both the virtual and physical address spaces. Each page table entry and TLB entry is augmented with a field that indicates how large the page is or equivalently, how many bits of the virtual address make up the page number (Figure 2-5). One implication of supporting variable page sizes in the TLB is that the TLB must be fully associative [Talluri 95]. Many current processor architectures support multiple page sizes, including the DEC Alpha, the PowerPC, the Intel Pentium, the Sun UltraSparc, and the HP Precision. The range of page sizes varies from system to system; some examples are shown in Table 2-1.

Table 2-1 Example TLB implementations. This table shows the parameters of the TLB implementations on some current microprocessors. Four of the processors shown have support for multiple pages in the primary TLB, while two (the PowerPC and the Pentium Pro) support a single page size in the primary TLB and have auxiliary hardware to support large pages.

Processor	Primary TLB		Auxiliary TLB	
	TLB size	Page size(s)	TLB size	Page size(s)
IBM PowerPC 603e [IBM 95]	64I+64D	4 KB	4I+4D	128 KB, 256 KB, …, 256 MB
Intel Pentium Pro [Int 97]	32I+64D	4 KB	4I+8D	4 MB
Alpha 21164 [Dig 97]	48I+64D	8 KB, 64 KB, 512 KB, 4 MB	not applicable	
HP PA-8000 [Hunt 95]	96	4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, 8 MB	not applicable	
MIPS R10000 [Heinrich 96]	64	4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, 8 MB	not applicable	
UltraSparc II [Sun 97]	64I+64D	8 KB, 64 KB, 512 KB, 4 MB	not applicable	

2.1.2 Caches and virtual memory

The policies that I present in this thesis for improving cache performance exploit the interaction between operating system virtual-to-physical mappings and cache conflict misses. This section provides background on the operation of caches and gives a simple example of the effect of virtual to physical mappings on cache behavior. As with virtual memory systems, a complete treatment of caches is beyond the scope of this thesis, and the reader is referred to the literature for additional background [Smith 82, Hennessy & Patterson 95].

A cache consists of high-speed memory that holds a subset of the data in virtual or physical memory. The cache is divided into cache *lines*, typically between 16 and 64 bytes. On a memory reference, the address is divided into three fields. The low order bits are the offset of the data within the cache line. The middle bits are used to index into the cache. A cache is called *virtually indexed* or *physically indexed* depending on whether the virtual or physical address is used to index into the cache. Virtually indexed caches have the advantage of not requiring an address translation before the cache lookup, but have the disadvantage of introducing aliases (a single virtual address referring to multiple physical addresses) and synonyms (multiple virtual addresses referring to a single physical address).

The high order bits of the address are the *tag*, and are used to distinguish addresses that index to the same cache location. If the address tag and the tag stored in the cache line match, then the reference is a cache hit. Otherwise, the reference is a miss, and the data must be fetched from the next level in the memory hierarchy. In a direct-mapped cache, each memory address maps to exactly one cache line. In an n-way associative cache, each address can map to any one of n lines.

Cache *conflict* misses occur because previously referenced data has been displaced by data that maps to the same cache location. For example, when the cache is large enough to hold the data in use, but the data is unfortunately mapped to lines that contend for the same cache locations, cache misses will occur. Such conflict misses could be prevented with a different mapping of addresses to cache lines.

In a physically indexed cache, the mapping from physical addresses to cache lines is fixed by the hardware cache index function. The mapping from *virtual* addresses to cache lines, though, is controlled by the combination of the operating system's virtual-to-physical mappings and the hardware cache index function. As a result, cache conflicts in a physically indexed cache are in part due to the operating system's assignment of physical pages to virtual pages. The virtual memory page size partitions a direct-mapped physically indexed cache into a set of *cache pages* [Wheeler & Bershad 92]. The set of physical pages, in turn, is partitioned into multiple equivalence classes, or *colors*, where pages of the same color map to the same cache page. Conflicts can only occur between pages of the same color, and no conflicts can occur within a single page as long as the page size is not larger than the cache size.

Any virtual-to-physical mapping that assigns the active virtual pages to physical pages of different colors pages is optimal in that it avoids conflict misses. A mapping that assigns two or more virtual pages to the same color may not be optimal and can induce cache conflict misses. For example, Figure 2-2 showed a "good" and a "bad" mapping for the same set of virtual pages. In the good case, the active virtual pages all map to different cache pages, so no cache conflict misses are possible. In the bad case, two of the virtual pages map to the same cache page, so addresses at the same offset within the two pages will contend for the same cache lines, resulting in conflict misses. In Section 2.2.2, I describe some common static mapping policies that use heuristics to avoid cache conflicts. Since these policies are static, however, they cannot be adjusted if the working set of the application changes or if the heuristic turns out to be inappropriate. As mentioned in the virtual memory section above, it is possible to change virtual to physical mappings transparently to the application. Later in this thesis I will introduce policies that take advantage of this opportunity to eliminate cache conflicts due to poor virtual to physical mappings.

The impact of cache conflict misses on application performance is illustrated by Figure 2-6. The figure shows the simulated improvement in execution times that result from replacing a direct-mapped cache with a two-way associative cache with the same size and access time. Unfortunately, real associative caches have higher cost and higher access time than a direct-mapped cache of the same size [Wood 86], so these results may not be attain-

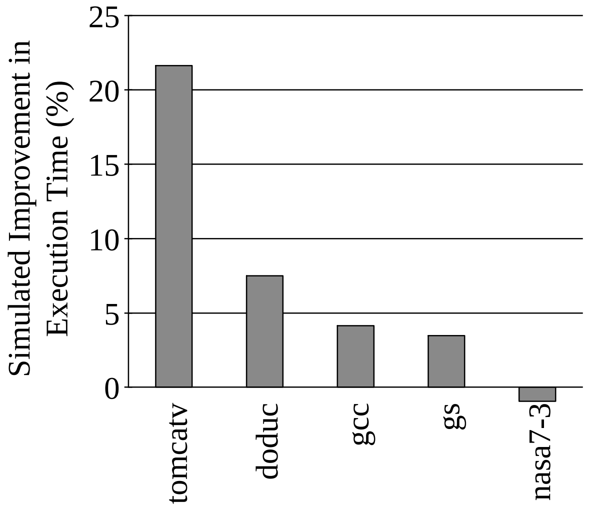


Figure 2-6 Effect of conflict misses (simulated). This graph shows the predicted improvement in execution time by replacing a direct-mapped off-chip cache with a hypothetical two-way associative cache with the same size (512 KB) and access time. The simulated memory system is based on the DEC Alpha 3000/500. The reduction in execution time indicates the opportunity to improve performance by eliminating cache conflicts. For further details on the simulation and the workloads, see Chapter 4.

able in practice. The goal of the policies introduced in this thesis is to use operating system virtual-to-physical mapping policies to eliminate cache conflicts in software without incurring the costs of hardware associativity.

2.2 Related work

In this section I briefly describe prior research based on the observation that the operating system's virtual memory policies affect cache and TLB performance. These techniques generally consist of finding static virtual-to-physical mappings that result in relatively good memory system performance. This thesis shows that by changing page sizes and virtual-to-physical mappings as a program runs, performance can be improved even further.

2.2.1 TLB management

A number of researchers have noted that large pages could improve the TLB performance of applications with large working sets. Most have proposed user-level policies for setting the page size, but have not actually demonstrated the effectiveness of such policies:

- Harty and Cheriton[Harty & Cheriton 92] propose mechanisms that would allow an application to manage its own physical memory, including varying the page size within segments of its address space. They do not discuss policies for selecting page sizes. Implementing the superpage construction policies described in this paper could use their mechanisms for controlling page sizes, but would require extensions to provide feedback on TLB performance to the application.

- In Chen *et al.*'s study of TLB performance [Chen et al. 92], the authors suggest using compiler feedback to identify regions of the program text that could be mapped by larger pages, and tuning memory allocation algorithms to map frequently used data to contiguous regions in order to facilitate the use of large pages. They do not evaluate these proposals, however.

- Mogul [Mogul 93] also suggests using large pages to increase TLB coverage. He describes qualitatively the tradeoffs involved in constructing superpages dynamically, and suggests this as a possible research direction.

Talluri *et al.* [Talluri et al. 92, Khalidi et al. 93, Talluri & Hill 94, Talluri 95, Talluri et al. 95] go further, proposing techniques that use the operating system to improve TLB performance without requiring user-level intervention. They consider superpages, but focus on proposed *partial subblock TLBs*. A subblock TLB, akin to a subblock cache [Hennessy & Patterson 95], has multiple entries associated with each tag. For example, in a TLB with base page size of 4 KB and subblock factor 16, a single TLB entry could map a contiguous aligned region of the virtual address space as large as 64 KB.

As with superpages, for partial subblock TLBs to be effective the operating system must map contiguous aligned regions of virtual memory to contiguous aligned regions of physical memory. Subblock TLBs have less stringent requirements than superpage TLBs, however. If one or more of the component virtual pages cannot be represented by the shared TLB entry (because it differs in protection or validity, or because it is mapped to a different part of the physical address space), it can be marked invalid and mapped with a separate TLB entry. To arrange that pages are aligned and contiguous in the virtual and physical address spaces, Talluri *et al.* propose an operating system policy of *page reservation* to support superpage and subblock TLBs. The virtual and physical address spaces are divided into 64 KB regions; the operating system makes a best effort to allocate virtual pages in one 64 KB region to physical pages in one 64 KB physical region.

This reservation policy can allow the creation of superpages with no runtime cost, and similarly can allow a single subblock TLB entry to map a large number of 4 KB pages. However, it restricts the freedom of the virtual memory system to map virtual pages to specific physical pages (to avoid cache conflicts), and effectively constrains the operating system to allocate pages in 64 KB regions, regardless of the actual access patterns of the application. As a result, this policy may needlessly fragment the physical address space. Specifically, reservation makes it difficult to construct superpages of varying size, since there is no way to decide *a priori* how many base pages to reserve at page allocation time. For small superpages, reservations may often hold until they are needed. As the superpage size grows, however, it is increasingly likely that a reservation will fail: that is, some base page within the reserved superpage is allocated for another purpose before the superpage is actually created.

Talluri *et al.*'s primary interest is in measuring the effectiveness of subblock TLBs. However, they do propose one superpage policy simply to provide a point of comparison. This policy creates a superpage as soon as some fraction of the component pages have been referenced. They do not explore the design space of this policy, nor do they consider policies that copy pages dynamically in order to create superpages at runtime. In effect, superpages are created only if the original reservation was successful. Both their superpage and subblock TLB policies effectively limit the maximum page size to 64 KB, and would be ineffective for applications with working sets larger than would be mapped by a TLB with 64 KB pages. They suggest that larger pages can be created in response to application hints, but do not consider using larger pages for arbitrary applications.

2.2.2 Virtual memory mapping policies

Several other researchers have investigated the impact of the operating system's virtual-to-physical mapping policies on both cache performance and address translation performance. Cache performance is affected by mapping policies as described in Section 2.1.2. As a result, prior research has focused on techniques for mapping simultaneously active virtual pages to physical pages of different colors. Knowing the operating system mapping policy in advance can allow optimizations in the implementation of address translation hardware.

Page coloring

The work of Chiueh and Katz [Chiueh & Katz 92], Taylor [Taylor et al. 90], and Lynch [Lynch 93] is based on a simple static mapping policy called *page coloring*. Page coloring assigns virtual pages to colors in a round robin fashion, assigning virtual page 0 to the first color, virtual page 1 to the second color, and so on. (To avoid inter-address space conflicts, different permutations of the color order can be used for different processes — for example, by hashing the virtual page number with the process identifier to determine the color.) This heuristic has the virtue that pages close together in the virtual address space will not conflict in the cache.

Generally page coloring is not strictly enforced by the operating system: if there is no free page of the desired color, a free page of another color is chosen. However, Lynch, in his thesis on the cache performance of various static page allocation policies, focuses on "perfect coloring," in which the operating system does enforce page coloring strictly. Perfect coloring has the advantage of allowing the placement of virtual pages in the cache to be easily controlled at user-level, and can allow the index into a physically indexed cache to be computed from the virtual address without performing address translation (since the low order bits of the physical page number are determined by the virtual page number.) Unfortunately, by constraining the associativity of the mapping from virtual to physical pages, perfect coloring may result in increased paging rates. Since fetching a page from secondary storage is many orders of magnitude more expensive than a cache miss, real systems do not implement strict page coloring.

Both Chiueh *et al.* and Taylor *et al.* consider how to reduce the hardware needed for address translation when the operating system attempts to map pages according to the page coloring heuristic. While the two papers propose different mechanisms, the basic observation is that in the presence of page coloring, the low order bits of the virtual page number are a good predictor for the low order bits of the physical page number. Taylor *et al.* exploit this observation in their design of the "TLB slice." The TLB slice is a hardware table that provides a translation hint for just the low order bits of the virtual page number in order to generate the index into a physically indexed cache. This allows a virtually-tagged, physically indexed cache to be accessed without performing a full TLB lookup. Page coloring helps the performance of the TLB slice by allowing multiple virtual pages to share a single entry in the table.

Other static mapping policies

Kessler *et al.* [Kessler & Hill 92] considered static page mapping policies that use varying degrees of sophistication to avoid conflicts in a physically indexed cache. They provide an excellent overview of the interaction of page mapping policies and cache performance. They introduce several “careful” mapping policies that use heuristics to avoid cache hot spots. They find that sophisticated policies that try to measure the number of pages allocated to any color provide some minor performance advantages. However, for the workloads under study a simple *bin hopping* heuristic works nearly as well as policies that maintain much more information about past allocation decisions. Bin hopping assigns virtual pages to colors cyclically according to the order in which they are initially mapped. This policy has the virtue that pages initially referenced close together in time will be assigned to different colors and hence will not conflict in the cache. In addition, it is feasible for user-level applications to manipulate the cache placement of virtual pages by controlling the order in which they initially reference pages [Bugnion et al. 96].

Dynamic mapping policies

Finally, Sites [Sites 95] proposes a technique for changing virtual-to-physical mappings in response to application cache miss behavior. This technique is based on hardware that effectively records the address of every cache miss. By periodically sampling the cache miss address, the operating system can identify pages suffering from large numbers of misses. Depending on the cause of the cache misses, such pages can be migrated (to reduce coherence misses on a multiprocessor) or remapped (to eliminate cache conflicts). A potential disadvantage of this policy is that the sampling technique may allow a large number of cache misses to occur before the offending pages are identified and remapped or migrated. In contrast, the techniques that I propose in Chapter 5 can identify conflicting pages after a relatively small number of cache misses.

2.3 Evaluation of prior work

While the prior research in this area has demonstrated some performance benefits, it falls short of the potential for operating system virtual memory policies to eliminate cache and TLB misses. The fundamental shortcoming of all of the preceding work, except for Sites’s proposal, is that it attempts to derive *static* solutions to what are inherently *dynamic* performance problems. Application memory reference patterns are difficult to predict statically, so optimizing memory system performance for general purpose applications requires dynamic analysis of application behavior. Static mapping policies cannot adapt as the working set of an application changes. There is a class of scientific codes that do have relatively predictable reference patterns, and a body of work devoted to the automatic optimization of memory use for these applications. But even for these applications, static memory management policies cannot detect contention between different applications for memory system resources, or between the operating system and user-level programs [Mogul & Borg 91, Chen 94].

Another optimization technique is to use profiles of past runs to predict the behavior of future runs. The appeal of this approach is that if future behavior is identical to past behavior, it should be possible to use profile information to provide optimal performance. Unfortunately the resulting optimizations will be specific to the program, to the input stream, and to the hardware platform. Attaining the maximum benefit from profile-driven optimization would require case-by-case profiling and optimization for each program, for each hardware platform, and for each input stream. In contrast, the techniques developed in this thesis seek to obtain the benefit of such custom optimizations while using generic policies that apply to all programs and platforms. These policies effectively profile and optimize performance dynamically, obtaining the benefits of profile-driven optimization as programs run rather than afterwards. Although application-specific optimizations will always perform at least as well as generic optimizations, the techniques proposed here often perform as well or nearly as well as optimizations that have exact information about the program's reference patterns, at a far lower implementation cost.

Drawing an analogy from the hardware domain, a cache is a simple and effective device that adapts at runtime to application reference patterns. A cache works because it takes into account the past behavior of the application, and uses the principle of locality to predict (usually correctly) that recently referenced addresses will be referenced again in the near future. However, a cache only considers a very small window onto the application's memory reference stream. As the cost of memory accesses increases relative to processor speed, it will become increasingly attractive to collect more detailed information and employ more sophisticated run time analysis to better manage the memory hierarchy. This thesis, then, explores how to apply the principle of locality within the operating system's virtual memory system to detect and eliminate cache and TLB misses.

In the remainder of this thesis I will describe techniques that go beyond the prior work by taking advantage of dynamic information about application memory reference patterns, and use this information to transparently remap pages, optimizing page sizes and page mappings to eliminate TLB misses and cache conflict misses. The policies introduced in this thesis rely on runtime analysis to trade off the cost of remapping pages with the expected benefits from reduced miss rates. They do not require changes to on-chip hardware, nor are they specialized to particular applications. The net result is operating system virtual memory policies that match the memory system resource requirements of existing applications to the resources available on existing hardware, improving upon static mapping policies in terms of memory consumption, overall execution time, or both.

Chapter 3

Using Virtual Memory to Improve TLB Performance

3.1 Introduction

In this chapter I show how to use the operating system to improve application execution time by dynamically managing the memory system resources provided by the TLB. The techniques I describe rely on variable size pages — superpages — to increase the coverage of the TLB and hence reduce TLB miss rates. A naive strategy for using superpages to improve TLB performance would be to use large pages throughout the address space. While this strategy would increase TLB coverage by increasing the amount of memory mapped by each TLB entry, the internal fragmentation of the large pages would cause memory consumption to increase relative to a system that used only small pages. The contribution of this chapter is to introduce a policy that selectively creates superpages at runtime, providing the good TLB coverage of large pages and the good memory consumption of small pages.

TLB performance can have a dramatic effect on application execution time. To quantify this effect, I measured the percentage of execution time spent handling TLB misses for a collection of programs running on a DEC Alpha 3000/700. (These programs are used as benchmarks throughout this chapter, and are described in Table 3-1.) Table 3-2 shows the results of this experiment: the applications spent between 5.2% and 41.4% of their time in the TLB miss handler.

Improving TLB performance requires either reducing the number of TLB misses incurred by an application, or reducing the cost of individual TLB misses. While some recent research has focused on reducing the cost of TLB misses [Chen et al. 92, Bala et al. 94, Uhlig et al. 94, Austin & Sohi 96], TLB miss paths are already highly optimized, with a single miss costing as little as 10–30 cycles [Kane & Heinrich 92, Dutton et al. 92]. The other alternative for improving TLB performance is to reduce the number of TLB misses. One approach would be to intervene at the level of the user or the language to obtain better locality, but this strategy is limited to applications that can be rewritten or recompiled, and

Table 3-1 This table describes the applications considered in this study. *compress, gcc,* the two *nasa7* programs, *fft*, and *spice* are drawn from the SPEC92 benchmark suite. *Coral* is a benchmark used in the Wisconsin TLB study. *Atom, cecil,* and *fpga* are tools used in the University of Washington research environment. The "working set size" columns shows the total amount of memory (in megabytes) used during the full run of the program, assuming a page size of 4 KB.

Benchmark	Description	Working set size (MB)
coral	The Wisconsin coral database performing a nested join [Ramakrishnan et al. 93].	33.0
compress	*compress* compressing a 977 KB input file.	0.8
nasa7-5	Gaussian elimination for a 3.8 MB matrix.	2.3
nasa7-4	Block tridiagonal matrix solver for a 131.8 KB matrix.	3.1
fpga	Logic bipartitioner targeted to FPGAs, using a 1626 KB input file [Hauck & Borriello 97]	41.6
cecil	*cecil* compiling a 21 KB input file [Chambers 93].	280.4
atom	*atom* instrumenting a 2.6 MB binary with the TLB simulator.	32.7
fft	Fast Fourier transform of a 32 MB array.	32.4
spice	The *spice* circuit simulation benchmark on the 15 KB reference input set	3.2
gcc	*gcc* compiling a 109 KB input file.	1.4

will not benefit programs that have inherently large working sets.

A technique for reducing the number of TLB misses for arbitrary programs is to increase the TLB coverage – the amount of virtual memory mapped by the TLB. TLB coverage is just the product of the number of TLB entries and the page size. For example, the DEC Alpha 21164 has 64 TLB entries for data, which together with a page size of 8 KB results in TLB coverage of 0.5 MB for data. This example points out the fundamental problem with current TLB performance: while modern systems frequently ship with hundreds of megabytes or even gigabytes of RAM to satisfy the requirements of memory-intensive applications, TLB coverage is on the order of one megabyte. Improving TLB coverage requires either increasing either the TLB size or the page size. Increasing the TLB size is unsatisfactory, however: if enough TLB entries are added to allow the TLB to map the working sets of large applications, TLB access time will rise, slowing down *every* memory reference.

Increasing the page size, in contrast, requires only modest hardware support and has the potential to greatly increase TLB coverage – current systems provide superpages as large as 4 MB. Increasing the page size system-wide is not a good solution, however: since pages mapped by the TLB must be physically resident, large pages that are sparsely populated will

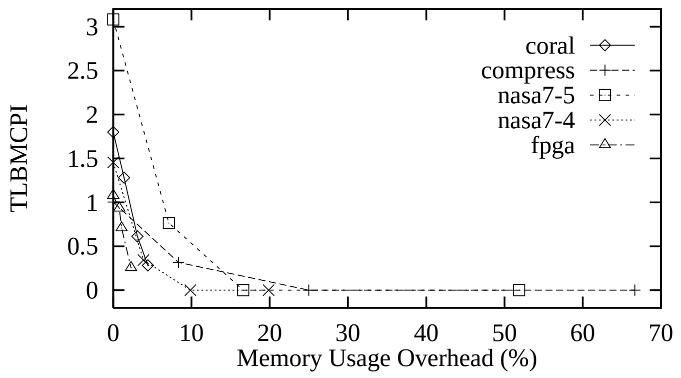
Table 3-2 Baseline performance. I measured the TLB performance of the benchmark suite on a DEC Alpha 3000/700 running DEC OSF/1 2.1. This system contained a 225 Mhz Alpha 21064 processor with a 32 entry DTLB and an 8 entry ITLB, a 2 MB off-chip cache, and 160 MB of main memory. The execution time is the average of five runs on an unloaded system, after an initial run to warm the file buffer cache. I collected the TLB miss statistics using a TLB miss handler instrumented with the Alpha cycle counter.

Benchmark	Exec. Time (s)	TLB Misses (100s)	$\frac{\text{TLB Time}}{\text{Exec. Time}}$ (%)
coral	51.6	1,242,456	41.4
compress	1.1	25,650	35.2
nasa7-5	9.0	205,845	34.7
nasa7-4	1.0	23,787	34.4
fpga	53.8	806,331	28.4
cecil	126.2	969,255	16.6
atom	9.8	94,304	16.1
fft	43.7	294,899	10.7
spice	210.5	1,212,271	9.4
gcc	1.5	4,389	5.2

increase the physical memory requirements of an application. To determine how small and large pages affect system performance, I measured the simulated performance of the benchmark suite on systems that used a range of fixed page sizes. The two metrics of interest are *TLB management cycles per instruction* (*TLBMCPI*) and *memory usage overhead.* TLBMCPI is the total number of cycles spent in service of the TLB divided by the total number of user instructions executed. On a single-issue machine with an infinite cache, overall CPI (cycles per instruction) would be 1+TLBMCPI. Memory usage overhead is the increase in memory consumed relative to a system with fixed-size 4 KB pages, reflecting the cost of memory lost to internal fragmentation.

A larger page size effectively trades higher memory consumption for lower TLBMCPI. Figure 3-1 illustrates this tradeoff, showing TLBMCPI as a function of memory usage overhead as the page size varies. While an ideal system would have both TLBMCPI and memory usage overhead of zero, the figure shows that there is no single page size that approaches this ideal point for any of the applications that have significant TLB overhead. With fixed-sized pages, program behavior is constrained to follow these curves.

The goal of this chapter is to develop an operating system policy that will *selectively* create superpages at runtime, using large pages to map virtual memory where the virtual



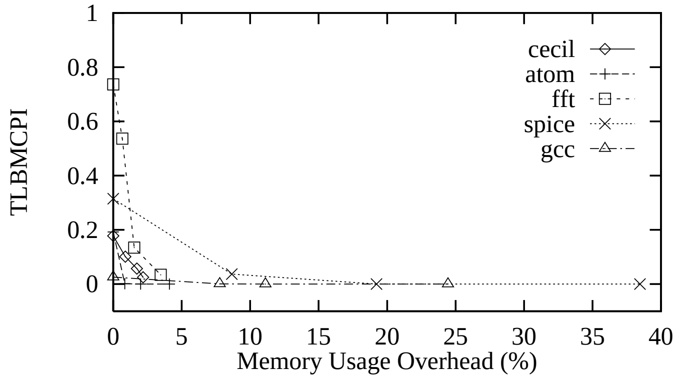


Figure 3-1 TLB overhead as a function of memory usage overhead for a range of page sizes. This graph shows that increasing the page size can sometimes both reduce TLBMCPI and increase memory usage. For each program, each point on the line represents a different page size, ranging from 4 KB (the points with the highest TLB miss rate) to 256 KB (the points with the highest memory usage). These data were gathered using trace-driven simulation. Note that the two graphs use different scales.

address space is used densely and frequently and small pages elsewhere. If such a policy is successful, application behavior will approach the origin on these tradeoff graphs.

Superpages have been used before to increase TLB coverage by mapping large, static regions of memory such as the operating system's text or the window manager's frame buffer, but they have not been used to support general applications. This support has been absent because it is difficult to predict statically whether superpages will benefit a given application: the appropriate set of superpages depends on how an application uses its virtual address space at runtime. Thus it is more effective to create superpages as an application runs.

Creating variable size pages dynamically requires modest extensions to the operating system virtual memory system. Most clients of the virtual memory system continue to see an interface based on a fixed page size, which I will refer to as the *base page size*. The machine-dependent layer of the virtual system must be extended to support two new operations, page *promotion* (or *"construction"*) and *demotion*. Recall that superpages must be contiguous and aligned in both the virtual and physical address spaces. Promotion is the operation of coalescing two or more adjacent virtual pages into a single large superpage. Promotion may involve copying one or more of the component pages to preserve the requirement of physical contiguity. (The experiments in this chapter assume that all of the physical pages must be copied on promotion.) Demotion is the operation of breaking a single page into two or more smaller pages. Demotion typically occurs when a client of the virtual memory system requests some operation on a base page.

While the operating system mechanisms to support superpage creation are straightforward, determining when to use superpages can be challenging. On the one hand, superpages can reduce TLB miss rates and hence execution time because the coverage of a single TLB entry increases as the page size increases. On the other hand, superpages can increase execution time, because pages may need to be copied to make the component pages of a superpage physically contiguous. Therefore, to use superpages effectively, the operating system must balance these factors, ideally creating just those superpages whose performance benefits outweigh their costs.

This chapter shows how to perform this cost-benefit analysis automatically at runtime. I describe online policies that take into account both past miss cost and superpage construction cost in order to determine which pages should be promoted into a larger superpage. Accounting takes place during the handling of TLB misses. While this approach increases the cost of each miss, using trace-driven simulation I show that it substantially reduces the total number of misses, improving overall system performance. Online superpage management policies reduce TLB overhead by as much as 99%, and overall execution time by as much as 48%, when compared to a system using small fixed-size pages. When compared to a system using large fixed-size pages, TLB overhead is approximately the same, but memory consumption is substantially less. The performance of these online policies across a range of applications comes close to that of a nearly optimal offline allocation policy that uses future reference knowledge to allocate superpages.

In summary, this chapter introduces an online policy for superpage management that:

- dramatically improves TLB coverage and hence overall program performance by responding dynamically to application reference patterns;
- does not require special modifications to hardware or changes to the operating system interface;
- operates automatically, requiring no modification to programs or compilers and requiring no hints from the user; and
- transparently improves performance by changing page sizes as required by the application.

The rest of this chapter

In this chapter I describe how I designed and evaluated online superpage management policies. I begin with some comments on the experimental methodology in Section 3.2.

The core of the chapter is the description of the online policies. In Section 3.3 I discuss the principles underlying the design of online superpage promotion policies. Then in Section 3.4, I describe the specific policies based on these principles, as well as several alternate policies to serve as points of comparison.

I evaluate the performance of these policies using trace-driven simulation. In Section 3.5 I describe how the experiments account for the space and time overhead incurred by different policies. In Section 3.6 I describe the performance impact of the various policies. The experiments quantify the benefits of creating superpages as applications run, showing that an online policy improves overall application execution time without the increase in memory consumption that would accompany using large fixed-size pages systemwide.

I conclude the chapter with a discussion of some potential avenues for future work. In Section 3.7 I describe issues that would arise in implementing the policies in Digital Unix on a DEC Alpha. In Section 3.8 I identify some possible refinements and extensions to the policies described in this chapter. Finally, in Section 3.9 I conclude.

3.2 Methodology

I measured system behavior using trace-driven simulation of the benchmarks described in Table 3-1. I used five of the benchmarks (*compress, nasa7-5, nasa7-4, fft,* and *gcc*) to develop and tune the online policies. Once the policy parameters were established, I augmented this "training set" with the remaining five benchmarks (*coral, fpga, cecil, atom,* and *spice*), without any further changes to the parameters. I simulated the TLB behavior of the applications using ATOM, a binary rewriting tool from DEC WRL [Srivastava & Eustace 94]. The simulated system had two 32-entry fully-associative TLBs, one for instructions and one for data, and used a least recently used (LRU) replacement policy. The TLBs support superpages with sizes that are power of 2 multiples of the base page size (4 KB) ranging from 4 KB up to 8 MB.

3.3 Policy design principles

The goal of this chapter is to show how to automatically create beneficial superpages at runtime in order to improve application performance. This is challenging, however, because without full knowledge of the future it is impossible to predict whether constructing a given superpage will in fact be beneficial. This section describes how to design policies that result in good program performance even in the face of incomplete information.

Constructing superpages at runtime carries both a cost and a benefit. The cost is the time required to coalesce pages into a contiguous, aligned region of physical memory. The benefit is the elimination of future TLB misses. With complete information about the future reference stream, it would be possible to design an optimal offline policy: that is, a policy that constructs exactly the set of superpages whose benefit outweighs their construction costs. Although this goal is unobtainable in practice, by applying principles drawn from competitive analysis it is possible to design online policies that have total TLB management costs that are bounded relative to the optimal offline policy. The policies proposed here do not explicitly attempt to optimize memory consumption relative to a system with small fixed-size pages. In practice, the online policies tend to create superpages in densely populated regions, so the increase in memory consumption is small.

Constructing superpages can also affect the performance of a physically-indexed cache, because larger pages limit the operating system's flexibility in mapping virtual pages to cache pages. In this chapter I assume that the operating system consistently uses page coloring to map virtual pages to physical pages, regardless of the page size. With this mapping policy the construction of superpages will not affect the behavior of a physically-indexed cache for memory references within a single virtual address space. In Chapter 6 I consider the interaction of superpage construction and cache performance for mapping policies other than page coloring.

3.3.1 The ski-rental problem

My approach to designing online superpage construction policies can be understood by considering an analogous situation that arises in the "ski-rental problem."[1] Consider a graduate student who occasionally goes cross-country skiing to avoid working on his thesis. He can either buy skis for \$100; or rent skis for \$10. With complete information about the future, the problem is straightforward: he should buy skis if he will go skiing 10 or more times, and rent skis otherwise. In practice, of course, the student does not know how long the ski season will last, or how long he'll be in graduate school, or how often he'll choose to procrastinate. Nonetheless, he would like to bound his total expenditures on ski equipment. To do this he should begin by renting skis. If he eventually goes on an eleventh ski trip, he should then buy skis. This policy guarantees that he will never spend more than twice as much as the optimal ski-rental policy, no matter how many times he goes skiing. (In the worst case, the optimal offline policy would have spent \$100, whereas the graduate student would have spent \$200.)

To apply a similar approach to the superpage construction problem, an online policy must take into account the cost and benefit of each superpage. In effect, the skiier in the example above maintained a small amount of bookkeeping information that reflected the past cost of not owning skis. Once this past cost equalled the cost of buying skis, he made the purchase.

Similarly, the online superpage construction policies will maintain information reflecting the number of TLB misses each superpage could have prevented had it been constructed earlier in time. A policy analogous to the one proposed for the cross-country skiier is to promote to a superpage when the cost of past TLB misses that the superpage would have prevented equals the cost of constructing the superpage. With respect to this superpage, such a policy will have total TLB management costs at most twice that of the optimal offline superpage construction policy.

The rest of this section describes how to maintain counters that reflect the history of past TLB misses, and how to select a promotion threshold based on these counters.

[1] This analogy was originally suggested by Larry Rudolph in explanation of the results described in [Karlin et al. 88].

3.3.2 Maintaining miss charges

Designing a superpage construction policy based on the ski-rental analogy requires "charging" each TLB miss to any superpage that could have prevented the miss. Once the charges to a given superpage reach some threshold value – such as the cost of creating the superpage – the superpage is promoted. I now describe the details of how TLB misses are charged to specific superpages.

A superpage can prevent a miss to a page p in one of two ways. First, the superpage may cover both p and a page already in the TLB. In this case the miss would have been prevented if the superpage had already been constructed, because the translation for the page already in the TLB would have covered p. Second, a superpage may increase the capacity of the TLB, so that a prior translation for p would not have been evicted from the TLB, avoiding the miss. Preventable misses can be tallied by maintaining two counters that keep track of these two different miss charges to a potential superpage P: *prefetch*(P) and *capacity*(P). The counters are updated on a TLB miss to a page p as follows:

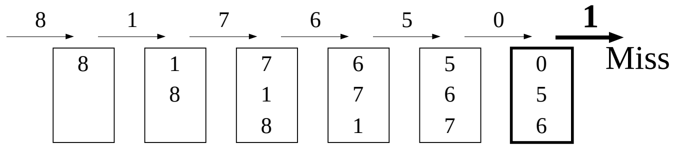
- *Prefetch* charges are updated on each miss by incrementing *prefetch*(P) for each potential superpage P that contains the currently referenced page p and one or more TLB entries.

- *Capacity* charges are computed by examining the TLB miss stream at the time of the reference to p. *Capacity*(P) is incremented for each potential superpage P that coalesces enough TLB entries to have kept p from having been evicted from the TLB in the past. (The TLB miss stream is maintained in a LRU stack data structure that reflects the set of referenced pages in order of most recently referenced to least recently referenced. That is, if the LRU stack at some time is $(p_1, p_2, \ldots, p_n)$, then p_i is the *i*th most recently referenced page.) [2]

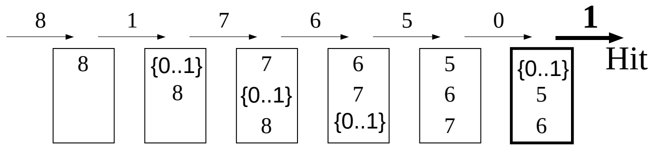
Figure 3-2 illustrates the use of these counters. Consider a TLB with three entries. Immediately after servicing the stream of virtual page references 8,1,7,6,5,0 the TLB contains translations for pages 0, 5 and 6. Suppose the next reference is to virtual page 1, resulting in a miss. If the system had created superpage $\{0, 1\}$ initially, this miss would not have

[2] The LRU stack is described as a totally ordered list to simplify the exposition. Since information about the LRU stack is updated on a TLB miss, it is only possible to record the *set* of pages that have translations in the TLB in addition to the precise LRU stack of pages that do not have translations in the TLB. This information is sufficient, though, to accurately update capacity charges.

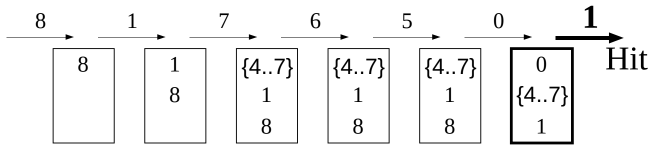
occurred: on the prior reference to page 0, a translation for $\{0, 1\}$ would have entered the TLB, and the subsequent reference to page 1 would have hit. Therefore, *prefetch*($\{0, 1\}$) is incremented. Alternatively, if the system had created superpage $\{4, 5, 6, 7\}$ initially, this miss would not have occurred. This can be seen by observing the entire LRU stack prior to the reference to page 1. In order of most recently to least recently referenced, the LRU stack contains 0,5,6,7,1,8. If $\{4, 5, 6, 7\}$ had been initially promoted, the LRU stack would reduce to 0,$\{4, 5, 6, 7\}$,1,8. Page 1 is now one of the top 3 entries in the stack and, under LRU replacement, its translation would be in the TLB. Since the promotion of $\{4, 5, 6, 7\}$ would have prevented virtual page 1 from being evicted from the TLB, *capacity*($\{4, 5, 6, 7\}$) is incremented.



(a) *A sequence of TLB references with no superpages. In this scenario, the final reference to page 1 results in a TLB miss.*



(b) *The same sequence of TLB references with the superpage {0,1}. In this scenario, the reference to page 0 effectively* prefetches *the TLB entry for page 1, so that the final reference is a TLB hit.*



(c) *The same sequence of TLB references with the superpage {4,5,6,7}. In this scenario, the existence of this large superpage effectively increases the* capacity *of the TLB, so that the final reference to page 1 results in a TLB hit.*

Figure 3-2 Example of prefetch and capacity charges. Each part of the figure shows the changing contents of the TLB when presented with the reference stream of virtual pages 8,1,7,6,5,0,1. In part (a), the final reference to page 1 results in a TLB miss. Part (b) shows why superpage {0,1} incurs a prefetch charge: had the superpage {0,1} existed, the final reference would have resulted in a TLB hit. Similarly, part (c) shows why superpage {4,5,6,7} incurs a capacity charge: had the superpage {4,5,6,7} existed, the final reference would have resulted in a TLB hit.

The two counters reflect substantially different information. The promotion of $\{0, 1\}$ in the previous example effectively causes a prefetch of the translation for virtual page 1, since the first reference to page 0 causes a translation for $\{0, 1\}$ to enter the TLB. On the other hand, the promotion of $\{4, 5, 6, 7\}$ effectively increases the capacity of the TLB by reducing a set of entries in the LRU stack to a single entry, thereby preventing the prior eviction of some other entry.

In Section 3.5 I show that in practice the prefetch counter is more important than the capacity counter in identifying the best potential superpage. Fortunately, the prefetch counter is the less expensive of the two to maintain. A simple way to compute prefetch charges is to scan the TLB on a miss to page p and check if some potential superpage contains both p and a current TLB entry. A more efficient algorithm for maintaining prefetch charges is given in Section 3.5.2. On the other hand, determining capacity charges requires scanning both the TLB and the LRU stack of pages referenced prior to the last reference of p to determine if there is a potential superpage that would have coalesced enough entries to have prevented p's eviction.

Resetting miss charges on promotion

The miss charge of a potential superpage should reflect the number of misses that promotion would have eliminated that were not *already* eliminated by previously promoted pages. So in addition to updating the counters on each TLB miss, the counters must be updated when promoting to a page P, to reflect the new, perhaps diminished benefit of subsequent promotions. To adjust the prefetch charges when P is promoted, $\mathit{prefetch}(P')$ is decremented by $\mathit{prefetch}(P)$ for all superpages P' containing P, since whenever $\mathit{prefetch}(P)$ is incremented, $\mathit{prefetch}(P')$ is also incremented. Adjusting the capacity charges when P is promoted is more difficult: for every superpage Q, $\mathit{capacity}(Q)$ must be decremented for each past miss that incremented both $\mathit{capacity}(Q)$ and $\mathit{capacity}(P)$.

3.3.3 Threshold selection

Given counters that associate past TLB misses with specific superpages, the next step is to choose a threshold value for promoting superpages. Intuitively, the goal in choosing this threshold is to ensure that a superpage is promoted early enough to avoid future misses, but late enough to ensure that the cost of promotion does not dominate TLB overhead. More rigorously, by using a carefully selected threshold, an online superpage construction policy can bound its cost relative to the optimal offline policy, as follows.

An optimal offline superpage construction policy has the luxury of observing all future references prior to making a promotion. Suppose that there is one candidate N KB superpage P consisting of two $N/2$ KB pages, and that in the run of an application, m TLB misses will become hits if P is constructed at time 0. The optimal offline policy, knowing m, will perform the promotion at time 0 if m is at least the ratio of P's construction cost (cycles lost due to copying) and the cost of a TLB miss. This ratio is called $R = \text{Construction Cost}/\text{TLB Miss Cost}$. The optimal offline policy will never promote to P if $m < R$. An online policy that promotes when the miss charges to a page reach R is guaranteed to deliver performance no worse than twice the optimal offline with respect to this single superpage. If $m \leq R$, then both the online and offline policies pay for m TLB misses. If $m > R$, the offline policy pays for a promotion at time 0, while the online policy pays for R TLB misses *and* the promotion, for a total cost of two promotions.

In practice, the online policy of waiting until the miss charges exceed R is overly conservative. Because of locality in page reference patterns, most potential superpages either never accumulate miss charges exceeding a fraction of R, or else accumulate miss charges greatly exceeding R. Therefore, a promotion threshold substantially less than R will work well. In Section 3.4.2 I present data that support this hypothesis.

The effect of bookkeeping overhead on threshold selection

The analysis presented above does not take into account the additional overhead incurred on every TLB miss by an online superpage construction policy. Intuitively, it might seem that given this additional cost, online policies should use a promotion threshold different from R in order to best bound their overhead compared to the optimal offline algorithm. However, I show here that this is not the case. Even when the additional bookkeeping overhead is included in the analysis, the online policy should still use a promotion threshold of R to minimize its overhead compared to the optimal offline policy.

Consider again the case in which constructing superpage P at time 0 would prevent m TLB misses, at a construction cost of C. Letting b be the baseline TLB miss cost, the ratio of construction cost to TLB miss cost is $R = C/b$. The optimal offline policy, knowing m, will perform the promotion at time 0 if m is at least R, and will never promote to P if $m < R$.

The remaining question is what value of *thresh*, the promotion threshold for the superpage, will minimize the worst-case ratio assuming that the online algorithm incurs an additional overhead of o time units on each TLB miss. The worst-case ratio occurs when the online algorithm promotes a page that will in fact never prevent any future misses.

This ratio will occur for a page P that would prevent exactly *thresh* misses — that is, when $m = \mathit{thresh}$.

In this case, the cost to the online algorithm is $(b + o) \cdot \mathit{thresh} + C$, and the cost to the offline algorithm is $\min(b \cdot \mathit{thresh}, C)$. Thus the ratio of the online cost to the offline cost is

$$\begin{aligned} &\max\left(\frac{(b+o)\cdot \mathit{thresh} + C}{b \cdot \mathit{thresh}}, \frac{(b+o)\cdot \mathit{thresh} + C}{C}\right) \\ &= \max\left(1 + \frac{o}{b} + \frac{C}{b \cdot \mathit{thresh}}, 1 + \frac{(b+o)\cdot \mathit{thresh}}{C}\right). \end{aligned}$$

To see that the online algorithm should use a threshold of $R = C/b$, consider the ratio of the online cost to the offline cost for each of three cases: (1) $\mathit{thresh} = C/b$, (2) $\mathit{thresh} < C/b$, and (3) $\mathit{thresh} > C/b$.

1. For $\mathit{thresh} = C/b$, the ratio of the online cost to the offline cost is:

$$\begin{aligned} &\max\left(1 + \frac{o}{b} + \frac{C}{b \cdot \mathit{thresh}}, 1 + \frac{(b+o)\cdot \mathit{thresh}}{C}\right) \\ &= \max\left(1 + \frac{o}{b} + 1, \frac{(b+o)}{b} + 1\right) \\ &= 2 + o/b. \end{aligned}$$

2. For $\mathit{thresh} < C/b$, the ratio of the online cost to the offline cost is:

$$\begin{aligned} &\max\left(1 + \frac{o}{b} + \frac{C}{b \cdot \mathit{thresh}}, 1 + \frac{(b+o)\cdot \mathit{thresh}}{C}\right) \\ &\geq 1 + \frac{o}{b} + \frac{C}{b \cdot \mathit{thresh}} \\ &> 2 + o/b. \end{aligned}$$

3. For $\mathit{thresh} > C/b$, the ratio of the online cost to the offline cost is:

$$\begin{aligned} &\max\left(1 + \frac{o}{b} + \frac{C}{b \cdot \mathit{thresh}}, 1 + \frac{(b+o)\cdot \mathit{thresh}}{C}\right) \\ &\geq 1 + \frac{(b+o)\cdot \mathit{thresh}}{C} \\ &> 2 + o/b. \end{aligned}$$

Hence, the promotion threshold for P that minimizes the online to offline performance ratio in the worst case is R. With this threshold, the online policy incurs at most $2 + o/b$ times the cost incurred by offline policy with respect to P.

3.3.4 Policy design principles: summary

The high-level goal of an online superpage construction policy is to automatically identify and construct beneficial superpages as an application runs. To achieve this goal, an online policy needs to "blame" each TLB miss on specific superpages. In particular, it must maintain one or more counters for each potential superpage, indicating how many past TLB misses the superpage could have prevented. By waiting to promote a given superpage until a threshold value is reached, the operating system can avoid the performance problems inherent in promoting too soon (high copying costs) or too late (high TLB miss rates). Further, this strategy results in a policy that bounds overall TLB management overhead relative to the optimal offline policy that has full knowledge of the future. In the following sections I will show how a policy based on these principles improves application performance.

3.4 Promotion policies

Using the policy design principles described in the previous section, I now introduce several TLB management policies that perform cost-benefit analysis in order to selectively construct superpages. As an initial reference point, I also include one offline policy, which constructs superpages based on complete knowledge of the application's memory reference stream. I then consider online policies that maintain the counters described in Section 3.3 in order to selectively construct superpages. The goal of the online policies is to approach the performance of the offline policy without using advance knowledge. Also, in order to evaluate whether the overhead associated with collecting the information needed by the online policies is justified, I also consider "oblivious" policies, which construct superpages *without* considering their cost and benefit.

The rest of the section motivates and describes each of the policies in more detail. The simulation results in Section 3.6 show that well-designed online policies can, with low overhead, eliminate nearly as many TLB misses as an offline policy, despite having no information about the future reference pattern of the application.

3.4.1 OFFLINE – *an approximation to the optimal offline policy*

A good lower bound on the performance of online superpage construction policies would be the performance of the optimal offline policy, which constructs a set of superpages that minimizes total TLB management costs using complete knowledge of the future. Unfortunately, computing the optimal offline solution is NP-complete by reduction from Set-Cover.

An alternate lower bound is provided by a greedy approximation to the optimal solution called *OFFLINE*. Like the optimal offline policy, *OFFLINE* eliminates future TLB misses by creating appropriate superpages. Unlike the optimal solution, however, it greedily selects the superpages to promote. Initially, *OFFLINE* analyzes the TLB miss stream and promotes the superpages with the highest ratio of misses eliminated to promotion cost. *OFFLINE* then continues promoting incrementally, examining a revised TLB miss stream based on the current set of superpages, iterating until it can find no superpages whose benefit exceeds the promotion cost.

The simulations presented in Section 3.6 show that *OFFLINE* reduces total TLB management overhead to less than 0.1 MCPI for all of the workloads I measured, indicating that it does in fact provide a reasonable approximation to the performance of the optimal offline promotion policy. Thus *OFFLINE* provides an approximate lower bound on the performance of online promotion policies.

3.4.2 ONLINE – *the basic online algorithm*

The algorithm *ONLINE* is based on the prefetch and capacity counters described in Section 3.3. Once the miss counters for a superpage reach a certain threshold, *ONLINE* promotes the superpage. To determine an appropriate threshold, I analyzed the distribution of final miss charges for all potential superpages that were *not* constructed by *OFFLINE*. For these unpromoted superpages, 99% of the capacity charges were less than $0.625 \times R$, and 99% of the prefetch charges were less than $0.125 \times R$, where R is the ratio of promotion cost to TLB miss cost. An online policy that uses promotion thresholds above these values is unlikely to promote unnecessarily, whereas one that uses thresholds below may. Therefore, *ONLINE* promotes to a superpage P if $capacity(P)$ exceeds $0.625 \times R$, or $prefetch(P)$ exceeds $0.125 \times R$. When a promotion occurs, the policy decrements prefetch charges for each superpage containing the promoted superpage, as described in Section 3.3. The policy does not decrement capacity charges precisely on a promotion, as properly resetting them requires keeping track of the capacity counters to which each miss contributed. Because this involves considerable bookkeeping overhead, capacity charges are instead simply reset to zero on a promotion. Although this is overly conservative, I found that the results are relatively insensitive to how sharply capacity charges were reduced following a promotion.

3.4.3 APPROX-ONLINE – *an approximate online algorithm*

While *ONLINE* collects precise information about the causes of each TLB miss, this precision comes at the cost of high bookkeeping overhead for maintaining the capacity counters. The algorithm *APPROX-ONLINE* reduces overhead while potentially reducing accuracy: it is the same as *ONLINE* except that it only maintains prefetch charges. As with *ONLINE*, when $\mathit{prefetch}(P)$ reaches $0.125 \times R$, the policy creates superpage P and decrements the *prefetch* counters by the threshold value for larger superpages containing P. Section 3.6 will show that *APPROX-ONLINE* is in fact almost exactly as effective as *ONLINE* in eliminating TLB misses, with much lower bookkeeping overhead.

3.4.4 ASAP – *the as-soon-as-possible algorithm*

The online promotion policies described so far add overhead to every TLB miss. To determine whether this runtime bookkeeping and analysis is valuable, I also considered two policies that create superpages but do not evaluate the expected costs and benefits of promotion.

ASAP promotes a superpage as soon as all of its component base pages have been referenced. Although it requires minimal bookkeeping, it has two potential drawbacks. First, the policy may copy too frequently, since it creates superpages without concern for promotion cost. Even if pages are rarely referenced, it will still pay the cost to create a superpage. Second, *ASAP* may fail to create beneficial superpages if any one of the component base pages is not referenced.

ASAP-4-64 is a variant of *ASAP* based on a policy used in Talluri's thesis [Talluri 95]. *ASAP-4-64* allows only two page sizes: 4 KB and 64 KB. The policy promotes to a 64 KB superpage as soon as half of the component 4 KB pages have been referenced. The measurements here differ from Talluri's in the way they account for reservation. Talluri is optimistic, assuming that reservations at the granularity of 64 KB always succeed. This effectively makes promoting to 64 KB pages free. In contrast, I conservatively assume that reservation is not used, so that promotions always require copying all of the component pages. A more detailed assessment of the impact of reservation is difficult, since the effectiveness of reservation depends on the level of contention for physical memory, and this in turn depends on the behavior of the entire system, not just a single application.

3.4.5 Promotion policies: summary

Together, the policies described in this section cover a cross-section of the design space for superpage promotion policies. Specifically, the policies vary in the amount of information they use to make promotion decisions. At one end of the spectrum, *ASAP* and *ASAP-4-64* only examine the set of pages in use, but not the frequency or interleaving of references to those pages. At the other end of the spectrum, *OFFLINE* uses complete knowledge of the future memory reference stream to construct a near-optimal set of superpages. In between, *ONLINE* and *APPROX-ONLINE* dynamically monitor the TLB miss stream, attributing each TLB miss to one or more potential superpages. By applying the policy design principles described in Section 3.3, these policies can identify and construct superpages in order to substantially reduce TLB miss rates without incurring excessive overhead. In Section 3.6 I present data that support this claim.

3.5 Policy space and time costs

For a TLB management policy to be effective, it must have reasonable time and space overheads. For example, while it is acceptable to increase the time to handle an individual TLB miss in exchange for reduced TLB miss rates, there is still a limit on the acceptable cost of a single TLB miss. In this section, therefore, I describe how I accounted for the time and space costs of the various policies, and I describe an algorithm for efficiently maintaining the prefetch counters used by the online policies. The performance results presented in Section 3.6 include these sources of overhead.

3.5.1 Baseline costs

Every TLB miss requires a page table lookup to satisfy the cause of the miss. To account for this overhead, every policy is charged 30 cycles per TLB miss, in addition to any policy-specific bookkeeping overheads. This is consistent with TLB miss costs on current systems. For example, I measured a minimum TLB miss penalty on a DEC Alpha 3000/700 of 31 cycles. All of the promotion policies also incur overhead when creating a new superpage. Each 1 KB of memory copied incurs a charge of 3,000 cycles. This cost reflects both the instructions to perform the copy and stall cycles due to cache misses that may occur during the copy.

3.5.2 Maintaining prefetch charges

I describe here the details of an algorithm for maintaining the prefetch counters on each TLB miss, as required by the *ONLINE* and *APPROX-ONLINE* policies, without incurring excessive space or time overhead. The algorithm executes on average 100 instructions per TLB miss to maintain the counters, and incurs a memory overhead of at most 0.3%.

Recall that the prefetch counters are updated as follows. On a TLB miss to a page p, *prefetch*(P) is incremented for each superpage P that covers both p and some page already in the TLB. To avoid the overhead of scanning the contents of the TLB on each miss, the system maintains an additional counter for each superpage P, called *tlbcount*(P), that indicates whether or not P or one of its component pages is currently in the TLB. This information allows the prefetch charges to be updated quickly on a miss.

tlbcount(P) takes on one of four values, and satisfies the following invariants:

- If P is part of a larger superpage that has already been promoted, *tlbcount*$(P) = -1$.
- If P and none of its component pages are in the TLB, *tlbcount*$(P) = 0$.
- If a translation for P is in the TLB, *tlbcount*$(P) = 1$.
- If a translation for a subpage of P is in the TLB and only one of *tlbcount*(P_1) and *tlbcount*(P_2) is positive, where P_1 and P_2 are the two component subpages of P, *tlbcount*$(P) = 1$.
- If a translation for a subpage of P is in the TLB and both *tlbcount*(P_1) and *tlbcount*(P_2) are positive, where P_1 and P_2 are the two component subpages of P, *tlbcount*$(P) = 2$.

On a miss to p, the invariants are maintained and the prefetch counters are updated as follows. Let $p = p_0, p_1, \ldots p_m$ be the set of superpages containing p, in increasing order of size. (The size of p_m is the maximum superpage size.) For each p_i with *tlbcount*$(p_i) = 0$, and the smallest i with *tlbcount*$(p_i) > 0$, *tlbcount*(i) is incremented. For all i with *tlbcount*$(p_i) > 0$, *prefetch*(i) is incremented. Suppose that the translation for q is replaced in order to bring p into the TLB. Let $q = q_0, q_1, \ldots, q_r$ be the superpages containing q in increasing order of size, such that q_r is the smallest one with *tlbcount*$(q_r) = 2$. Then for each q_i, $0 \leq i \leq r$, *tlbcount*(q_i) is decremented.

Finally, if a page P is promoted, *tlbcount*(P') is set to -1 for each P' component to P. (Note that *tlbcount*$(P) > 0$ already.) If a page P is demoted, *tlbcount*(P) is reset to 0.

While the algorithm as described so far avoids the overhead of scanning the TLB on every miss, it still updates a counter for pages as large as the maximum page size. Since these large superpages are promoted only rarely, the bookkeeping overhead of the algorithm can be reduced further by maintaining precise prefetch charges for large pages only when they are candidates for promotion.

For example, in the simulations, *APPROX-ONLINE*'s promotion threshold for a 128 KB superpage is 1600 ($0.125 \times R$, where R is the ratio of promotion cost to TLB miss cost for a 128 KB page). Thus it is impossible that a 128 KB superpage will be promoted until at least 1600 TLB misses have occurred. Rather than performing 1600 individual updates to superpages of sizes 128 KB through 8 MB (the maximum superpage size), the updates can be deferred until the system experiences 1600 TLB misses, and the updates can be performed all at once, sharply reducing the average overhead per TLB miss.

Implementing an algorithm based on this idea of "batching" updates requires choosing how accurately to maintain the prefetch counters. If the system maintains precise prefetch counters for pages that are too large, then the overhead of each TLB miss remains too high. On the other hand, if the system maintains precise information only for very small pages, then the batched updates will have to be performed too frequently.

I simulated an algorithm that uses the *tlbcount* counters and batched updates to reduce the overhead per TLB miss. I found that propagating updates eagerly to superpages of size 64 KB or less and deferring updates to larger superpages resulted in the best overall performance. This approach requires an additional counter, *deferred*(P), that reflects the updates to prefetch counters that have not yet been propagated to superpages containing P. The algorithm maintains *deferred*(P) for all superpages of size 64 KB and larger.

When I profiled the execution of this algorithm with the TLB miss stream of the benchmark suite, I found that on average it executed 100 instructions per TLB miss. The algorithm as implemented also consumes space: one *prefetch* counter for every potential superpage, one *tlbcount* counter for every potential superpage and every base page, and one *deferred* counter for every superpage of size 64 KB or larger. Altogether this results in an average of 3.125 counters per base page. Thus with 4 KB base pages, even at one four-byte word per counter, the space overhead is $(3.125 * 4)/4096$ (0.4%) to maintain the prefetch counters.

3.5.3 Maintaining capacity charges

The *ONLINE* policy maintains capacity charges as well as prefetch charges. I describe here a simple algorithm to estimate of the cost of maintaining the capacity counters. Since in practice the capacity counters do not significantly affect the ability of the online policy to eliminate TLB misses, I did not refine the algorithm further.

Recall that the capacity count is incremented for each superpage that would have coalesced enough TLB entries to prevent the current miss to page p. The capacity count is incremented as follows. The number of TLB entries that must be coalesced depends on the depth of the previous reference to p in the LRU stack, referred to as *lru-depth*. The pseudo-code below computes the number of TLB entries covered by each superpage Q, storing the result in the variable *coverage*(Q). The pseudo-code then increments *capacity*(Q) for every potential superpage Q that covers at least *lru-depth* TLB entries.

lru-depth:	the depth of the page that missed in the LRU stack
coverage(Q):	number of TLB entries covered by superpage Q, initially 0.

```
for each page $P$ in the TLB {
    for each superpage $Q$ containing $P$ {
        increment coverage($Q$);
        if ( coverage($Q$) == lru-depth ) {
            increment capacity($Q$);
            check for promotion;
        }
    }
}
```

In a system with 32 TLB entries and page sizes ranging from 4 KB to 8 MB (*i.e.* each base page is contained in 11 superpages), the body of the loop will be executed at least 32*11 = 352 times. With the assumption that the loop body requires 7 instructions (load, increment, store, subtract, conditional branch, update index, conditional branch), updating the capacity counters will require $352 * 7 = 2464$ instructions per TLB miss. In terms of space, the algorithm requires an average of one counter per base page, resulting in memory overhead of $(1 * 4)/4096$ (0.1%) to maintain the capacity counters.

3.5.4 Policy costs: summary

Table 3-3 summarizes the TLB miss cost incurred by each of the policies evaluated in this chapter, determined as follows. All of the policies incur the baseline cost of 30 cycles per miss. *APPROX-ONLINE* must also maintain the prefetch counters, adding 100 cycles per miss for a total TLB miss cost of 130 cycles. *ONLINE* additionally maintains the capacity counters, adding approximately 2470 cycles for a total of 2600 cycles per miss. *ASAP* and *ASAP-4-64* are not charged any overhead beyond the baseline miss cost, since the necessary bookkeeping can be performed as pages are mapped. Likewise, *OFFLINE* does not incur extra overhead, since it can perform all of its accounting computation offline.

These cycle counts assume that the bookkeeping code is not slowed by cache misses. When I simulated the cache effects of counter updates (using the memory system simulator described in Chapter 4), I found that cache misses on average doubled the number of cycles to update the counters. There are a number of potential optimizations that could compensate for this increase in the counter update cost. For example, rather than reacting to every TLB miss, the TLB miss stream could be sampled. Sampling every other miss, for example, would reduce bookkeeping cost by a factor of two. Another optimization would be to reduce the range of potential page sizes, decreasing the number of levels in the tree of counters updated on each miss. I do not attempt to measure the precise effect of such optimizations here. However, I expect that taking into account both the effect of memory system delays and optimizations on the cost of counter updates, the bookkeeping costs shown in Table 3-3 are in fact pessimistic estimates.

Table 3-3 Cost of a single TLB miss for each policy, computed as baseline TLB miss cost, plus any bookkeeping cost. Sections 3.5.2 and 3.5.3 describe the calculation of the bookkeeping cost for *ONLINE* and *APPROX-ONLINE*. The value for *ONLINE* reflects the high overhead of maintaining capacity counters. *ASAP* and *ASAP-4-64* can perform the necessary bookkeeping as pages are mapped, while *OFFLINE* can perform its bookkeeping offline, so these three policies have the same overhead as *FIXED*.

Policy	TLB miss cost (cycles)
FIXED (4 KB, 16 KB, 64 KB)	30
OFFLINE	30
ONLINE	2600
APPROX-ONLINE	130
ASAP	30
ASAP-4-64	30

3.6 Simulation results

To evaluate the effectiveness of constructing superpages at runtime using cost-benefit analysis, I used trace-driven simulation of the benchmarks described in Section 3.2. I simulated each of the policies introduced in Section 3.4. For comparison, I also simulated policies that use fixed-size pages, referred to as *FIXED 4 KB*, *FIXED 16 KB*, and *FIXED 64 KB*. This section describes the impact of each of the policies on TLB miss rates, TLBMCPI, and memory consumption. The results show that the online policies are nearly as effective as large fixed size pages in eliminating TLB misses, without incurring the memory consumption costs of large pages. Further, the *APPROX-ONLINE* policy performs nearly as well as the *OFFLINE* policy, despite operating without knowledge of the future reference pattern of the application.

3.6.1 Impact on TLB miss counts

Table 3-4 shows the impact of the policies described in Section 3.4 on TLB miss counts. All of the promotion policies are capable of eliminating the majority of TLB misses for many of the programs. However, *ASAP* performs poorly for programs with fragmented memory

Table 3-4 The number of TLB misses, in 100s, that occurred using each policy. These numbers are proportional to the "TLB Miss Handler CPI" components of the bar graphs shown in Figure 3-3.

Benchmark	*FIXED 4 KB*	*FIXED 16 KB*	*FIXED 64 KB*	*OFFLINE*	*ASAP*	*ASAP-4-64*	*ONLINE*	*APPROX-ONLINE*
coral	1,205,951	858,980	410,424	405	1,193,847	412,253	12,517	16,107
compress	29,198	9,201	0	1	2	1	85	117
nasa7-5	407,140	100,880	5	3	6	8	249	249
nasa7-4	30,044	7,084	3	38	8	18	316	352
fpga	744,183	643,967	487,609	251	734,965	489,516	8,827	10,446
cecil	482,885	274,927	156,059	42,006	433,048	158,025	28,731	35,497
atom	49,511	428	70	930	1,164	128	864	1,041
fft	317,447	231,161	58,162	10	83	58,199	4,112	4,112
spice	1,852,305	215,924	355	3	1,002,570	380	605	823
gcc	1,207	48	0	237	21	2	187	232

access patterns, such as *coral, fpga, cecil,* and *spice*, since it never promotes a superpage that is not fully populated. For *coral, fpga, cecil* and *fft*, the policies that impose too small a maximum page size (the *FIXED* policies and *ASAP-4-64*) do not result in a large reduction in the miss counts. The table also shows that *ONLINE* and *APPROX-ONLINE* sometimes cause slightly more TLB misses than the two *ASAP* policies because the online policies defer promotion until there is evidence that it will be beneficial. Finally, the table shows that

ASAP and *ASAP-4-64* sometimes promote pages that *OFFLINE* does not, indicating that these two policies create some superpages that do not eliminate enough misses to recover the promotion cost.

3.6.2 Impact on TLBMCPI

The TLB miss counts alone are not enough to evaluate the effect of the different policies, because they do not include the bookkeeping and page copying overheads incurred by the promotion policies. Figure 3-3 provides a more accurate depiction of the impact of the policies on performance. It shows the TLBMCPI for each application and each policy, including these sources of overhead. The figure illustrates that:

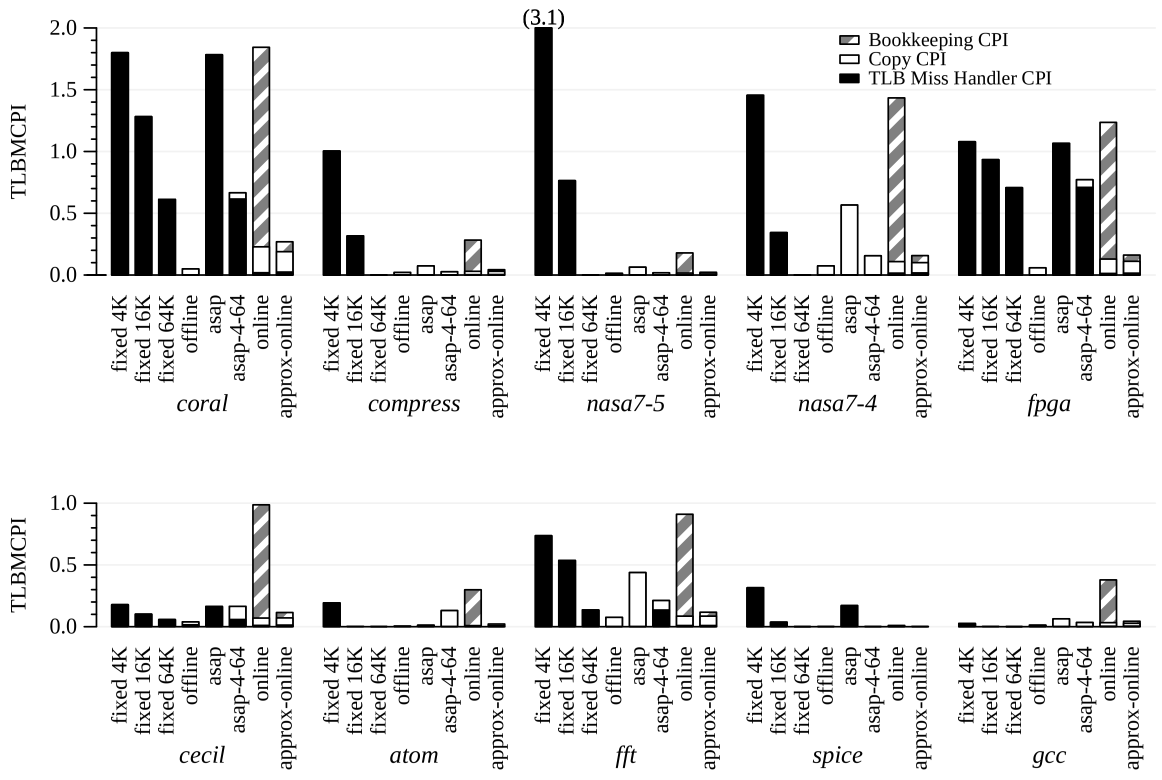


Figure 3-3 This figure shows the time overhead due to TLB management for policies using fixed and variable-sized pages. The fixed-size page policies incur a fixed overhead on each TLB miss (TLB Miss Handler CPI). The variable-size page policies also incur promotion overhead when superpages are created (Copy CPI), and bookkeeping overhead on each TLB miss (Bookkeeping CPI).

- *OFFLINE* reduces TLBMCPI substantially when compared to *FIXED 4 KB* or *FIXED 16 KB*.

- Although *APPROX-ONLINE* does not perform as well as *OFFLINE*, for most of the workloads its performance is significantly better than a policy that uses small fixed-size pages.

- For *APPROX-ONLINE* the largest component of TLBMCPI is page promotion cost, which reflects the overhead of physically coalescing pages. This indicates that both baseline TLB miss cost and bookkeeping overhead can be negligible in a system that eliminates most TLB misses, as long as the bookkeeping overhead per TLB miss is not excessive.

- While *ONLINE* has similar performance to *APPROX-ONLINE* in terms of TLB miss rates, the high cost of maintaining the capacity counters diminishes or even overwhelms the benefit of reduced TLB misses. This indicates that the prefetch counters are more relevant than the capacity counters in capturing an application's current working set, and that the overhead of maintaining the capacity counters is not justified.

- *ASAP-4-64* achieves an overall TLBMCPI that is close to *APPROX-ONLINE* except for applications with extremely large working sets such as *coral*, *fpga*, *atom* and *fft*. For these applications, *ASAP-4-64*'s imposition of a maximum superpage size of 64 KB results in performance worse than that of *APPROX-ONLINE*.

- Some workloads, such as *gcc*, do not benefit from larger pages. This is unsurprising, since *gcc* has a low TLB miss rate even with 4 KB pages. In such cases, no policy performs better than one that uses fixed-size 4 KB page. It would be straightforward to modify the online policies so that they remain inactive until the TLB miss rate exceeds some threshold.

3.6.3 Execution time

While the TLBMCPI measurements focus on the effect of superpage management on TLB performance, it is also important to understand the effect of the various policies on end-to-end execution time. To estimate this effect, I first measured the *non*-TLB execution time of each application on the DEC Alpha. I instrumented the TLB miss handler with the Alpha cycle counter, and then subtracted the total time spent in the miss handler from the execution time. I then added the simulated TLB overhead for each program and policy to the non-TLB execution time. Finally, I computed the percentage improvement of each policy relative to a system with fixed-size 4 KB pages. The results of this computation are shown in Table 3-5. The table shows that for these workloads, the reduction in TLBMCPI due to the *APPROX-ONLINE* policy translates into a reduction in program execution time of as much as 48% relative to a hypothetical system with small fixed-size pages. [3]

Table 3-5 This table shows the estimated percentage improvement in end-to-end execution time relative to a hypothetical system with fixed-size 4 KB pages. Negative entries reflect a slowdown.

Benchmark	*FIXED 16 KB*	*FIXED 64 KB*	*OFFLINE*	*ASAP*	*ASAP-4-64*	*ONLINE*	*APPROX-ONLINE*
coral	10.0	22.9	33.8	0.3	21.9	-0.8	29.5
compress	23.7	34.6	33.9	32.0	33.7	24.8	33.1
nasa7-5	36.2	48.1	47.9	47.1	47.8	45.3	47.8
nasa7-4	28.7	37.6	35.7	22.9	33.5	0.6	33.5
fpga	2.8	7.1	19.4	0.2	5.8	-3.0	17.4
cecil	2.5	3.9	4.5	0.5	0.4	-26.2	2.0
atom	7.3	7.4	7.2	6.9	2.4	-4.1	6.6
fft	2.7	8.0	8.8	4.0	7.0	-2.3	8.2
spice	1.1	1.3	1.3	0.6	1.3	1.2	1.3
gcc	1.1	1.1	0.6	-1.6	-0.4	-15.0	-0.8

3.6.4 Memory usage overhead

The results presented so far have focussed on the impact of TLB management on execution time. Without taking into account memory consumption, though, the picture is incomplete. For example, a trivial way to drive TLB overhead towards zero would be globally increase the page size towards infinity. Taking into account memory consumption, though, shows that increasing the page size is not without cost. As shown in Section 3.1, large fixed-size pages increase memory consumption because of internal fragmentation.

Figure 3-4 shows that the promotion policies can obtain improvements in TLBMCPI *without* the increase in memory usage associated with large fixed-size pages, because the online policies delay promotion until warranted by TLB miss patterns. The figure shows

[3] These improvements in execution time are greater than those possible on a DEC Alpha, which has a base page size of 8 KB, not 4 KB.

that the online policies never increase memory usage by more than 4%, and only in one case by more than 2%. In contrast, *ASAP 4-64* increases memory usage by 5% or more for four of the benchmarks, and as much as 12% for one.

The effectiveness of *APPROX-ONLINE* in limiting memory consumption is due to its ability to dynamically create a wide range of page sizes. Table 3-6 quantifies this effect, showing the distribution of final page sizes created with *APPROX-ONLINE*.

3.6.5 Simulation results: summary

Section 3.1 illustrated how large fixed-size pages obtain improved TLBMCPI at the expense of increased memory consumption. Figure 3-5 revisits the tradeoff between TLBMCPI and memory usage in the context of policies that use superpages. As before, the ideal policy would have both zero TLBMCPI and zero memory usage overhead, and so would be plotted at the origin of these graphs.

The graphs show that *APPROX-ONLINE*, by paying a small overhead on each TLB miss to analyze the cause of TLB misses, is able to selectively construct superpages that will improve future performance. The result is an overall improvement in performance nearly as great as that of the omniscient *OFFLINE* policy. Furthermore, *APPROX-ONLINE* outperforms all of the other promotion policies on every benchmark in terms of TLBMCPI, memory usage overhead, or both.

3.7 Implementation issues

Given the performance benefits of dynamic superpage construction revealed in the previous section, a practical future project would be an implementation of such a policy on a real system. In this section I outline a strategy for implementing an online superpage construction policy. While the broad strategy should be applicable to any system with appropriate hardware support for superpages, to make the discussion concrete I will focus on a possible implementation in the context of Digital Unix running on an Alpha processor. I first describe the differences between this environment and the system simulated in this chapter, and how these differences would affect the policies introduced here. I then describe software extensions to the Digital Unix TLB miss handler and virtual memory system to support dynamic superpage construction.

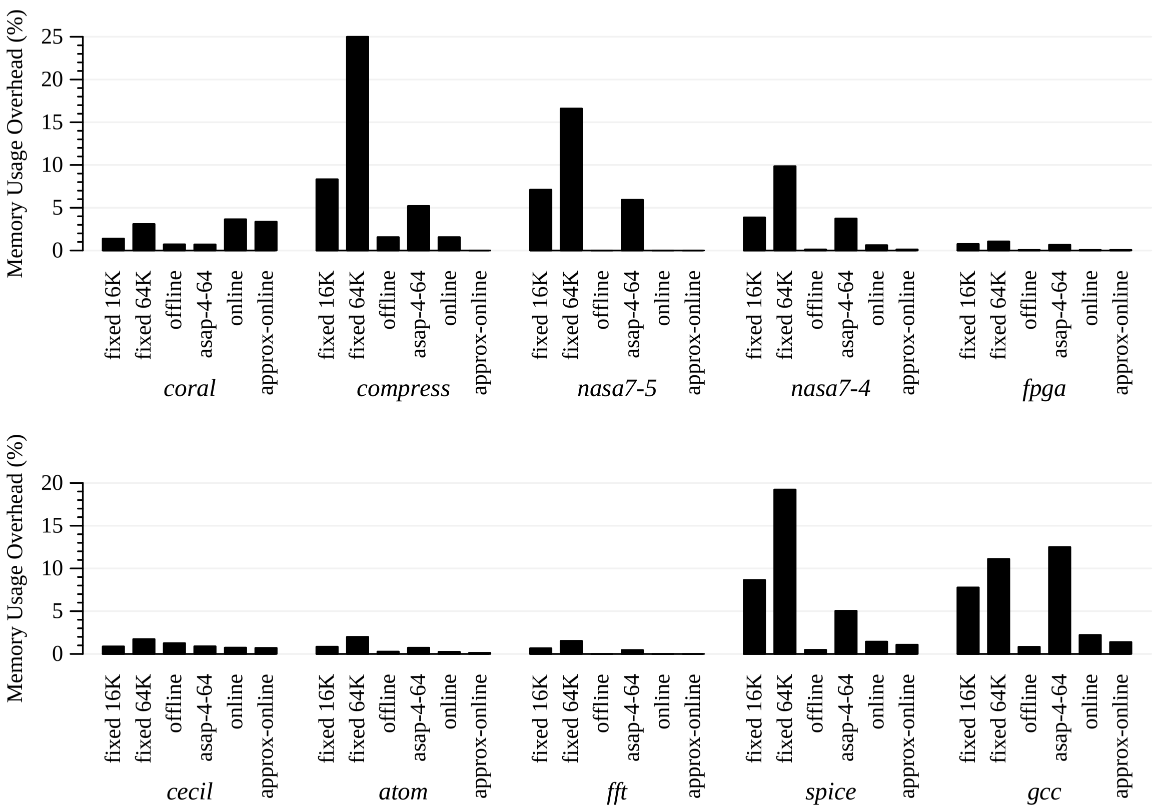
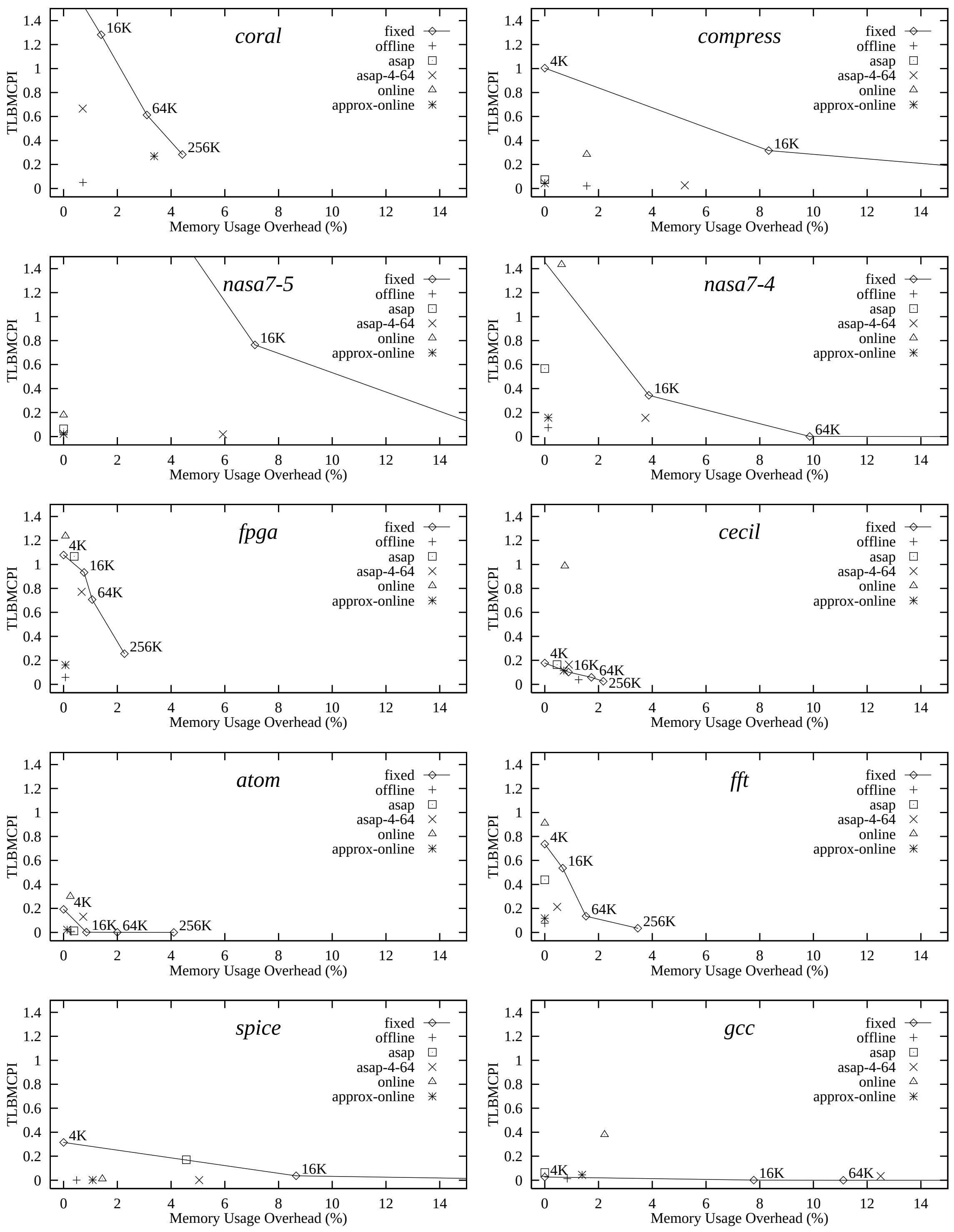


Figure 3-4 Memory usage overhead. *ASAP* uses exactly as much memory as *FIXED 4 KB*, and is not shown. *ASAP-4-64* promotes as soon as a 64 KB superpage is at least half full, so its memory usage can be greater than *ASAP*.

Table 3-6 The final page distributions for *APPROX-ONLINE*. This table shows the number of pages of a given size at the end of the run of each program.

Benchmark	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB	1 MB	2 MB	4 MB	8 MB	Total 4 KB Pages
coral	105	9	11	4	8	3	4	1	11	8	1	0	8,743
compress	22	1	6	10	2	1	0	0	0	0	0	0	192
nasa7-5	126	0	0	0	1	0	1	1	1	0	0	0	590
nasa7-4	406	0	1	13	10	4	0	0	0	0	0	0	802
fpga	378	2	0	5	1	1	3	12	13	2	4	0	10,646
cecil	44,526	154	61	45	40	19	14	9	4	8	8	5	72,286
atom	8,134	17	11	6	3	2	0	0	0	0	0	0	8,372
fft	97	0	0	0	0	2	1	1	1	1	1	3	8,289
spice	117	2	4	4	8	9	4	0	0	0	0	0	841
gcc	109	8	18	15	3	0	0	0	0	0	0	0	365

Figure 3-5 (Facing page.) Tradeoff between memory usage and TLBMCPI. The dynamic policies (*ONLINE*, *APPROX-ONLINE*, and *OFFLINE*) can achieve the benefits of small pages, in terms of memory usage, and those of large pages, in terms of TLBMCPI. The origin represents the ideal point in the tradeoff between memory usage and TLBMCPI. The graphs show that the dynamic policies come close to that ideal point. For comparison the figures include the performance of fixed-size pages as large as 256 KB. In some cases the *FIXED* policies have such high TLBMCPI or memory usage that they do not fit on these graphs, and in one case (*coral*) *ASAP* has TLBMCPI that is too high to fit on the graph.


coral
compress
nasa7-5
nasa7-4
fpga
cecil
atom
fft
spice
gcc
fixed
offline
asap
asap-4-64
online
approx-online
TLBMCPI
Memory Usage Overhead (%)
4K
16K
64K
256K

3.7.1 *Reality vs. simulation*

The experiments described in this chapter assume a TLB that supports all superpage sizes in power of 2 multiples of the base page size, ranging from 4 KB up to 8 MB. The Alpha, however, supports only power of 8 multiples of the base page size, ranging from 8 KB up to 4 MB. The smaller maximum page size on the Alpha could result in slightly lower TLB coverage for the few applications that benefit from 8 MB pages (of the benchmarks studied here, only *cecil* and *fft*). The smaller number of page sizes will result in later promotions, since the promotion threshold is proportional to the superpage size. On the other hand, the space and time required to maintain the policy's counters will be reduced, since there will be fewer nodes in the tree of counters.

The simulations also assume that the precise contents of the TLB are known on each TLB miss. On the Alpha, unfortunately, this information is not readily available. The Alpha TLB replacement policy depends on the order of both TLB hits and misses. However, only TLB misses are visible to the operating system, so it is impossible to emulate the hardware replacement policy accurately in software. Implementing the policies would require a technique for approximating the contents of the TLB. This would have the effect of making the counters maintained by the policies less precise, possibly resulting in less optimal promotions.

3.7.2 *Digital Unix implementation strategy*

Implementing superpage support in Digital Unix would require changes to the TLB miss handler, and to the virtual memory system's page table and free list data structures.

TLB miss handler modifications

The TLB miss handler must be modified to vector control to the operating system on each TLB miss. The operating system can then update the counters that drive the promotion policy described in this chapter, possibly resulting in the construction of a superpage.

As mentioned earlier, the DEC Alpha does not easily allow the operating system to identify which TLB entry is replaced on a TLB miss, although this would be a simple extension to the TLB hardware. While it is possible to scan the TLB, this is a time consuming operation, requiring 3 machine instructions per TLB entry – *e.g.*, 96 instructions to scan the 32-entry DTLB on the Alpha 21064. Rather than rescanning the TLB on each miss, it may be more effective to approximate the TLB contents. For example, the operating system could simulate a FIFO replacement policy. This approximation could be refreshed by periodically flushing the TLB and allowing it to refill. Whether the discrepancy between the prediction of the simulated FIFO replacement policy and the actual NLU (not-last-used) replacement policy of the Alpha TLB would result in an inefficient promotion policy is an open question, but is one that could be answered via simulation.

Virtual memory data structures

The operating system page tables must be extended in order to represent variable page sizes. A simple solution, suggested by Talluri *et al.* [Talluri et al. 95], is to simply extend the page table entry to include a page size field, and to replicate page table entries for each of the component pages of a superpage. In addition, existing virtual memory operations must be extended to preserve the illusion of fixed-size pages to higher levels of software. Whenever the protection or validity of a virtual page component to a superpage changes, the superpage must be broken into smaller superpages. A disadvantage is that the cost of protection changes will increase. Another disadvantage is that from the point of view of an application the cost of such virtual memory operations will be unpredictable.

In addition, the free lists must be modified to efficiently represent multiple page sizes. A variant on the "buddy system" for free storage management [Knuth 73] would be appropriate, with a set of free lists for each page size, and within each set, a free list per cache color. (While page color is not considered in this chapter, it will become relevant in subsequent chapters that consider the interaction of virtual memory and physically indexed caches.)

Promotion

A promotion is triggered by a TLB miss that in turn causes the counter value for some potential superpage to cross the threshold value. If all of the component pages are already resident or are zero-fill pages that can be easily made resident, the superpage can be constructed. A page is taken from the free list for the appropriate superpage size; if the free list is empty, a larger free page is split into smaller pages. If free physical memory is so fragmented that there is no free page large enough to satisfy the request, a compaction step will be necessary. (While standard garbage collectors also compact free storage, the size and placement constraints of superpages may change the tradeoffs involved in deciding when and how to compact physical memory.)

Once a free page is found, the component pages are copied to the new page and then returned to the pool of free pages, where they may be coalesced with their "buddies" to form larger free pages. The page table entries for each component page are updated to reflect the new physical pages backing the virtual pages, as well as the size of the superpage. Any stale translations for the component pages must be evicted from the TLB, and finally the new translation can be inserted into the TLB.

Demotion

A demotion occurs because of a change to the attributes of a base page contained within a superpage. The affected superpage is divided into the largest possible smaller superpages, and the page size field for each component page is updated appropriately. No copying is necessary since the physical pages backing the virtual pages have not changed. However, the now invalid translation for the former superpage must be evicted from the TLB. An optimization that would allow rapidly reconstructing superpages would be to maintain a per-base page bit indicating whether a page is "interfering" with the existence of a superpage. This bit would be set for the page whose status caused the superpage to be demoted. If the status of the page changed again, the system could attempt to immediately reconstruct the superpage at low cost.

3.8 Open questions

While this chapter demonstrates the effectiveness of dynamic superpage construction for a broad range of workloads on a simulated system, there remain a number of open questions. These questions can be divided into three categories:

1. those that ask whether systems with different (larger, faster) TLB configurations would also benefit from this technique;

2. those that ask whether further policy refinements are necessary in order to benefit additional applications and to "do no harm" to applications that do not benefit from increased TLB coverage; and

3. those that ask whether the ideas that underly online superpage construction could be extended to provide additional benefits, such as improving I/O performance.

3.8.1 Alternative TLB Hardware

Since the original measurements for this chapter were made, TLB sizes have been increasing in response to growing application memory requirements [Dig 97]. Some systems include set-associative TLBs, allowing larger TLBs to be built with the same access time as a smaller fully-associative TLB [Uhlig et al. 94]. Others have proposed building a software cache of the TLB to reduce TLB miss times [Bala et al. 94]. The impact of superpage construction policies will be diminished as the number and cost of TLB misses decrease, so a reasonable question would be whether the new larger TLBs are sufficient to map most applications' working sets. It would be straightforward to simulate these TLBs in order to answer this question for a given set of applications. However, the policies proposed here allow TLB coverage to be increased by orders of magnitude without a corresponding increase in hardware resources. Doubling the TLB size will not improve TLB performance if application working set sizes also double.

The policies in this chapter represent an alternate point in the design space, one that allows TLB coverage to scale with application memory requirements, while requiring only modest TLB sizes. Smaller TLBs have the advantages of potentially lowering access time and occupying less chip real estate. While it is possible that the reduced access time to a direct-mapped or associative TLB would allow a faster processor cycle time, cycle time alone does not predict performance. As the importance of memory latency increases, it becomes increasingly attractive to trade slower cycle time for improved TLB hit rates.

3.8.2 Policy refinements

While the relatively straightforward *APPROX-ONLINE* policy presented in this chapter provides significant performance improvements for a variety of applications, further refinements might be necessary to provide these benefits across an even broader range of programs without harming the performance of others. For example, in Chapter 6 I show that *APPROX-ONLINE* increases TLBMCPI for some applications unless it is augmented with a throttling mechanism. As another example, although a fixed promotion threshold worked well in the experiments in this chapter, in practice it might be necessary to tune the promotion threshold in response to the measured TLB miss rates and copying rates of each application.

Another refinement would be to allow the promotion threshold for superpages to vary according to the cost of constructing the superpage. For example, if many of the component pages of a superpage are already in place, the copying cost will be much lower, justifying an earlier promotion. At the other extreme, if memory is fragmented, it may be necessary to vacate a region of memory in order to construct the superpage, increasing the copy cost. (Promotion costs will also increase if the target application is using multiple processors, since multiple TLBs must be synchronized to reflect the new mappings.) The first issue to address is whether it would be beneficial to take into account variable promotion costs, or whether the approximation that construction cost is simply proportional to superpage size is close enough. It might be sufficient to handle a few special cases, such as the case of a temporary protection change to a component of a superpage. Generally supporting variable promotion thresholds would require extending the counter update algorithms to allow efficiently representing and testing the per-superpage thresholds.

3.8.3 Policy Extensions

Combining superpages and page reservation

The policies introduced in this chapter differ from those proposed by Talluri *et al.* in that they do not rely on page reservation. The advantage of page reservation is that for applications that can be covered by a TLB with 64 KB pages, there will be virtually no TLB overhead. The primary disadvantage is that applications with very large working sets will still suffer a large number of TLB misses. A hybrid policy could initially allocate pages in 64 KB chunks wherever possible, promoting and demoting superpages as needed. This approach could offer minimal TLB management costs for applications with relatively small working sets, while still allowing TLB coverage to scale up for applications with very large working sets. Unfortunately, compaction of physical memory would become more frequent. In addition, it is not immediately clear how the system should choose which 64 KB pages to demote when physical memory becomes scarce.

Subpages

Superpages trade away fast fine-grained virtual memory protection operations in return for improved TLB miss rates. For some applications efficient virtual memory operations are more important than TLB efficiency. For these applications "subpages", pages smaller than the base page size, could be beneficial. For example, Jamrozik *et al.*'s global memory system [Jamrozik et al. 96] simulates subpage protection in order to support bringing partial pages in from secondary storage. A disadvantage of subpages is that a naive implementation would require more bits in each TLB and page table entry, and larger page tables.

(For example, halving the minimum page size would double the page table size.) An area for research, then, is page table structures that efficiently represent subpages, assuming that subpages are only used infrequently. The solution used by Jamrozik *et al.* is to maintain a separate table containing the subpage-specific bits. In their case, the only bit of interest is the valid bit, so the space cost is very small: one bit per subpage.

Improved I/O performance

The TLB monitoring system described in this chapter effectively collects information about the spatial locality of virtual pages. While superpages offer one way of taking advantage of spatial locality, there are potentially other opportunities to use this information. For example, this information could be used to drive prefetching policies when a program is paging. If a page is brought in from disk that was formerly part of a superpage, it is likely that other components of the superpage will be referenced soon, and should be prefetched from disk. Also, information about superpage mappings from past runs of a program could be used to eagerly allocate superpages for future runs of the same program. This would require additional infrastructure to maintain a database of profile information, but such infrastructure is useful in other contexts [Grove et al. 95, Hookway & Herdeg 97, Zhang et al. 97]. Alternatively, it might be effective to simply tune the initial page size on a per-program or per-segment (stack, heap, static data) basis, rather than attempting to recreate a full set of superpages at the beginning of a program's run.

Policy extensions: summary

With the exception of the prefetching and subpage proposals, these extensions are primarily targeted at eliminating the copy costs associated with superpage construction. Before pursuing them further, it would be necessary to show that copying costs are in fact a significant source of overhead. For the workloads studied in this chapter, though, copying costs did not make a large contribution to overall TLBMCPI under the *APPROX-ONLINE* policy – typically less than 0.1 MCPI. This could change with workloads that make more fragmented use of the virtual address space and whose use of the address space varies dynamically over time.

3.9 Conclusions

This chapter provides the first demonstration of my thesis: that the operating system can effectively manage hardware memory resources in order to improve application performance. By making the TLB miss handler slower but smarter, the operating system can dynamically construct a set of superpages that increase TLB coverage and dramatically reduce TLB miss rates. The savings from eliminating future TLB misses not only pays for the costs of the longer TLB miss path and of superpage construction, but improves the end-to-end performance of applications with large working sets. While increasing the page size system-wide would also reduce TLB miss rates, this would come at the price of a significant increase in memory consumption due to internal fragmentation. The *APPROX-ONLINE* policy, which performs a runtime cost-benefit analysis to identify beneficial superpages, obtains the low TLB miss rates of systems with large fixed page sizes with the low memory requirements of systems with small pages. *APPROX-ONLINE* also outperforms policies that promote superpages obliviously in terms of execution time, memory consumption, or both. This policy achieves these benefits without requiring exotic hardware, and without requiring modifications to software beyond the lowest level of the operating system virtual memory system. By collecting and analyzing the information needed to adjust page sizes dynamically, a superpage construction policy can increase the coverage of a small TLB to meet the demands of memory-intensive applications.

Chapter 4

Using Virtual Memory to Improve Cache Performance

4.1 Introduction

In this chapter I describe the design and implementation of an operating system policy that manages cache resources in order to transparently improve the execution time of applications running on standard hardware. I show that the operating system can use mechanisms available on existing systems, namely a cache miss counter and a software-filled TLB, to detect cache misses. I design a virtual memory policy that uses this information to dynamically change the mapping from virtual addresses to cache offsets, eliminating future cache misses. I describe how I implemented this policy in Digital Unix on a DEC Alpha workstation. The measurements of this implementation show that the policy can in fact eliminate enough cache misses to improve end-to-end application execution time by as much as 21% compared to a static page coloring mapping policy, without requiring any changes to user-level programs or to the underlying hardware.

This operating system policy improves application execution times by dynamically managing cache resources in order to eliminate cache *conflict* misses. In a direct-mapped cache, conflict misses occur when two or more active addresses index to the same cache line. More generally, conflict misses occur when more active addresses index to a cache set than there are lines in the set. If the cache is large enough to hold the application's working set, conflict misses will never occur as long as all the active addresses in the program index to distinct cache lines. Conflict misses are a problem because they cause programs to run slowly and with unpredictable execution time [Hill 87, Chen & Bershad 93, Hosking & Moss 93, Wahbe et al. 93, Chaiken & Agarwal 94]. Caches also suffer from *compulsory* and *capacity* misses. Compulsory misses occur on the first reference to an address, and do not yet have a large impact on overall performance in current systems. Capacity misses occur when an application's working set is too large to fit in the cache. Eliminating capacity misses requires reducing the working set size of the application, which requires reprogramming or recompilation, and may simply be impossible.

While in some cases conflict misses can be eliminated statically by carefully analyzing an application's memory reference behavior and rearranging data structures and loop nests in order to minimize conflicts [Lam et al. 91, LaMarca 96, Chilimbi et al. 98], this approach only benefits applications with relatively predictable reference patterns. An effective way to eliminate cache conflict misses for arbitrary applications is to use a set-associative cache, which allows two or more addresses that index to the same set to reside in the cache simultaneously. However, direct-mapped caches are cheaper, simpler, and faster than associative caches [Wood 86]. In particular, constructing an associative off-chip cache will consume one or all of pin-bandwidth, time, or dollars [Hill 88, Przybylski et al. 88]. If the tag comparison is performed on-chip, then either additional wires must run between the cache and the chip, (consuming valuable chip pin-bandwidth), or the tag comparison must be performed serially (slowing down access time on some cache hits). Performing the tag comparison off-chip requires building an expensive custom cache controller.

In this chapter I show how to provide associativity in software, eliminating conflict misses in physically indexed caches without incurring any of the costs of an associative cache. The techniques that I propose take advantage of the flexibility in the mapping from virtual to physical pages to introduce associativity in a direct-mapped physically indexed cache. Recall from Chapter 2 that in a physically indexed cache the mapping from virtual address to cache line is determined in part by the mapping from virtual to physical pages. Ideally the operating system will arrange that all active virtual pages map to physical pages that in turn map to distinct cache pages. That is, the virtual pages should map to physical pages of different *colors*. If this ideal is met, then no cache conflict misses will occur. Since the set of active pages in an application cannot generally be predicted statically, in practice finding and maintaining a virtual-to-physical mapping that results in good cache performance requires a dynamic mapping policy.

The basic operation of a generic dynamic mapping policy is straightforward, and is most easily seen by considering a simple example. Suppose an application repeatedly references just two virtual pages, and that the operating system initially maps these pages to physical pages that conflict in the cache. Unless this poor initial mapping is changed, the application will incur a large number of cache conflict misses and run slowly. But if the operating system can detect the problem it can eliminate the cache misses by copying one of the virtual pages to a physical page of a different color, and then updating the virtual to physical mappings appropriately—that is, the system can *recolor* one of the pages. Although recoloring involves a physical memory copy and thus costs thousands of cycles, as the program runs the improvement in cache miss rates will lead to an overall reduction in execution time for this application.

While correcting conflicts between virtual pages is straightforward, detecting such conflicts is not. In Chapter 3 I considered how to dynamically detect and correct a different memory resource performance problem, namely the problem of high TLB miss rates. Detecting the a TLB performance problem is easy, because in a system with a software-filled TLB the operating system is notified on every TLB miss. Cache misses, in contrast, are handled transparently in hardware. Thus dynamically eliminating cache misses requires a method for *detecting* those misses at runtime.

In this chapter I develop techniques that infer the location of cache conflict misses using feedback mechanisms available on modern systems. An ideal source of information about cache miss behavior would be a mechanism that notified software of every memory reference with no overhead. In this chapter I consider techniques that approximate this ideal information source, but that provide incomplete information and do incur some runtime overhead.

The basic insight behind the techniques I introduce is that even though hardware does not explicitly reveal conflict misses to the operating system, some memory references *are* visible to software. In particular, in systems with software-filled TLBs, memory references that result in TLB misses are visible to the operating system. Furthermore, in systems with a cache miss performance counter, the operating system can collect additional data about the behavior of the memory system, although not about individual cache misses. Software-filled TLBs and cache miss performance counters are common features on modern systems [Dig 92, Heinrich 96].

Designing a policy that optimizes cache performance using feedback available on existing systems presents both a challenge and an opportunity. The challenge lies in determining how to use incomplete information to analyze and improve cache behavior. Once this challenge is met, the opportunity is to implement the resulting policy on a live system and improve the execution time of real applications.

In the first part of this chapter I explore the design space of virtual mapping policies that use the contents of the TLB to determine the working set of the application, and use the system cycle counter or cache miss counter to regulate their level of activity. Because these policies incur additional TLB overhead to examine the state of the system, they must balance the cost of monitoring against the value of the information collected. Through simulation I show that one of these policies, which I call *Snapshot-Miss*, can detect and eliminate cache conflicts with low overhead, improving overall application performance. Snapshot-Miss uses a cache miss counter to determine how frequently cache misses are occurring, and uses the TLB to determine which pages could be responsible for the misses.

Having designed an effective operating system policy for transparently managing cache resources, in the second part of this chapter I describe the implementation and performance of the Snapshot-Miss policy on a DEC Alpha workstation running Digital Unix. The implementation is straightforward, requiring only modest extensions to the operating system. The measurements of the implementation show that a dynamic mapping policy can in fact reduce cache conflict miss rates, and that this reduction in turn improves the execution time of applications running on a real system.

Together, the results presented in this chapter demonstrate that the operating system is an effective vehicle for dynamically managing cache resources in order to improve overall application performance. By observing the state of the memory system at runtime, the operating system can infer the presence of cache conflicts and update virtual-to-physical mappings in order to remove those conflicts. Despite the incomplete information available about the cause of cache misses on current hardware, a well-designed policy can still use this information to eliminate enough cache misses to more than compensate for monitoring and recoloring overheads and hence improve the end-to-end application execution time. The results also indicate that the policies designed in this chapter do not fully take advantage of the opportunity to improve performance by eliminating cache conflicts. This observation motivates the work presented in the next chapter, in which I consider how to design a hardware feedback mechanism that can provide more accurate information about cache behavior with low overhead, and how to design an operating system cache management policy driven by this information.

The rest of this chapter

In Section 4.2 I explore the design space of policies that use different sources of feedback and different levels of intrusiveness in attempting to detect and resolve cache conflict misses. I explain the evolution of the Snapshot-Miss policy by considering a series of progressively more sophisticated policies, each correcting a drawback of the previous one. I then use trace-based simulation of ten applications to predict the effect of each policy on program performance. In Section 4.3 I describe the tracing system and the experimental methodology. In Section 4.4 I present the simulation results, quantifying the performance of the policies presented in Section 4.2. Motivated by these measurements, I implemented Snapshot-Miss on a DEC Alpha workstation running Digital Unix. Section 4.5 describes this implementation and its performance. Finally, I conclude in Section 4.6.

4.2 Policy design

My goal in this chapter is to design and evaluate an effective operating system policy for dynamically improving cache performance and hence overall application execution time. My design goals for this policy are as follows.

1. The policy should eliminate cache conflict misses across a range of applications.

2. The policy should rely on mechanisms available on existing hardware, enabling the benefits of the policy to be realized in an implementation.

3. The policy should not have overhead that overwhelms its benefits—simply put, the policy should result in a net improvement in application execution time.

I reach two conclusions in this section. First, a policy that meets these goals must balance the cost and accuracy of information about the cause of cache conflict misses. Second, a policy that improves overall application performance using mechanisms available on existing hardware requires information about the frequency of cache misses as well as their location.

The outcome of this section is a policy that reflects these two conclusions and meets the design goals. This policy, Snapshot-Miss, uses a cache miss counter to determine when cache misses are a problem, and uses TLB misses to determine where in the virtual address space conflicts are a problem. Snapshot-Miss meets the first design goal by monitoring memory references as revealed by TLB misses to collect a "snapshot" of an application's active virtual pages. Virtual pages that cause conflict misses will appear in the snapshot, and can be recolored. The policy meets the second design goal because cache miss counters and software-filled TLBs are common features on modern systems. The policy meets the third design goal by using the cache miss counter to calibrate its level of activity. Detecting cache conflicts with the TLB and correcting cache conflicts by recoloring are both high-overhead operations. By monitoring TLB misses and recolor pages only when the cache miss counter indicates that a performance problem exists, Snapshot-Miss incurs overhead that is bounded and is proportional to the cache miss rate.

I developed the Snapshot-Miss policy by beginning with a simple policy that eliminates cache conflicts but has high overhead, and then refining this policy in order to reduce policy overhead while still eliminating cache misses. I continued refining the policy until I arrived at the Snapshot-Miss policy. In this section I will describe the policies that serve as intermediate steps in the design process—not because they have good performance, but because they illustrate the problems that an operating system policy must solve if it is to improve overall cache performance using standard hardware.

In the rest of this section I first briefly describe the mechanisms that can provide feedback on memory system behavior on modern systems, and then describe the series of policies that led to the Snapshot-Miss policy. For convenience I divide these policies into two groups, *Active* (policies that continuously monitor the TLB) and *Periodic* (policies that periodically examine the TLB).

4.2.1 Sources of feedback

The cache management policies I develop in this section collect information about memory system behavior available on many current systems. In particular, these policies rely on a software-filled TLB and on hardware performance counters. While most memory references are handled transparently in hardware, the TLB reveals information about specific references to the operating system, and performance counters reveal information about the aggregate behavior of the system over time. I briefly describe here the interfaces to the TLB and performance counters assumed by the policies. These interfaces are consistent with those found on modern CPUs such as the DEC Alpha 21164 and MIPS R10000.

A software-filled TLB

The TLB contains translations from virtual pages to physical pages. The operating system can read the contents of the TLB and insert new entries into the TLB. (Whether the operating system can control the TLB replacement policy is not important for the results in this chapter.) When the CPU refers to a virtual page that is not present in the TLB, the operating system is notified of the TLB miss so that it can insert the translation for the page into the TLB.

Since the operating system controls the contents of the TLB, it controls which memory references to mapped memory result in TLB misses and are therefore visible to the operating system. As an extreme example, if the TLB is restricted to map just one page at a time, the operating system can observe every reference to distinct pages, although it will not see consecutive references to the same page. This allows the operating system to collect detailed information on the memory reference stream at the cost of high TLB miss rates. While this specific example results in high overhead, it illustrates one of the ideas behind the cache management policies developed in this chapter: the operating system can manipulate the state of the TLB to collect information about an application's set of active virtual pages.

Hardware performance counters

Modern systems provide hardware performance counters that allow software to count a variety of events. In this chapter I consider only cycle counters and cache miss counters. These counters can be configured to deliver an interrupt when they reach a software-controlled value. The operating system can use these counters, for example, to determine when cache miss rates are high. The policies I describe here use the counters to tie their level of activity to the cache miss rate or to program execution time, thus bounding policy overhead.

4.2.2 Active policies

The first set of policies that I introduce actively monitor the TLB to maintain accurate information about the set of pages in use as an application runs. These active policies use this information to eagerly recolor any active pages that conflict, maintaining the invariant that all pages in the TLB are of different colors. To see how eagerly recoloring pages could improve overall performance, consider a hypothetical application whose steady state working set size fits in *n* virtual pages, running on a system with a physically indexed cache containing at least *n* cache pages. For this application, regardless of the initial virtual-to-physical mapping, a policy that recolored conflicting pages in order to satisfy the invariant would eventually reach a state in which the program ran with no cache misses and with no additional page copying overhead. The only long-term overhead would be the cost of testing the invariant on each TLB miss.

I designed three active policies that maintain this invariant that all of the pages in the TLB have different colors. The first, Active-Naive, checks whether the invariant would be violated on each TLB miss, and if so, recolors one of the offending pages. This policy can reduce conflicts by distributing memory references over the cache, but it will recolor excessively. The policy may unnecessarily copy data that are on pages of the same color but not in long-term competition for cache space: that is, pages that will not incur enough future cache misses to justify the overhead of recoloring.

The next policy, *Active-Delay*, addresses Active-Naive's flaw of recoloring unnecessarily. Active-Delay still prevents pages of the same color from appearing in the TLB. But instead of immediately performing an expensive recolor operation, this policy instead invalidates the TLB entry for one of the conflicting pages, and increments a counter corresponding to the color of the page. Only when the counter reaches a threshold value is one of the pages recolored. Effectively this policy provokes TLB misses in order to count how many times two pages appear to be in conflict. Although this strategy has the advantage of eliminating some unnecessary copies, it comes at the price of increasing the number of TLB misses and reducing the responsiveness of the policy to cache conflicts.

The last of the active policies, *Active-Throttle*, addresses the high TLB miss rates that occur when the delay parameter for Active-Delay is set high enough to eliminate wasteful page copying. Active-Throttle is like Active-Delay, except that it bounds the number of copies in a given time interval. This reduces TLB overhead, because the monitoring mechanism can be disabled once the copying limit has been reached. Once the timer expires, TLB monitoring resumes. The Active-Throttle policy increases implementation complexity slightly and reduces the accuracy of the information collected about the memory reference stream in exchange for reduced policy overhead. Although capping the recoloring rate can reduce policy overhead, when it is active this policy still incurs many TLB misses in order to identify conflicting pages.

Active policies: summary

The top part of Table 4-1 summarizes the active polices and their advantages and disadvantages. These policies all collect very accurate information about the cause of conflict misses, and then eagerly recolor pages in order to map the application's current working set to distinct cache pages. While this approach eliminates cache conflict misses, the active policies can incur one or both of high TLB miss rates and high recoloring rates. The results in Section 4.4 show that these overheads overwhelm the reduction in cache misses, leading to the conclusion that eagerly copying pages is not an effective way of improving application performance.

4.2.3 Periodic policies

The active policies incur high overhead by continually monitoring the TLB. I show in Section 4.4 that this overhead causes these policies to harm rather than improve application performance. The periodic policies are designed to have lower overhead, although in ex-

Table 4-1 Dynamic mapping policies. This table summarizes the policies introduced in this chapter for detecting and eliminating cache conflict misses using standard hardware.

Policy	Summary	Advantages	Disadvantages
Active Policies			
Active-Naive	Recolor on every TLB miss to conflicting pages.	Detects all cache conflicts in mapped memory.	Excessive recoloring.
Active-Delay	Recolor after several TLB misses to conflicting pages.	Detects most persistent cache conflicts.	Excessive TLB overhead.
Active-Throttle	Recolor after several TLB misses to conflicting pages, with a cap on the recoloring rate.	Bounded copy overhead.	Many conflicts may not be detected.
Periodic Policies			
Periodic-Random	Periodically recolor a randomly selected page that appears in the TLB.	Repairs poor initial mappings.	Destroys good initial mappings.
Periodic-Color	Periodically recolor one of a set of pages that conflict and appear in the TLB.	Repairs poor initial mappings.	May recolor inactive pages.
Snapshot	Periodically flush the TLB. Recolor when two conflicting pages reappear in the TLB.	Recolors only active pages.	Recolors in response to short-lived conflicts.
Snapshot-Delay	Recolor when two pages conflict in two consecutive snapshots.	Recolors in response to persistent conflicts.	Difficult to determine good period and delay values.
Snapshot-Miss	Like Snapshot-Delay, but snapshot interval and length measured in cache misses rather than cycles.	Adapts to cache miss rate.	Requires cache miss counter.

change they must use slightly less accurate information about application working sets than the active policies. In particular, the periodic policies bound their overhead by examining the state of the TLB at a rate that is tied to either the cycle count or the cache miss count. The periodic policies have the potential to work well because in practice it may *not* be necessary to respond to conflict misses as soon as they occur. In particular, the conflict misses that have the greatest impact on performance are those that persist for a long time. The performance benefit of repairing such conflicts will be significant even if recoloring is delayed somewhat from the onset of the problem.

The periodic policies bound the overhead of detecting cache misses by collecting information less frequently than the active policies, and also bound the overhead of recoloring. The policies that I introduce here use progressively more sophisticated approaches to identify appropriate pages to recolor, so that the benefit of a reduced cache miss rate outweighs the overhead of monitoring TLB misses and copying pages.

The simplest periodic policy is *Periodic-Random*, which periodically recolors one randomly chosen page that appears in the TLB. Since it may recolor pages that do not conflict, Periodic-Random could introduce as many conflicts as it removes. The *Periodic-Color* policy is more selective, scanning the TLB and recoloring one page from a set of pages that have the same color.

The Periodic-Random and Periodic-Color policies assume that the contents of the TLB accurately reflect the current working set. In practice, inactive pages can appear in the TLB, causing unnecessary recolors. The *Snapshot* policy is similar to Periodic-Color, but filters out these inactive pages. The policy periodically takes a "snapshot" of the working set by flushing the TLB, and observing which pages incur TLB misses over a short interval. By definition, the working set consists of these pages. The Snapshot policy recolors one page from the set of pages that appear to conflict with other pages in the snapshot.

The Snapshot policy assumes that conflicting pages will continue to conflict long enough to justify recoloring. If conflicts are short-lived, the policy will recolor excessively. To confirm that a conflict is long-lived, the *Snapshot-Delay* policy takes a second "confirmation" snapshot after a specified delay time, and only recolors a page when it is found to conflict in both the original and confirmation snapshots.

The periodic policies described so far invoke the cache conflict detection mechanisms at a fixed rate, whether or not cache misses are occurring. This may lead to excessive recoloring when the cache miss rate is naturally low, or to delayed conflict resolution when the cache miss rate is high. The *Snapshot-Miss* policy addresses these problems by using a cache miss counter to regulate its level of activity. Snapshot-Miss is like Snapshot-Delay, except that the policy parameters are measured in cache misses rather than in machine cycles. The system delivers an interrupt when the cache miss counter reaches a given threshold, at which point the policy takes a snapshot of the working set as described above. In effect, this policy uses the cache miss counter to determine *when* excessive cache misses are occurring, and only then monitors the TLB to determine *where* the misses are occurring.

4.2.4 Policy design: summary

Table 4-1 reviews the policies presented in this section, and summarizes some of their advantages and disadvantages. All of the policies attempt to infer the location of cache conflict misses based on information about the application memory reference pattern derived from the TLB. This is challenging because this information may mislead the policy as to the actual cache behavior. For example, any of the policies may conclude that two pages conflict even though the pages in fact are not causing cache misses. For example, suppose that physical pages P1 and P2 have the same color, and that both are in use. Even if no reference to P1 conflicts with references to P2 (say only the top half of P1 and only the bottom half of P2 are active), the policies may conclude that P1 and P2 conflict. Snapshot-Miss is the least likely to exhibit this behavior, because it is active only during periods of high cache miss activity. Nevertheless, all the software policies may find such "false positives," because they do not use information about real cache misses, only potential ones. Another problem with these policies is that they cannot detect or resolve conflicts between mapped and unmapped pages, because unmapped pages are not present in the TLB.

In light of these problems, a policy that manages the cache in order to reduce overall execution time must balance the accuracy and cost of information about application behavior. The results in Section 4.4 will show that of the policies considered here, Snapshot-Miss comes closest to achieving this balance. This policy

1. regulates its level of activity according to the cache miss rate;

2. defers recoloring until a conflict between pages has persisted; and thereby

3. recolors pages when and where it is likely that the benefit will not be overwhelmed by the policy overhead.

In Section 4.4 I present measurements that quantify the advantages and disadvantages of the policies introduced in this section.

4.3 Experimental methodology

I evaluate the effectiveness of the dynamic mapping policies introduced in Section 4.2 using trace-driven simulation of ten Unix workloads. The traces include user and system references, and were collected using epoxie [Wall 92, Borg et al. 89, Chen 93]. In this section I describe the tracing system, the simulated memory system, the performance metrics used, and the benchmark programs.

4.3.1 The tracing system

I used address traces, which include user and system references, that were collected using epoxie [Wall 92, Borg et al. 89, Chen 93] running on a DECstation 5000/200 running Ultrix 4.2A. Epoxie rewrites MIPS [Kane & Heinrich 92] object files to insert instrumentation instructions before each basic block and memory reference. The instrumented binary emits enough information to reconstruct a complete trace of memory operations made during execution. During tracing, programs and the kernel fill a large trace buffer with accurately interleaved user and kernel references. When the buffer is full, the kernel suspends tracing and notifies a user-level process that consumes the trace.

Generating a faithful trace

The accuracy of trace-driven simulation that includes operating system references can be influenced by the effects of *memory* dilation and *time* dilation.

Memory dilation occurs because a traced program is on average twice as large as its untraced counterpart and may induce additional load on the virtual memory system (for example, TLB misses and page faults). The effects of memory dilation were avoided by collecting traces on a machine with a large physical memory (192 MB, of which 64 MB was devoted to the trace buffer) to avoid paging, and by simulating rather than tracing TLB behavior. Each benchmark was run on an otherwise unloaded machine.

Time dilation occurs because a traced program runs more slowly than its untraced counterpart, causing external events to appear to complete faster than they would in an untraced system. To avoid time dilation the traced system's clock was configured to interrupt at one fifteenth the standard rate (traced code runs about fifteen times slower than untraced code). In addition, tracing was disabled during the operating system's idle loop, which executes during all synchronous I/O operations. Eliminating idle loop references also improves the accuracy of memory system measurements because the idle loop exhibits good cache behavior (simulated cache miss rates would otherwise approach zero as idle activity increases).

4.3.2 The simulated memory system

I use the traces to drive a simulated memory system described in Table 4-2. The memory system parameters are based on a DEC 3000/500 Alpha workstation [Dutton et al. 92], which contains a 150 Mhz superscalar Alpha 21064 processor [Dig 92]. To accurately account for the overhead of the dynamic policies, I wrote and compiled code fragments that could implement TLB monitoring and page recoloring for each policy. I then modified the simulator so that the corresponding instruction and data references are injected into the application reference stream.

The simulations use an initial mapping policy of either page coloring or bin hopping. Page coloring assigns virtual pages to colors in a round robin fashion, assigning virtual page 0 to the first color, virtual page 1 to the second color, and so on. Bin hopping assigns virtual pages to colors cyclically in the order in which they are initially mapped. The simulations assume infinite memory (so that a free page of a given color is always available), and do not include paging effects.

Table 4-2 The parameters of the simulated memory system, based on a DEC 3000/500 Alpha workstation containing a 150 Mhz Alpha 21064 processor.

Page size	8 KB
Line size	32 bytes
First-level cache (L1)	8 KB instruction, 8 KB data virtually indexed direct-mapped
Policy	write-through, read-allocate
Write buffer	4 entries
Miss penalty	5 cycles
TLB	16-line ITLB, 32-line DTLB fully associative
Second-level cache (L2)	512 KB, unified, physically indexed, direct-mapped, unified
Policy	write-back, read/write-allocate
Miss penalty	25 cycles

I will also present measurements of cache management policies running on a DEC Alpha workstation running Version 2.0 of the Digital Unix operating system. These measurements sometimes differ from the measurements of the simulated system. There are a number of significant differences between the simulated and live systems that could account for these discrepancies:

- The traced and real programs ran on architectures with different instruction sets, and hence were compiled with different compilers.

- The traces were collected on a 32-bit MIPS system running the Ultrix operating system, while the implementation ran on a 64-bit Alpha system running Digital Unix.

- The simulated memory system makes a number of simplifying assumptions. For example, the first-level data cache on the Alpha allows execution to continue with one cache miss outstanding, as long as the result of the memory reference is not needed. The simulator does not emulate this behavior. The simulator also assumes a single-issue processor, and does not consider sources of instruction latency other than memory system delays. The Alpha processor in the real system is a dual-issue processor with variable-latency instructions.

The simulation results, then, do not precisely predict execution times on a real system. Instead they provide a quantitative but approximate guide to the advantages and disadvantages of different strategies for cache management.

4.3.3 Metrics

I use the measurements of the simulated memory system to compare the performance of the dynamic policies. As in Chapter 3, the primary metric for this comparison is *memory cycles per instruction* (MCPI), which is the total number of cycles spent servicing cache misses, write buffer stalls, and memory management (including TLB misses and cycles devoted to implementing the mapping policy), divided by the number of user and system instructions (excluding memory management instructions). Page mapping policies primarily affect the performance of the second-level cache—the first-level caches on the Alpha 21064 are each the same size as a page of physical memory, and so are largely independent of virtual-to-physical mappings. A more focused metric, then, is memory cycles per instruction due to second-level cache misses and memory management overhead (L2 MCPI). Since the policies considered in this section introduce additional TLB overhead to monitor the memory system, and additional overhead to recolor pages, I will sometimes separate L2 MCPI into three more detailed categories:

- TLBMCPI (MCPI due to TLB misses);

- recolor CPI (MCPI due to recoloring); and

- L2 miss CPI (MCPI due to references that miss in the second-level cache, excluding references induced by the cache management policy.)

4.3.4 Workloads

I simulated the performance of ten benchmarks, shown in Table 4-3, that are representative of a range of applications such as compilers, simulators, scientific codes, and graphical (X11) workloads. One program, *strawman*, is an artificial workload included to "show off" the recoloring policies in a worst-case scenario. The others are a mix of SPEC92 benchmarks and real workloads taken from the University of Washington computing environment. The traces include references from the applications and the operating system. The X11 workloads also include references from the X11 server.

This benchmark suite differs from the one I used in Chapter 3. This simply reflects the fact that I consider different performance problems in each chapter. In the previous chapter the benchmark suite was skewed towards programs with high TLB miss rates, while in this chapter it is skewed towards programs that suffer from high cache miss rates.

Baseline memory system behavior

Table 4-4 shows the baseline memory system behavior of these benchmarks using a static page coloring mapping policy (the policy that ships with Ultrix 4.2A). The table shows only the performance of the first-level cache and the write buffer between the first- and second-level caches. These numbers remain nearly constant for all mapping policies because the behavior of the program and the L1 cache are largely unaffected by the contents of the L2 cache.

4.4 Policy evaluation

In this section I use trace-driven simulation to show that dynamically remapping pages to eliminate cache misses can improve application performance. I show that of the policies I developed in Section 4.2, Snapshot-Miss offers the greatest improvement in performance across a range of benchmarks, reducing MCPI by as much as 0.27. The remaining policies do not perform as well as Snapshot-Miss, because even though they sometimes eliminate more cache misses, they also have higher policy overhead.

I divide the simulation results in this section into two parts. First, I measure L2 MCPI—the contribution of second-level cache misses and policy overhead to MCPI—for each of the

Table 4-3 Benchmark descriptions. Memory references, MCPI, and L2 miss CPI (MCPI due to second-level cache misses) were measured on the simulated memory system when using static page coloring as the mapping policy. Except as noted, the programs are written in C. With the exception of *gcc*, program text and input files are brought into the buffer cache before tracing. Traces of *gs* and *splot* include both client and X11 server activity. *Gcc*, *cecil*, *doduc*, *tomcatv*, and the *nasa* benchmarks are all from the SPEC92 suite. *Splot* and *gs* are used in the local computing environment. *Strawman* is an artificial benchmark designed to induce cache conflict misses. All of the traces include operating system references.

Benchmark	Description	Memory References (millions)	MCPI	L2 miss CPI
strawman	a program that repeatedly alternates references to two conflicting cache-pages.	16	5.79	4.81
gcc-a	gcc compiling a 17 KB preprocessed program into Sun-3 assembly code.	41	0.81	0.17
gcc-b	gcc compiling a 149 KB program into Sun-3 assembly code.	279	0.50	0.13
cecil	a 6000-line unoptimized Cecil program, including the 20-queens problem, Towers of Hanoi, the game of Life, and compiler test code. (Cecil is a pure object oriented language being developed at the University of Washington [Chambers 93].)	248	0.56	0.16
doduc	Monte-Carlo simulation of the time evolution of a nuclear reactor component described by an 8K input file. (Fortran.)	368	0.38	0.12
tomcatv	a program that generates a vectorized mesh. (Fortran.)	278	1.09	0.77
nasa7-2	Fast Fourier transform component of the SPEC Nasa7 floating point benchmark. (Fortran)	255	0.46	0.06
nasa7-3	Cholesky factorization. (Fortran)	128	0.79	0.31
splot	the X11 program splot run four times on four different input files. Total input file size is 94 KB.	259	0.99	0.15
gs	an X11 Postscript previewer on a 251 KB input file.	667	1.09	0.22

policies introduced in Section 4.2. These measurements quantify the strengths and weaknesses of the policies, and motivate the refinements that result in the design of Snapshot-Miss. Second, I compare Snapshot-Miss to other approaches to cache management: static mapping policies and hardware associativity.

These measurements lead to three conclusions:

- Neither of the static mapping policies consistently outperforms the other—there is no clear "best" static mapping policy.

Table 4-4 Baseline memory system behavior. All counts are in 1000's. The columns labeled "%s" show the percentage of references and misses that are due to operating system activity (rounded to the nearest integer). The column "Write Buffer Stalls" shows the number of cycles spent waiting for writes to the second-level cache to be retired.

	Total	Uncached		Level 1 Instruction Cache				Level 1 Data Cache				Write Buffer			
Benchmark	Refs	Refs	%s	Refs	%s	Misses	%s	Refs	%s	Misses	%s	Writes	%s	Stalls	%s
strawman	16131	3	100	13363	4	25	100	2657	4	2572	0	107	100	36	100
gcc-a	44902	431	100	34178	33	1670	30	6416	29	493	32	3876	30	2011	28
gcc-b	309240	422	100	232728	9	10495	12	45903	7	4104	13	30185	15	12114	13
cecil	273241	671	100	205263	12	6278	23	42127	8	5471	12	25178	9	10922	11
doduc	396803	79	100	287158	1	10209	4	81102	1	4541	4	28461	2	11968	3
tomcatv	305757	74	100	206055	3	346	89	71857	1	12593	2	27770	5	16769	3
nasa7-2	301394	73	100	207742	5	207	90	47318	3	14217	1	46259	1	6176	6
nasa7-3	142526	58	100	95790	4	181	90	32601	2	8716	1	14076	5	6463	5
splot	278812	4091	76	211907	42	10134	60	43473	41	4115	60	19339	44	9493	60
gs	722887	11883	82	538827	23	23414	33	116076	26	9938	38	56099	24	23831	32

- Augmenting either of the static mapping policies with a carefully designed dynamic mapping policies can improve the performance of a system with a direct-mapped cache.

- A system with a hypothetical two-way associative cache with the same size and access time as a direct-mapped cache generally outperforms a system that relies on a dynamic mapping policy to eliminate cache misses. This indicates that there remains additional opportunity to eliminate conflict misses beyond the techniques considered in this chapter.

4.4.1 Comparison of dynamic policies

I measured the effect of each of the policies from Section 4.2 when using an initial mapping of page coloring. I empirically selected policy parameter settings that provided a good balance between reduction in cache misses and policy overhead. That is, I tested different parameter values to find configurations that provided the lowest MCPI. Table 4-5 shows the resulting parameter settings.

Figure 4-1 shows the effect of the dynamic mapping policies and a static page coloring mapping policy on L2 MCPI (that is, the portion of MCPI due to L2 cache misses and cache management). I use L2 MCPI as a metric because the mapping policy has a negligible effect on other components of MCPI. Dynamic policies that increase L2 MCPI compared to static page coloring have policy overhead that overwhelms the improvement in cache miss rates. To distinguish the effects of policy overhead from the effects of cache misses, Figure 4-1

Table 4-5 Parameters of simulated dynamic mapping policies.

Policy	Max recoloring rate	Delay
Active-Naive	Unbounded	None
Active-Delay	Unbounded	100 TLB misses
Active-Throttle	$1/(1 \times 10^6)$ cycles	100 TLB misses
Periodic-Random	$1/(2 \times 10^6)$ cycles	None
Periodic-Color	$1/(2 \times 10^6)$ cycles	None
Snapshot	$1/(2 \times 10^6)$ cycles	None
Snapshot-Delay	$1/(2 \times 10^6)$ cycles	200×10^3 cycles
Snapshot-Miss	$1/(10 \times 10^3)$ L2 misses	1×10^3 L2 misses

divides L2 MCPI into three components: L2 miss CPI, Recolor CPI, and TLBMCPI.

- L2 miss CPI is the memory penalty due directly to misses in the second-level cache (excluding misses incurred by the mapping policy). Those policies that have lower L2 miss CPI than page coloring eliminate cache conflicts.

- Recolor CPI is the cost of copying pages physical memory, including the cache effects of the copy.

- TLBMCPI reflects the TLB management penalty for the given policy. The dynamic policies have higher TLBMCPI than page coloring because they use the TLB to collect information about the current working set, increasing the number or cost of TLB misses, or both.

The figure illustrates the performance effects of the policy refinements that culminate in the Snapshot-Miss policy. Considering the policies in sequence:

- Active-Naive has high recolor CPI because it recolors pages even when conflicts will not persist, increasing L2 MCPI relative to page coloring for all but three programs.

- Active-Delay and Active-Throttle are more cautious in recoloring pages than Active-Naive, reducing Recolor CPI. The reduced recoloring rate does not have an adverse effect on L2 miss CPI for most of the applications. However, these two policies have higher TLBMCPI than Active-Naive because they monitor the TLB continuously. The net effect is that the increase in TLBMCPI is greater than the reduction in L2 miss CPI for most of the benchmarks, again increasing overall L2 MCPI relative to page coloring.

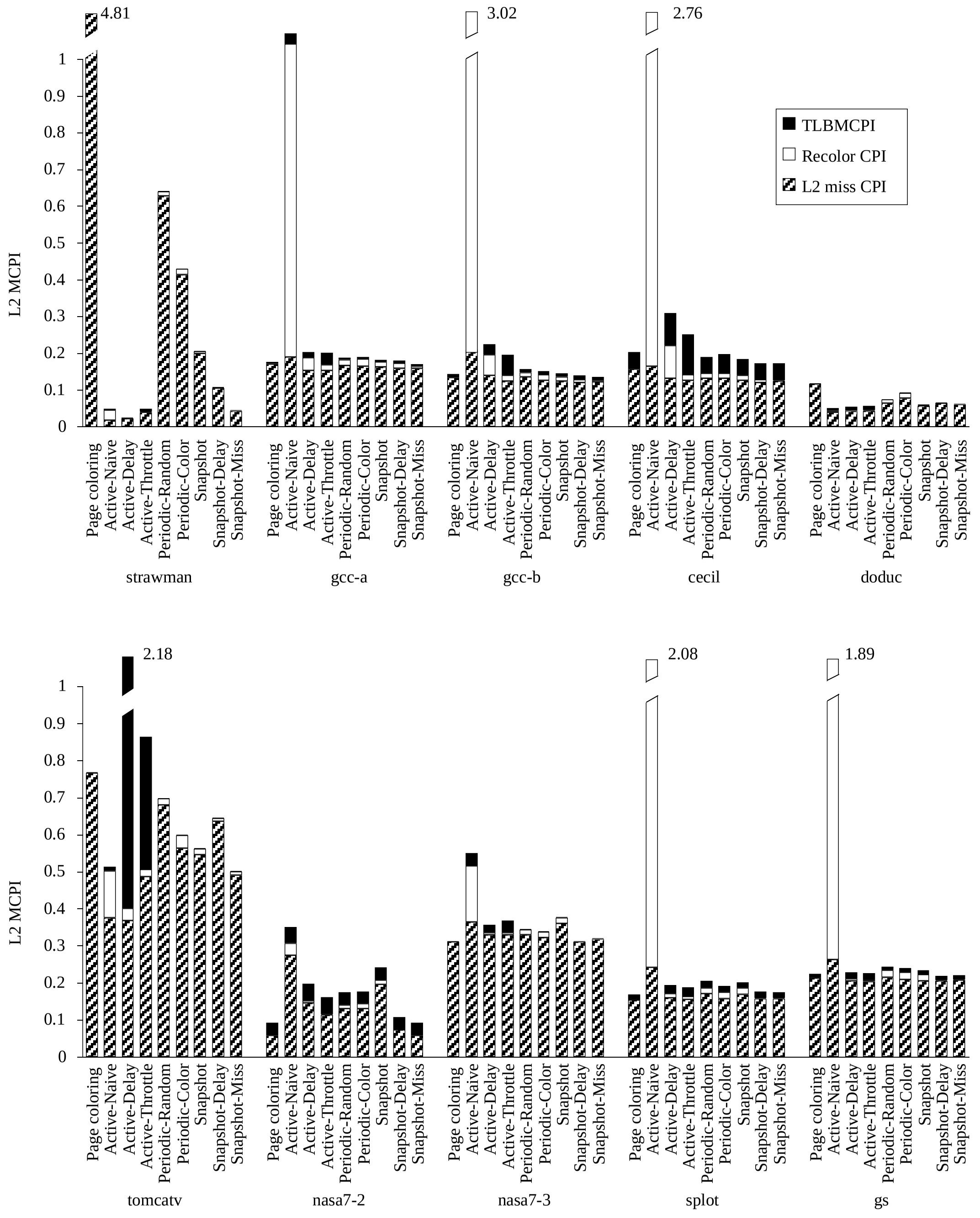


Figure 4-1 Software-based dynamic policies. This figure shows the impact that software-based dynamic mapping policies have on L2 MCPI, MCPI due to second cache misses and management. L2 miss CPI is the cost of second-level cache misses incurred by the application under the given policy. Recolor CPI includes all cycles spent in dynamic mapping policy code, including the instructions executed and cache misses incurred when recoloring pages. TLBMCPI includes all overhead due to TLB misses. Changes in TLBMCPI relative to page coloring reflect the overhead of software-based conflict detection.

- Periodic-Random and Periodic-Color bound TLBMCPI by examining the TLB periodically rather than continuously. For example, for *tomcatv* these two policies have higher L2 miss CPI but lower TLBMCPI than Active-Delay and Active-Throttle, improving overall L2 MCPI. But for several other applications Periodic-Random and Periodic-Color have higher L2 MCPI than page coloring because Recolor CPI exceeds the improvement in L2 Miss CPI.

- Snapshot refines Periodic-Color by flushing the TLB to avoid recoloring pages that are inactive but are still present in the TLB, effectively collecting a snapshot of the current working set. This modification improves performance relative to Periodic-Color for six of the ten programs because conflicts are resolved sooner and with fewer recolor operations.

- Snapshot-Delay refines Snapshot by only recoloring pages that appear in two consecutive snapshots, thus avoiding recoloring pages that conflict only briefly. This change improves performance relative to Snapshot for eight of the ten programs.

- Finally, Snapshot-Miss refines Snapshot-Delay by tying its rate of activity to the cache miss counter instead of to the cycle counter. By attempting to resolve cache conflicts only when the cache miss rate is high, Snapshot-Miss either outperforms or does as well as all of the other dynamic policies. By taking into account both the frequency and the location of potential cache misses, Snapshot-Miss detects cache and eliminates cache conflicts with low overhead and improves upon the performance of static page coloring.

4.4.2 Comparison to associativity and alternate static mapping policies

I now consider the effectiveness of four different strategies for cache management: two static mapping policies, the Snapshot-Miss dynamic mapping policy, and a hypothetical two-way set associative cache with the same size and access time as a direct-mapped cache. Table 4-6 shows L2 MCPI for the benchmark suite when an initial mapping of either page coloring or bin hopping is combined with (1) a direct-mapped cache; (2) a direct-mapped cache and the Snapshot-Miss policy; or (3) a two-way set associative cache.

The table shows that:

- When using a direct-mapped cache alone, neither bin hopping nor page coloring performs best across all applications. For example, page coloring performs better

Table 4-6 Performance of three cache management strategies. This table shows the total cost of second-level cache management (L2 MCPI) for each of three strategies (direct-mapped, Snapshot-Miss, and two-way set associativity), and each of two initial mapping policies (page coloring and bin hopping).

Benchmark	L2 MCPI					
	Page coloring			Bin hopping		
	Direct-mapped	Snapshot-Miss	Two-way associative	Direct-mapped	Snapshot-Miss	Two-way associative
strawman	4.81	0.04	0.02	0.02	0.02	0.02
gcc-a	0.18	0.17	0.12	0.16	0.17	0.12
gcc-b	0.14	0.14	0.10	0.13	0.13	0.10
cecil	0.20	0.17	0.13	0.20	0.17	0.14
doduc	0.12	0.06	0.03	0.04	0.04	0.03
tomcatv	0.77	0.50	0.38	0.39	0.39	0.38
nasa7-2	0.09	0.09	0.10	0.19	0.17	0.24
nasa7-3	0.31	0.32	0.32	0.32	0.32	0.32
splot	0.17	0.18	0.12	0.21	0.20	0.14
gs	0.22	0.22	0.18	0.22	0.22	0.18

for *nasa7-2* and *splot*, while bin hopping performs better for *doduc* and *tomcatv*. This indicates that it is difficult to choose a static mapping policy that will work well across a range of programs.

- Snapshot-Miss improves upon the performance of the two static policies for the majority of programs because it can dynamically adjust virtual-to-physical mappings in response to observed memory reference patterns.

- A two-way associative cache results in a greater reduction in L2 MCPI than Snapshot-Miss because it can eliminate conflicts immediately with no overhead.

- While most of the programs benefit from added associativity, *nasa7-2* and *nasa7-3* do not. For these two programs, L2 miss CPI is dominated by capacity misses rather than conflict misses. As a result, they can perform worse with dynamic policies because capacity misses are misidentified as conflict misses, resulting in analysis and recoloring that only serve to increase overhead.

4.4.3 Policy evaluation: summary

The simulation results in this section show how the quality of information about the cause of cache misses and the cost of collecting that information affect the design and performance of dynamic mapping policies. While an idealized two-way associative cache can eliminate conflict misses with no overhead, a virtual memory mapping policy that is aware of the cache configuration and program behavior can allow a direct-mapped cache to attain some of the benefits of hardware associativity. Even so, this performance improvement was not possible without supplementing the information provided by a software-filled TLB with the contents of a cache miss counter, and without carefully dissecting that information in order to make good recoloring decisions.

4.5 Implementation and performance

The policies that I introduce in this chapter are designed to work using information that could be collected on existing systems. The simulation results show promise for a policy that remaps pages to eliminate cache misses based on the information provided by a software-filled TLB and a cache miss counter. The DEC Alpha provides exactly this hardware support. In this section I describe how I implemented the Snapshot-Miss policy on the Alpha, and how the policy affects the end-to-end execution time of several applications. The measurements confirm that no single static mapping policy works well across all programs, and that a dynamic mapping policy can improve reduce the execution time of programs that suffer from cache conflict misses.

4.5.1 Implementation

I implemented the Snapshot-Miss policy on a DEC Alpha workstation running Version 2.0 of the Digital Unix operating system. The Alpha cache miss counters can be configured to deliver an interrupt to the processor after every 4,096 misses to the second-level cache. The implementation of Snapshot-Miss operates as follows: every 32,768 misses, two snapshots are collected, separated in time by 4,096 misses. A snapshot is collected by flushing the TLB and recording the first 16 pages to incur TLB misses. If two or more pages conflict in both the first and second snapshots, one of the pages is recolored.

The implementation required minimal changes to the vendor's operating system. The policy code is contained in a module devoted to analyzing the snapshots and deciding which page, if any, to recolor. Conflict detection is triggered by a cache miss counter interrupt, which required writing a simple interrupt handler to initiate a snapshot. A modified TLB miss handler collects the snapshots during periods of high cache miss rates. Table 4-7 summarizes the dynamic policy's implementation complexity and overhead.

Table 4-7 Overhead of dynamic policy operations. The cost of delivering cache miss counter interrupts was measured by comparing the execution time of programs with cache miss counter interrupts enabled and then disabled. The other overheads were measured with the Alpha cycle counter. PAL code is Alpha machine language extended with "privileged architecture library" instructions.

Operation	Avg. Cost	Code Size
Deliver cache miss counter interrupt	70 cycles	20 lines of C
Determine whether to record TLB miss	6.5 cycles	4 PAL instructions
Record TLB miss	10 cycles	10 PAL instructions
Analyze TLB miss information	11,000 cycles	200 lines of C
Remap page and update page tables	29,000 cycles	30 lines of C

4.5.2 Performance

For each application I measured the performance of page coloring and bin hopping (the policy that ships with Digital Unix [Bugnion et al. 96]), and then augmented those basic policies with the Snapshot-Miss policy. To dramatize the potential of a dynamic mapping policy to improve performance, I begin by presenting measurements of a synthetic benchmark that is poorly served by static mapping policies. I then present measurements of the end-to-end performance of several applications.

A synthetic workload

To illustrate the potential benefit of the Snapshot-Miss policy, I wrote a synthetic symbol table lookup benchmark, *stab*. The code for *stab* is drawn from the Cecil compiler [Chambers 93], and uses a standard shared library routine to compute a hash function. When the shared library code and the calling routine are mapped to the same cache page and are linked at overlapping page offsets, performance will suffer. In this case, with a static policy, the program executes in 18.3 seconds, whereas with Snapshot-Miss, execution time drops to 9.0 seconds. The conflict between the caller and the callee occurs on every iteration of the loop in the benchmark program, so the Snapshot-Miss policy is able to take advantage of the locality of the cache misses to detect the conflict and recolor one of the pages. With a non-conflicting initial mapping, execution time is 8.6 seconds with either a static policy or Snapshot-Miss. This synthetic benchmark represents a worst-case scenario, with conflicts that static policies cannot avoid. Bin hopping's heuristic of temporal locality is unlikely to hold because the shared library will likely be mapped long before it is called by the client routine. Page coloring's heuristic of spatial locality is unlikely to hold since the caller and the callee reside in separate parts of the virtual address space. As a result, the mappings produced by a static policy are effectively random. The dynamic policy serves to correct such poor mappings created by static policies that cannot accurately predict application reference patterns. However, realistic programs will rarely benefit so dramatically from a dynamic mapping policy.

Application workloads

While the synthetic benchmark described above illustrates that a dynamic mapping policy can in fact eliminate conflict misses on a real system, more interesting is the policy's effect on real applications. I measured the end-to-end performance of a subset of the applications described in the Section 4.3, as well as some new workloads for which I did not have address traces. The workloads, together with elapsed execution times under different mapping policies, are described in Table 4-8 along with relative improvements and overheads. The differences between the policies are due to differences in the virtual-to-physical mappings and, in the case of the dynamic policies, the overhead of monitoring the TLB and recoloring pages. The first only affects the performance of the L2 cache, while the second only increases execution time. Thus any improvement in execution time with the dynamic policies is due to a reduction in conflict misses in the L2 cache.

The table shows that no one static policy performs best for all the benchmarks. Bin hopping does better than page coloring for four of the eight benchmarks (*gcc1*, *tomcatv*, *gs*, and *mpeg*), but page coloring outperforms bin hopping for *doduc*, *nasa7-3*, and *nasa7-6*. Both policies perform about the same for *nasa7-2*.

The table also shows that Snapshot-Miss policy succeeds in significantly reducing execution time for *doduc* and *nasa7-6* with bin hopping, and *tomcatv* with page coloring. The dynamic policy has only a small effect on overall performance for *mpeg, gs*, and *gcc1*. Finally, two applications, *nasa7-2* and *nasa7-3*, are slowed down by the dynamic policy. These programs have working sets much larger than the second-level cache, causing the dynamic policy to recolor in response to capacity misses that it cannot eliminate, degrading performance. For all of the workloads the overhead of the Snapshot-Miss policy was negligible in comparison to the total running time.

Table 4-8 Static vs. dynamic policies. This table shows performance for page coloring and for the native Digital Unix mapping policy (bin hopping), each with and without a dynamic mapping policy (Snapshot-Miss). All of the programs except *mpeg* and *gs* are drawn from the SPEC92 benchmark suite. The programs *mpeg* and *gs* displays images through the X11 server. The columns labeled mean and σ show the mean and standard deviation of the execution time for five runs in milliseconds. The improvement column shows the percentage improvement in execution time of the dynamic policy over the static policy. The overhead column shows the time attributed to dynamic policy overhead in milliseconds. The recolors column shows the total number of pages recolored.

Bench-mark	Description	Page coloring							Bin hopping						
		Static		Snapshot-Miss					Static		Snapshot-Miss				
		exec. time		exec. time		improve-ment	over-head	recolors	exec. time		exec. time		improve-ment	over-head	recolors
		mean	σ	mean	σ				mean	σ	mean	σ			
gcc1	The full gcc1 SPEC benchmark.	112,079	128	112,388	319	-0.3%	135	143	111,548	75	111,335	158	0.2%	103	60
doduc	Nuclear reactor simulation.	23,160	210	23,077	119	0.4%	8	9	23,986	1	23,093	157	3.7%	8	7
tomcatv	The full tomcatv SPEC benchmark.	33,339	6	26,380	53	20.9%	220	345	25,071	24	25,368	32	-1.2%	185	111
nasa7-2	Fast Fourier transform.	26,913	3	27,457	344	-2.0%	33	15	26,912	5	28,296	853	-5.1%	63	35
nasa7-3	Cholesky factorization.	31,571	49	32,540	125	-3.1%	142	55	31,651	3	32,748	162	-3.5%	139	34
nasa7-6	Matrix ma-nipulation.	6,900	11	6,906	21	-0.1%	4	2	7,301	4	6,954	36	4.7%	5	6
gs	ghostscript with a 363 KB input file.	23,236	173	23,109	60	0.5%	20	11	23,188	139	23,017	127	0.7%	19	10
mpeg	mpeg_play with a 217 KB input file.	18,958	591	18,698	409	1.4%	27	19	17,990	117	17,929	114	0.3%	18	5

4.5.3 Implementation and performance: summary

The implementation of Snapshot-Miss on the DEC Alpha shows that it is in fact practical to infer the location of cache conflict misses at runtime, using a cache miss conflict counter to determine when cache misses are a performance problem and using the TLB to collect a snapshot of the application working set during periods of high cache miss activity. The policy can then recolor conflicting pages to eliminate cache misses. The implementation is straightforward, requiring only a few hundred lines of code, and is transparent to the majority of the operating system and to applications. The policy also has low run time overhead, since in the common case it only adds a few instructions to each TLB miss. Although for two applications the policy does recolor in response to capacity misses, slowing execution time, for several applications performance improves significantly. In summary, the implementation results show that even when using incomplete information, a well-designed policy can use careful analysis to identify pages that will benefit from recoloring.

4.6 Conclusions

This chapter provides the second demonstration of my thesis: that by observing memory reference behavior at runtime, the operating system can dynamically match application memory resource requirements to the underlying hardware memory resources. In this case, the resource of interest is a physically indexed direct-mapped cache, and the resource management problem is to map active virtual pages to disjoint cache pages. The key insight is that when virtual pages conflict in the cache, the offending pages can be found using the TLB, and cache miss counts will be high. This motivates the development and implementation of an operating system virtual memory mapping policy, Snapshot-Miss, that detects when cache conflicts are occurring by examining a hardware cache miss counter, and discovers where cache conflict miss are occurring by flushing the TLB and observing the subsequent stream of TLB misses. The policy can then recolor one of the conflicting pages, bringing the system closer to the ideal state in which all active virtual pages are mapped to physical pages of different colors.

This chapter explored how to detect and eliminate cache conflict misses using mechanisms available on current systems. The next chapter builds on these results by considering the question of whether low-cost, low-overhead hardware can provide more detailed information about the cause of conflict misses, and thus enable the design of a simpler and more effective operating system cache management policy.

Chapter 5

Using Virtual Memory and Low-Cost Hardware to Improve Cache Performance

5.1 Introduction

In this chapter I consider how to improve upon the virtual memory policies for cache management introduced in Chapter 4 by adding a low-cost hardware cache miss monitor. To review, cache conflict misses occur in a physically indexed cache when two virtual pages are mapped to physical pages of the same color, that is, to physical pages that map to the same offset in the cache. In Chapter 4 I designed an operating system cache management policy, Snapshot-Miss, that detects cache conflict misses using mechanisms available on existing hardware, namely a software-filled TLB and a cache miss count register. When the operating system detects that two virtual pages conflict, it can eliminate the conflict by recoloring, that is, by mapping one of the pages to a physical page of a different color. This policy suffers from the weakness that it cannot detect cache misses with both precision and low overhead. Precision is only possible at the cost of high monitoring overhead — so high that the benefit of reduced cache miss rates is negated. Reducing overhead results in imprecision, which in turn results in delays in resolving conflicts. In this chapter I introduce a simple hardware device, the Cache Miss Lookaside (CML) buffer, that detects cache misses accurately *and* has low overhead. The CML buffer is an off-chip device that monitors the memory reference stream, and provides per-page cache miss counts to the operating system. Because it is implemented off-chip, it requires no changes to existing CPU implementations. Because it provides precise information about the pages responsible for cache misses, it enables the design of a straightforward operating system mapping policy that responds quickly to cache conflicts with low overhead. The results of trace-driven simulation in this chapter show that a system augmented with a CML buffer can eliminate cache conflict misses more efficiently than Snapshot-Miss, resulting in an overall improvement in simulated application execution time.

The techniques I introduce to obtain these performance improvements rely on virtual memory policies that dynamically manage the mapping from virtual pages to physical pages and hence from virtual pages to cache pages. An ideal virtual memory policy for cache management would arrange that the set of active virtual pages always be mapped to physical pages of different colors, so that they do not cause conflict misses in a physically indexed cache. Realistic policies can approach this ideal by observing memory reference patterns as programs run and remapping virtual pages when cache conflicts are detected. The result is improved cache performance without reprogramming, recompilation, or the collection of profile data. Because these policies are concealed entirely within the operating system, they can benefit both legacy code and new code without requiring any effort on the part of the user, programmer, or compiler. In essence these policies expand the set of resources managed by the operating system to include the cache.

Using the operating system to manage cache resources does carry a cost. When the operating system identifies two virtual pages as causing conflict misses, it recolors one of the pages to eliminate future conflicts. Recoloring requires allocating a physical page of a different color, copying one of the conflicting pages to the new page, and updating the virtual to physical mappings. Altogether this operation costs thousands of cycles. Since a cache miss costs only 10–100 cycles, recoloring must prevent a large number of future cache misses or else it will result in a net *increase* in execution time.

The ability of the operating system to make good recoloring decisions – that is, recoloring decisions that reduce execution time – depends on the information available about the state of the memory system. Ideally the operating system would have future knowledge of the entire memory reference stream, and could analyze this information offline to determine an optimal initial color for each virtual page as well as optimal times to recolor. This approach, though, relies on a separate profiling run, requires extensive offline analysis, and applies only to programs for which behavior in past runs predicts behavior in future runs. Considering instead information that can be collected at runtime, ideally the operating system could examine a complete history of the memory reference stream. The system could then analyze this history to predict which virtual pages would conflict in the future. Collecting this information is impractical, though, since it would require new on-chip hardware to record the address of every memory reference.

In this chapter I consider how to design a low-cost, low-overhead memory system monitor that approximates these ideal sources of information while still allowing the operating system to make good recoloring decisions. In Chapter 4 I also considered this problem, but with the additional requirement that the monitoring mechanism have no new hardware implementation cost – that is, that it could be implemented on an existing system. To summarize the results of Chapter 4, I found that although it is possible to use feedback from existing hardware to make good recoloring decisions, the resulting operating system policy has some shortcomings. That policy, Snapshot-Miss, uses a cache miss counter to determine when cache misses are a problem, and then examines the stream of TLB misses to determine virtual pages could be responsible for the problem. Because the policy monitors memory references at the granularity of pages, not cache lines, it can over- or underestimate the importance of an apparent conflict between virtual pages. Because monitoring the TLB has high overhead, the policy is sometimes suspended, resulting in delays between the onset of a conflict and its resolution. The net result is that although the Snapshot-Miss policy can eliminate enough cache misses to reduce application execution time, it leaves some misses unresolved.

In this chapter I revisit the question of how best to provide feedback to the operating system to eliminate cache conflicts between virtual pages, this time allowing the introduction of new hardware. The specific problem I solve here is how to design low-cost hardware that enables the operating system to quickly eliminate cache conflict misses with low overhead. I introduce a new hardware device, the Cache Miss Lookaside (CML) buffer, that meets the following four design criteria: (1) it has low implementation cost; (2) it imposes low runtime overhead; (3) it requires no modification to existing CPU implementations; and (4) it provides information that enables a straightforward operating system policy to improve end-to-end execution time by quickly and precisely identifying cache conflicts.

The CML buffer maintains a table that summarizes per-page cache miss counts. It collects this information by snooping on the memory bus and observing the address of every reference to physical memory. The CML buffer will see every instruction and load reference that results in a cache miss, and in a write-allocate cache will see every store reference that results in a cache miss. The device meets the design criteria as follows. First, it has low implementation cost because, as I will show, it can be small and can be implemented in low-speed hardware. Second, it has low runtime overhead because it updates cache miss information without software intervention. Third, since it is located off-chip it requires no changes to on-chip structures. Fourth, the simulation results in this chapter show that an operating system policy that uses the information provided by the CML buffer can improve overall application performance by eliminating cache conflict misses.

Because the CML buffer provides better information with lower overhead about the cause of cache conflict misses than is possible with existing hardware, it enables an operating system policy that is simpler and has better performance than the Snapshot-Miss policy. The new policy works for a wide range of system configurations, and for some applications allows a system with a direct-mapped cache to perform nearly as well as a system with an associative cache without incurring the complexity or cost of hardware associativity.

The rest of this chapter

In Sections 5.2 and 5.3 I describe the design of the CML buffer hardware and the software policy that interprets that hardware to decide when and where to recolor pages. I then quantify the benefits of the CML buffer through trace-driven simulation. In Section 5.4 I show that a policy based on the information provided CML buffer outperforms the Snapshot-Miss policy. In Section 5.5 I show that a CML buffer allows a virtual memory policy for cache optimization to improve performance when compared to alternative hardware techniques for eliminating conflict misses, including a realistic two-way set associative cache, a victim cache, and a column-associative cache. In Section 5.6 I present the results of experiments varying the CML buffer hardware and software parameters, as well as the parameters of the simulated memory system, showing that the policy design is largely insensitive to these changes. In Section 5.7 I discuss some open questions. Finally, I conclude in Section 5.8.

5.2 The Cache Miss Lookaside Buffer

The Cache Miss Lookaside (CML) buffer is a hardware device that provides feedback to the operating system about the cause of off-chip cache conflict misses. For the purpose of detecting cache conflicts, the ideal memory system monitor would provide miss counts at arbitrary granularity for all of memory, would incur no runtime monitoring overhead, could be queried by software at no cost, would have low implementation cost, and would interface with an arbitrary memory system and CPU.

The CML buffer design that I describe here approximates this ideal. It maintains miss counts for blocks of memory at a configurable but fixed granularity. It maintains these miss counts for only those blocks that have missed frequently or often. It updates the counts in parallel with the operation of the rest of the system, incurring no runtime overhead. It allows software to query the device with low cost in two ways: by scanning the contents of the buffer through memory-mapped I/O operations, and by arranging to have the device deliver an interrupt when the cache miss count on any page reaches a configurable threshold.

It interfaces to existing systems by monitoring the memory bus, observing every memory reference and hence every cache miss. It does not make any assumptions about the CPU or memory system implementation, other than to assume that the memory bus interface allows the addition of a simple device. The measurements presented later in this chapter show that this design allows the implementation of an operating system policy that detects and eliminates cache conflict misses with low overhead.

5.2.1 The CML Buffer hardware

A block diagram of the CML buffer appears in Figure 5-1. The device is placed between the off-chip cache and main memory and consists of a small array of `(page number tag, counter)` tuples that are associatively indexed by page number. The tuples are broken into three segments: LRU, HOT, and FIXED. On a cache miss to physical page *p*, an associative lookup into the CML buffer is performed using *p* as the lookup tag. If there is a tag match (*p* is already in the CML buffer), then the counter for the matching tuple is incremented. If no match occurs, then the least-recently used tuple in the LRU segment is assigned to *p* and the tuple's counter is initialized to one.

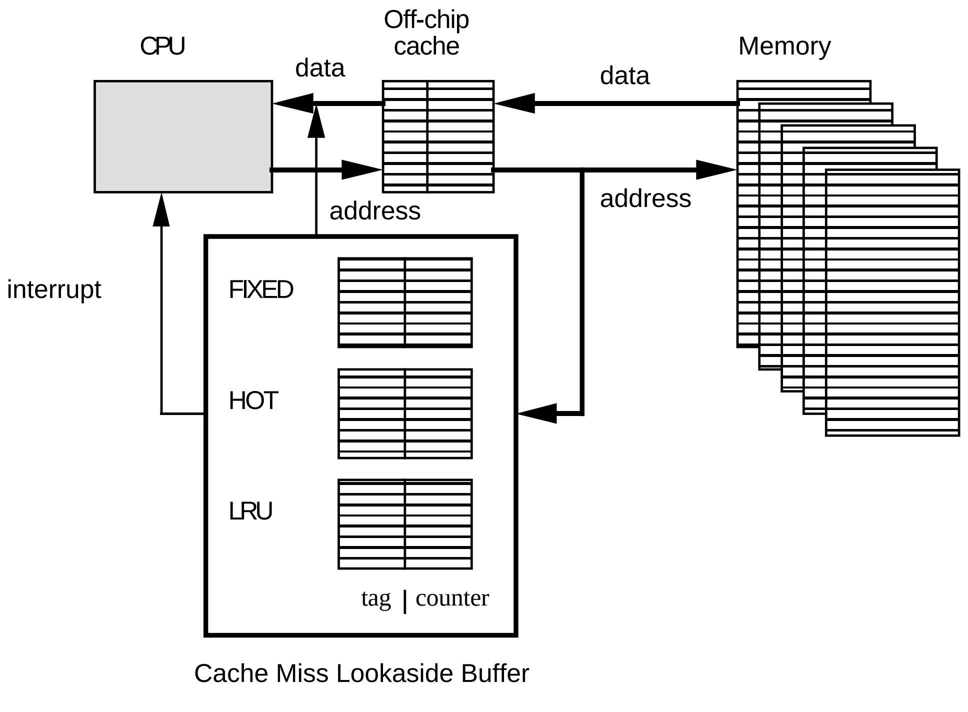


Figure 5-1 The Cache Miss Lookaside Buffer.

The LRU segment reflects information about recent cache misses. To prevent pages with low cache miss counts from displacing heavily conflicting pages from the buffer, pages with high miss counts are automatically promoted from the LRU segment into the HOT segment, as follows. Whenever the count of a tuple in the LRU segment exceeds that of a tuple in the HOT segment, the tuples are swapped. This prevents potentially valuable information about "hot" pages (those suffering many misses) from being evicted by the LRU policy.

The replacement policy for FIXED tuples is managed entirely by software. Applications or the operating system could use these tuples to monitor the cache behavior of specific pages. The FIXED tuples also allow the dynamic detection mechanism implemented in the CML buffer hardware to be refined by future improvements in the operating system cache management policy. This thesis considers only the behavior and use of the LRU and HOT segments.

The size of the LRU and HOT segments are important parameters for CML buffer performance. If these segments are too small, then conflicts can be missed. If they are too large, then the CML buffer cost and complexity increases unnecessarily. The measurements in this chapter are generally based on a simulated CML buffer with LRU and HOT sizes of 16. In Section 5.6.1 I consider the effect of varying the CML buffer size on system performance, and show that this configuration is sufficient for the workloads considered here.

CML Buffer implementation cost

An implementation of the CML buffer would have low implementation cost because the device can be both small and slow. It can be small because it only requires a small number of table entries. It can be slow because the only constraint on the time to update the CML buffer is that it be less than the time to access memory, and because the LRU and HOT replacement policies are simple. As a result the CML buffer should not require custom logic, but could instead be implemented using commodity hardware such as a field programmable gate array (FPGA).

5.2.2 The CML Buffer software interface

The CML buffer is controlled by and provides feedback to the operating system through a simple memory-mapped I/O interface. This interface is intended to allow some flexibility in the operating system policy that uses the CML buffer. The interface allows the operating system to:

- set the CML buffer page size, controlling the granularity at which the CML buffer maintains cache miss counts;
- scan the contents of the CML buffer;
- clear the CML buffer, setting the cache miss counters to zero; and
- set the CML buffer *interrupt threshold* – when a counter in the CML buffer reaches this threshold, the device interrupts the operating system.

Section 5.3 describes the operating system policy that uses this interface.

5.2.3 The Cache Miss Lookaside Buffer: summary

The design of the CML buffer achieves four goals: efficiency, low cost, integrability, and flexibility. First, the device does not increase the cycle time of the main processor or cache. Second, the device is inexpensive to implement enabling its use on a wide range of systems. Third, the device is independent of the CPU and cache, making it easy to apply to both existing and future systems. Finally, the device is flexible, enabling conflicts to be detected under a wide range of memory reference patterns.

5.3 Policy design

An operating system policy that dynamically manages virtual-to-physical mappings to eliminate cache conflicts must answer two fundamental questions. First, when does it identify a page as causing cache conflicts and therefore recolor the page? Second, how does the policy choose the target page of a recolor operation? In this section I describe how an operating system policy based on the information provided by a CML buffer answers these questions. I will refer to this policy simply as "CML".

5.3.1 Identifying conflicting pages

While the CML buffer maintains counts of cache misses to pages, the operating system is responsible for setting CML buffer parameters, and for interpreting the counts and recoloring conflicting pages. The operating system identifies a page as conflicting when it has a large number of cache misses, and has the same color as another page with a large number of cache misses.

To find pages that are conflicting and hence have high miss counts, the operating system relies on the information provided by the CML buffer. The operating system sets the CML buffer interrupt threshold, so that it is notified when some page is suffering from many cache misses. On receiving the interrupt, the operating system scans the CML buffer to find pages that are conflicting. The policy identifies any two or more pages of the same color above a *recolor threshold* as conflicting. It then recolors all but one of the conflicting pages of the same color. To ensure that the CML buffer does not become filled with miss counts for pages that are no longer active, the policy clears the CML buffer once every five million cycles.

Both the interrupt and recolor thresholds should be slightly greater than the number of cache lines per page. If lower, then the small number of compulsory misses that occur when locality changes will result in the operating system incorrectly identifying a page as conflicting. The interrupt threshold should be slightly higher than the recolor threshold to ensure that two or more conflicting pages can be found when an interrupt occurs. Specifically, the experiments in this chapter use a recolor threshold that is equal to the number of cache lines in a page plus 4, and an interrupt threshold that is equal to the recolor threshold plus 60. For a system with 32 byte cache lines and 8 KB pages (256 lines per page), this will result in a recolor threshold of 260 and an interrupt threshold of 320. In Section 5.6.2 I show that this policy works well for a wide range of settings of the two thresholds.

5.3.2 Target page selection

When the operating system selects a page of a new color in order to recolor a conflicting page, it is important to avoid introducing new conflicts in the process. A simple and effective page selection policy, *sequential-target*, is to select a page of a color not recently selected and of a different color than the conflicting pages. In Section 5.6.3 I compare this policy to several others, including one that has complete information about the frequency of references as well as hits to cache pages, and finds that it performs well. In particular, I find that only rarely is the same virtual page recolored more than once, indicating that this policy generally does not result in new cache conflicts when recoloring.

5.3.3 Policy design: summary

The information provided by the CML buffer allows the design of a straightforward operating system policy for cache management. The operating system can quickly (as soon as the cache miss count on a page reaches the interrupt threshold) and easily (by searching the

CML buffer for pages with miss counts above the recolor threshold) identify those pages that are suffering from cache conflicts. In contrast, the Snapshot-Miss policy must infer which pages are causing cache conflicts, because it does not have access to per-page miss counts.

5.4 Policy evaluation

In this section I use trace-driven simulation to evaluate the performance of an operating system cache management policy based on a CML buffer. The methodology, metrics, and workloads are the same as used in the previous chapter, and are described in Section 4.3. To review, the primary metric I use for comparison is MCPI. MCPI is the total number of cycles spent servicing cache misses, write buffer stalls, and memory management (including any overhead incurred by the mapping policy), divided by the number of user and system instructions (excluding memory management instructions). Since these mapping policies primarily affect second-level cache performance, I will often make comparisons based on L2 MCPI, which is defined as memory cycles per instruction due to second-level cache misses and memory management overhead.

I consider the performance of the CML policy and alternative strategies for cache conflict management. I performed experiments with two different initial mapping policies, page coloring and bin hopping. Given these initial mappings, I then simulated the performance of four different systems:

- a traditional direct-mapped cache using a static mapping policy (DM);
- a direct-mapped cache using the Snapshot-Miss page mapping policy (Snapshot-Miss);
- a direct-mapped mapped cache using a dynamic mapping policy that uses the information provided by a CML buffer to detect cache conflict misses (CML); and
- a two-way set associative cache using a static mapping policy (A2).

The comparison between CML and DM shows that CML can improve the MCPI of a system with a direct-mapped cache without requiring hardware associativity. The comparison between Snapshot-Miss and CML shows that an accurate, low-overhead device (the CML buffer) for monitoring the memory system allows an operating system policy to manage the cache more effectively than a policy that uses software monitoring mechanisms. Finally, the comparison between CML and A2 shows that for some applications CML can approach the performance of a hypothetical associative cache with the same size and access time as a direct-mapped cache.

5.4.1 CML policy overhead

I include CML buffer software overhead when computing MCPI. The components of software overhead are the time to scan the CML buffer, the time to analyze its contents, and the time to recolor a conflicting page. The time to scan is proportional to the CML buffer size. The time to analyze the CML buffer contents is proportional to the number of pages detected. The time to recolor is determined by the number of pages on which conflicts have been detected, and includes the time to both copy and remap. Table 5-1 shows the estimated cost of these operations for a system with 8 KB pages using CML. I simulated the cache effects induced by recoloring, so the table does not show a fixed cost for that component.

Table 5-1 CML policy costs for a system with 8 KB pages and a CML buffer with 16 LRU entries and 16 HOT entries.

Conflict analysis	
Scan CML buffer	600 cycles
Analyze CML buffer	300 cycles

Cost to recolor one page	
Copy instructions	4800 cycles
Update page table entry	200 cycles
Cache effects of copying	variable

5.4.2 The effect of the CML Buffer on MCPI

Figure 5-2 shows the MCPI for each benchmark when using an initial mapping policy of page coloring and three cache configurations: DM, CML, and A2. MCPI is shown as an aggregate of various sources, including the CML buffer itself, which reflects the time to scan the CML buffer and recolor pages. Figure 5-2 reveals that:

- For some of the programs (*strawman, doduc* and *tomcatv*) CML performs nearly as well as A2, indicating the presence and resolution of conflicts.

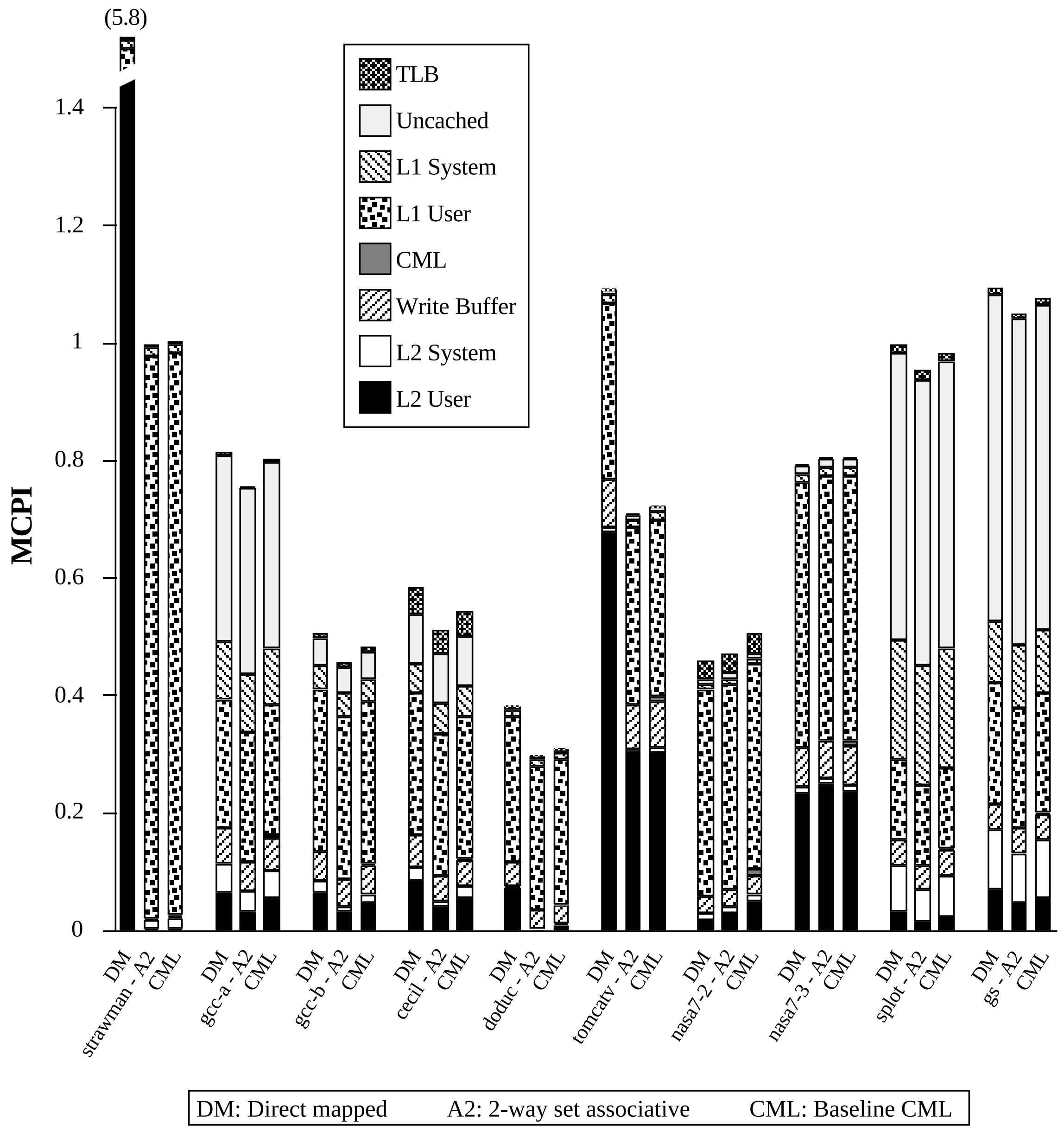


Figure 5-2 Baseline memory system performance. The height of each bar shows the total MCPI for each cache configuration. Each bar is subdivided into a eight components, four of which are largely unaffected by the L2 cache configuration (TLB overhead, uncached references, and L1 cache system and user misses), and four of which vary for the three cache configurations (CML overhead, write buffer stalls, and L2 cache system and user misses).

- For the programs that suffer primarily from capacity misses (*nasa7-2* and *nasa7-3*), neither A2 nor CML improves performance.
- For the remaining programs, CML has a larger MCPI than A2 but smaller than DM. This indicates that A2 is able to resolve conflicts more quickly than CML's threshold-based policy.
- The CML buffer and A2 improve MCPI by reducing the number of L2 user misses, and to a lesser extent by reducing the number of L2 system misses and write buffer stalls.
- CML buffer overhead is insignificant in nearly all cases.

5.4.3 The effect of initial mappings and conflict detection mechanisms

To gauge the effect of the operating system's initial mapping policy and to compare the effectiveness of Snapshot-Miss and CML, I measured the performance of two different mapping policies (page coloring and bin hopping) and four different cache management strategies (DM, Snapshot-Miss, CML, and A2). Figure 5-3 shows the impact on L2 MCPI for each of these configurations. The figure shows that regardless of the initial mapping policy, CML generally outperforms Snapshot-Miss, except for the two applications that suffer from capacity misses (*nasa7-2* and *nasa7-3*). The improvement in performance resulting from using hardware conflict detection rather than software conflict detection is almost entirely due to improvements in L2 miss CPI. This indicates that CML is, as expected, more effective at detecting and eliminating cache conflicts than Snapshot-Miss. The trend seen in Figure 5-2, that CML usually outperforms DM but not A2, continues to hold when the initial mapping policy is bin hopping rather than page coloring.

5.4.4 CML Buffer policy evaluation: summary

The measurements in this section show that the information provided by the CML buffer enables a simple operating system policy to dynamically allocate cache resources to better match the reference patterns of applications. A system that uses simple hardware to measure memory system behavior, and uses an operating system policy to analyze and act upon those measurements, can automatically and transparently improve the performance of a wide range of applications. This section showed that CML, which uses hardware to

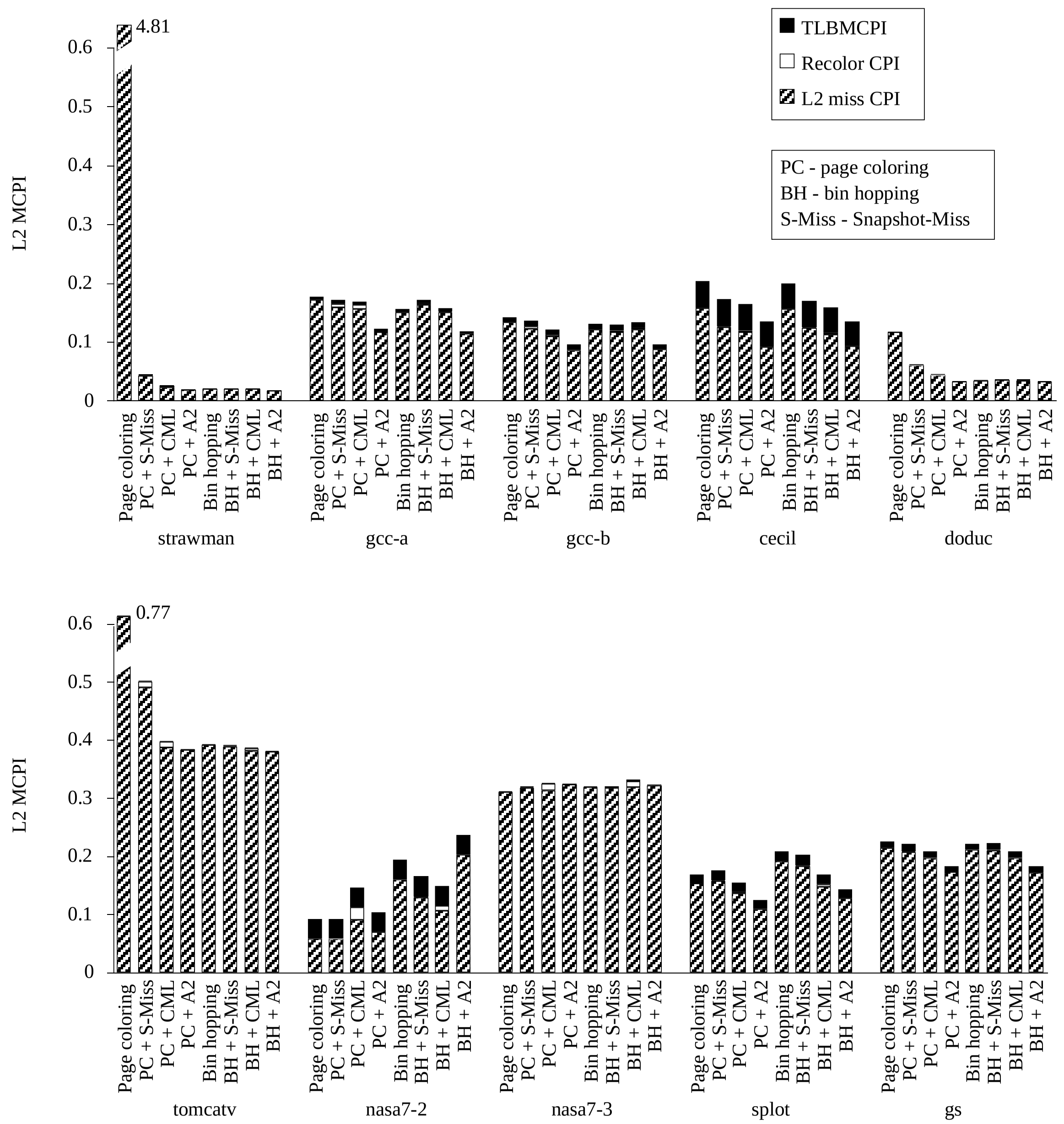


Figure 5-3 Comparing software-based and hardware-based policies. For each benchmark the figure shows two sets of four policies, one set based on page coloring and one based on bin hopping. “S-Miss” is the Snapshot-Miss policy introduced in Chapter 4. CML is a policy based on a CML buffer. A2 is a two-way associative cache. L2 miss CPI is the cost of second-level cache misses incurred by the application under the given policy. Recolor CPI includes all cycles spent in dynamic page mapping policy code, including the instructions executed and cache misses incurred when recoloring pages. TLBMCPI includes all overhead due to TLB misses.

detect conflicts and software to eliminate conflicts of hardware and software, is preferable to Snapshot-Miss, which relies on software alone to detect and eliminate conflicts. The next section will show that CML is also preferable to techniques that use hardware alone to detect and eliminate conflicts.

5.5 Hardware alternatives

The CML buffer enables the operating system to introduce associativity into a direct-mapped cache, eliminating cache conflict misses. In this section I compare the CML buffer to three hardware-only techniques for providing associativity: a two-way associative cache, a column-associative cache [Agarwal & Pudar 93], and a victim cache [Jouppi 90].

The two associative caches require a more complex cache implementation, potentially increasing cache access time. In contrast, the CML buffer requires no changes to the implementation of a direct-mapped cache and hence has no impact on access time. The measurements in this section show that an associative cache or a column-associative cache can outperform a system using direct-mapped cache and a CML buffer only if the associative cache has a small impact on access time.

The effectiveness of a victim cache depends on its size. If the number of conflicting lines exceeds the size of the victim cache, conflict misses will continue to slow execution time. The measurements in this section show that for several of the benchmarks, a system with a direct-mapped cache and a CML buffer outperforms a system with a direct-mapped cache and an eight-entry victim cache.

The conclusion of this section is that conflicts can be eliminated more efficiently by using a combination of hardware feedback and a software policy for cache management than by using hardware alone to provide associativity.

5.5.1 Comparison to a two-way set associative cache

The measurements of A2 in the previous section assume that a large associative cache has the same access time as a similarly sized direct-mapped cache. In practice the A2 cache is likely to be slower than the DM cache [Hill 88, Przybylski et al. 88, Wood 86]. Overall performance then is determined by the cache miss rate and the cache access time. The left side of table 5-2 shows the access time required by the A2 cache for it to perform no worse than CML. For all applications, the A2 cache would perform worse than CML if it adds as little as one cycle to the cache access time.

Table 5-2 Break-even analysis of A2 and column-associative cache vs. CML. An A2 cache requires a cache access time less than the break-even point in order to outperform CML, and a column-associative cache requires that the access time to its slower lines be lower than the break-even point to outperform CML. The percentages show the relative slowdown that can be tolerated by the A2 cache and the slower lines of the column-associative cache. The mean and standard deviation do not include strawman. Nasa7-2 and nasa7-3 are omitted since DM has better performance than both A2 and CML for these benchmarks.

Benchmark	Break-even point			
	A2		Column-Associative	
	cycles	% over baseline (5 cycles)	cycles	% over baseline (5 cycles)
strawman	5.03	1%	5.01	0%
gcc-a	5.71	14%	9.06	81%
gcc-b	5.40	8%	7.70	54%
cecil	5.49	10%	7.08	42%
doduc	5.22	4%	5.68	14%
tomcatv	5.21	4%	5.17	3%
splot	5.44	9%	8.34	67%
gs	5.41	8%	8.02	60%
mean	5.36	7%	7.72	22%
std.dev.	0.21		1.56	

5.5.2 *Comparison to a column-associative cache*

A column-associative cache [Agarwal & Pudar 93] is a two-way associative cache with fast access to lines that would normally hit in a direct-mapped cache. For lines that would conflict in a direct mapped cache, there is fast access to the most recently used line of the set and slower access to the least recently used line of the set. This scheme reduces the cost of conflict misses by allowing conflicting lines to remain in the cache. The right side of Table 5-2 shows the access time required by hits to the slower line for it to perform no worse than CML. On average, the column-associative cache must access the slower lines in fewer than eight cycles to outperform CML. That is, if the column-associative cache requires as few as three additional processor cycles to access the slower cache lines, it will not perform as well as CML.

5.5.3 *Comparison to a victim cache*

A victim cache is a small fully associative cache that stores the last few lines evicted from the main cache. A miss in the main cache that hits in the victim cache is delayed only a few cycles, rather than a full memory access time. The requirement of fast associative lookup constrains the size of the victim cache. The victim cache is therefore effective for eliminating closely-spaced conflicts, as it simulates the behavior of a set associative cache for a small number of lines. If the number of cache misses between cache line reuse is larger than the size of the victim cache, though, the victim cache will be ineffective.

Tomcatv is an example of a program that is poorly served by a victim cache. This program accesses several large matrices that are allocated contiguously in virtual memory. When the operating system mapping policy assigns these arrays to physical pages that overlap in the cache, the conflict misses that occur as the program references the arrays swamp a victim cache. For programs with reference patterns like *tomcatv*'s, as the number of conflicting lines grows, the victim cache must also grow to be able handle the conflict misses. CML, in contrast, can detect and recolor an arbitrary number of conflicting pages using a CML buffer with just a small number of entries.

Table 5-3 Performance of DM, an eight entry victim cache, and CML.

Benchmark	DM	Victim	CML
strawman	5.79	5.04	1.00
gcc-a	0.81	0.81	0.80
gcc-b	0.50	0.50	0.48
cecil	0.56	0.55	0.52
doduc	0.38	0.38	0.31
tomcatv	1.09	0.95	0.72
nasa7-2	0.46	0.46	0.50
nasa7-3	0.79	0.79	0.80
splot	0.99	0.99	0.98
gs	1.09	1.08	1.07

Table 5-3 compares the MCPI of DM, an eight-entry victim cache that sits between the second-level cache and memory, and CML. The measurements assume that there is a penalty of five processor cycles to access the victim cache on a miss to the second-level cache [Jouppi 90]. Compared to the victim cache, CML improves performance by 0.03–0.23 MCPI for three of the programs (*cecil*, *doduc*, and *tomcatv*). For *nasa7-2* the victim cache has better performance than CML by 0.04 MCPI, because CML recolors in response to capacity misses. For the remaining programs (other than *strawman*) the performance of CML and the victim cache differs by at most 0.02 MCPI.

Although CML and the victim cache have similar performance for several of the programs, the CML buffer has three advantages over the victim cache. First, and most importantly, CML improves MCPI by more than the victim cache for three of the benchmarks, because CML can eliminate conflict misses that are distributed among a large number of cache lines. Second, the CML buffer is likely to have a lower implementation cost than the victim cache, because while access time to the CML buffer is not critical for performance, for the victim cache to be useful its access time must be significantly less than the memory access time. Third, the number of conflicts that can be eliminated by the CML buffer is not limited by its size: once a page is assigned to the appropriate color, it will incur no more conflict misses, and hence will not occupy space in the buffer. In contrast, the victim cache can only eliminate as many conflicts as it has entries. The primary disadvantage of the CML buffer compared to the victim cache is that the victim cache eliminates conflict misses immediately, while the CML buffer only eliminates conflict misses after one of the conflicting pages is recolored. However, the results shown in Table 5-3 indicate that for these programs this difference does not result in an advantage for the victim cache.

5.5.4 Hardware alternatives: summary

While these alternative hardware techniques for eliminating cache conflicts can improve application performance, the CML buffer offers several advantages. It has low hardware implementation cost. It requires minimal redesign of existing CPUs and memory systems. It enables an operating system policy that is more effective than a victim cache in eliminating cache conflicts. While it removes fewer conflict misses than an associative cache, it does so without impacting cache access time, resulting in a net improvement in performance if the associative cache increases access time by as little as three processor cycles. By relying on a simple and fast direct-mapped cache implementation, and by relying on operating system software to manage that cache, CML can outperform hardware-only strategies for cache conflict management.

5.6 The effect of CML Buffer design and policy parameters

There are a large number of design parameters for a system that uses a CML buffer, including the size of the LRU and HOT segments, appropriate values for interrupt and recolor thresholds, strategies for selecting a new page when recoloring, and the virtual memory page size. This section shows that nearly any reasonable setting for these parameters results in roughly the same performance improvement. This reflects the fact that the purpose of the CML buffer is to accurately identify persistently bad cache behavior. Consequently, the operating system only requires that the CML buffer raise a "red flag" when a large number of misses to a set of conflicting pages occurs. Exactly how large, how quickly, and how often the red flag is raised matters less than that it is raised at all.

5.6.1 Size of the CML Buffer

The size of the CML buffer reflects a tradeoff between effectiveness and cost. It should be large enough to record all misses due to conflicting pages, but it should not be so large that the implementation and software management costs become unnecessarily high. Figure 5-4 shows the impact of varying the LRU and HOT area sizes. I simulated a range of different CML buffer sizes for four representative applications: *gcc-b* and *cecil* (applications for which CML performs better than DM but not as well as A2), and *doduc* and *tomcatv* (applications for which CML performs as well as A2). For *tomcatv*, misses are concentrated in only a few pages, so a small LRU segment captures all relevant miss activity, and the HOT segment is largely unnecessary. In contrast, *cecil*, *doduc* and *gcc-b* tend to scatter misses over a larger number of pages, requiring either a much larger LRU segment, or a small number of entries in the HOT segment. For a given total CML buffer size, then, a combination of HOT and LRU entries is more effective than HOT or LRU alone. For all workloads, increasing CML buffer size beyond 16 LRU entries and 8 HOT entries did not improve L2 MCPI by more than 0.01.

5.6.2 Threshold selection

The CML policy uses interrupt and recolor thresholds slightly greater than the number of cache lines per page. Lower thresholds will result in compulsory misses being misidentified as conflict misses, while higher thresholds may result in a large number of conflict misses occurring before a page is recolored, leading to a large opportunity cost. The opportunity cost is the cost of conflict misses that accrue before recoloring. Specifically, the experiments elsewhere in this chapter use a recolor threshold that is equal to the number of cache lines in a page plus 4, and an interrupt threshold that is equal to the recolor threshold plus 60. For a system with 32 byte cache lines and 8 KB pages (256 lines per page), this means that the recolor threshold is 260 and the interrupt threshold is 320. In this section I consider the effect of varying these thresholds.

The range of reasonable values for both the interrupt threshold and the recolor threshold turns out to be quite large. For recolor thresholds larger than 256 and interrupt thresholds less than 512, there is little increase (less than 0.01) in MCPI relative to the default

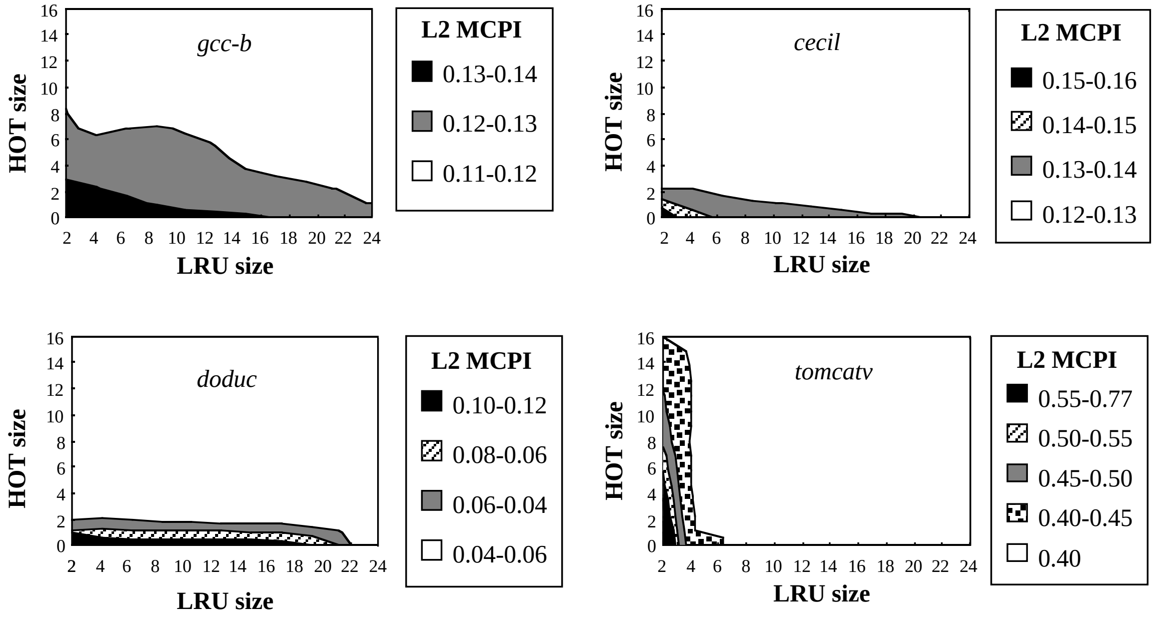


Figure 5-4 Effect of varying CML buffer HOT and LRU sizes for four applications. As CML size increases, MCPI falls. The shading corresponds to incremental changes in MCPI. Closely spaced contour lines indicate a high sensitivity to CML buffer size and configuration.

thresholds. On the other hand, when the recolor threshold is 256 or less, the MCPI of several applications increases (by more than 0.01) because compulsory misses are misidentified as conflict misses. When the interrupt threshold is 1024 or higher, some applications suffer because conflicts are never resolved.

5.6.3 Selecting a new page when recoloring

Recoloring eliminates conflicts by moving the contents of a page of memory from a portion of the cache which is hot to one which is cool, where temperature refers to the level of cache reference activity. The recoloring policy should not select a target physical page that maps to an already hot region of the cache, because if it does so it will merely move conflicts from one portion of the cache to another.

I simulated four different selection policies. The first, *sequential-target,* described earlier and used in the other experiments in this chapter, selects pages by cycling through colors in sequence. The second, *coldest-ref,* selects a target page whose color is that of the coldest (least frequently referenced) cache page. Coldest-ref should accurately identify cold pages, but it is unrealistic since it requires that a counter on every cache page be incremented on each cache reference. The third policy, *coldest-miss*, attempts to approximate coldest-ref by selecting a page whose color is that of the least frequently missing page in the CML buffer. Finally, *random-target* simply selects pages at random. In no case will any of the policies select a target page with a color the same as that being recolored, nor that of another currently conflicting page.

None of the policies performed significantly better or worse than the others. For applications that fit in the cache, there is always a sizable pool of cool cache pages, so the specifics of the policy are not critical. For applications that do not fit in the cache, there is no good color to select, so all the policies perform uniformly poorly. Coldest-miss sometimes performs worse than either sequential-target or coldest-ref because it may select a target page of the same color as a page on which a conflict was recently resolved (the recently resolved page will have a low miss count). In the process it reintroduces conflicts in the target page.

Recoloring stability

When the operating system recolors a page, it may introduce new conflicts, resulting in additional recoloring operations. If this behavior persists, the same virtual page can be involved in multiple conflicts over the program's lifetime revealing an instability in the recoloring strategy. Instability is much more likely to occur if the program is suffering from capacity misses.

To determine whether instability is a problem for these workloads, I counted the number of times each virtual page is involved in a conflict – that is, it contributes to a decision by the operating system to recolor a page, because it is one of a group of pages that have miss counts above the recolor threshold. Figure 5-5 shows that few virtual pages are involved in more than one conflict. This indicates that the CML buffer is usually allows the operating system to resolve conflicts without introducing new ones. The exception is *nasa7-2*, in which pages are copied without improving performance. As noted previously, *nasa7-2* suffers primarily from capacity misses, not conflict misses.

5.6.4 The effect of virtual memory system characteristics

The performance of dynamic mapping policies is affected by several basic parameters of the virtual memory system, including the virtual memory page size and the use of unmapped memory. I performed additional experiments in order to understand the effect of these characteristics on the performance of CML.

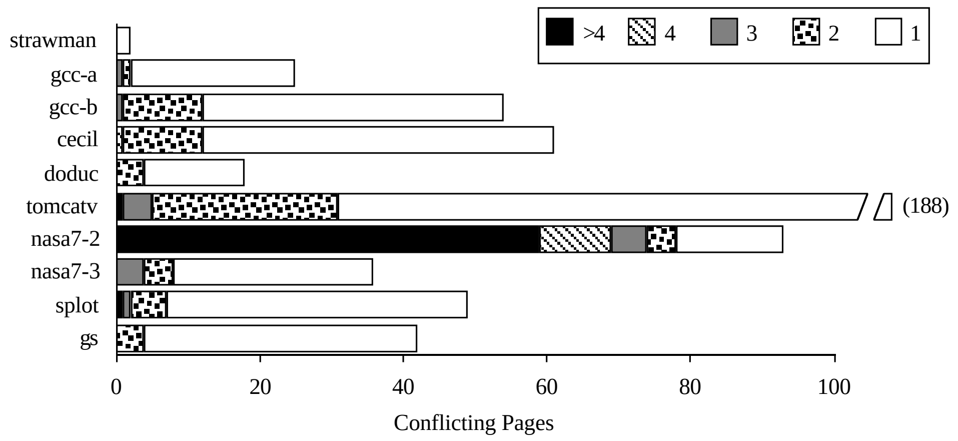


Figure 5-5 Stability of CML. The length of each bar shows the total number of distinct virtual pages involved in conflicts. Each bar is divided into segments showing how many distinct virtual pages were involved in one conflict, two conflicts, etc.. The figure shows that most pages are moved only once.

Page size

Dynamically recoloring pages to eliminate cache conflicts relies on collecting cache miss information on a per-page basis, and on being able to recolor entire pages to remove conflicts. Small pages are therefore advantageous, since they provide fine-grained information about cache miss behavior and fine-grained control over recoloring. As the page size shrinks to approach the cache line size, the operating system can emulate greater cache associativity. As the page size grows to approach the cache size, manipulating the virtual-to-physical mapping to provide associativity becomes less effective. Figure 5-6 shows the effect of varying the page size on L2 MCPI. For each page size, the recolor threshold is set to the number of cache lines in a page plus a small constant (4). The interrupt threshold is set slightly larger (60) than the recolor threshold.

The figure shows that the effectiveness of CML is only slightly sensitive to page size. As page size increases, fewer recolors occur for most applications because of the increased recolor thresholds that follow from the larger page size. For *nasa7-2*, the higher threshold improves MCPI with 16 KB pages because capacity misses are no longer misidentified as conflict misses. For *tomcatv* the increased recolor overhead with larger pages is offset by the reduced cache miss rate. In summary, for this range of page sizes, there is enough associativity in the mapping from virtual pages to cache pages to allow the operating system to use the information provided by the CML buffer to detect and eliminate cache misses.

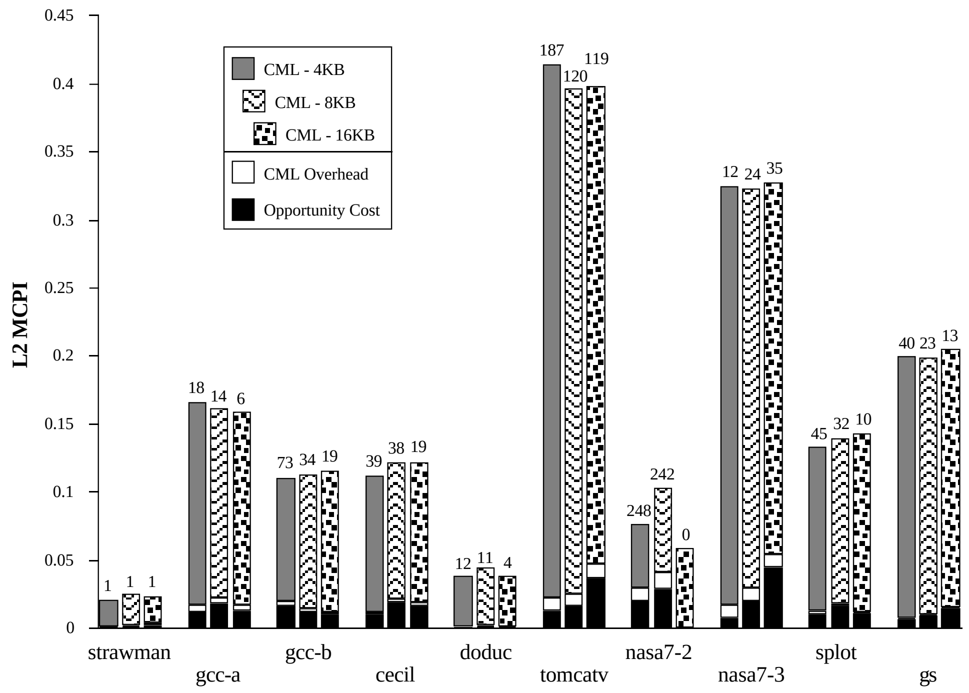


Figure 5-6 The effect of page size. This figure shows L2 MCPI for CML using three different page sizes, as well as the components due to CML buffer overhead and opportunity cost. The number of recolors for each benchmark is shown at the top of each bar.

Mapped kernel memory

In the traced and simulated system, system text and much of its data is in unmapped memory. Unmapped memory preserves TLB entries for user programs, but makes it impossible to resolve conflicts between two unmapped pages. To determine whether unmapped memory has a measurable effect on the ability of the CML buffer to improve performance, I used a version of the simulator in which the kernel's memory is mapped at the virtual memory page size. This change enables kernel-internal conflicts to be resolved using the CML buffer. The result was that overall L2 MCPI improved by at most 0.004 in the best case (for *gs*). This indicates that conflicts entirely confined to the kernel's unmapped memory were rare for these workloads and the simulated memory system, and that increased associativity is not generally beneficial for kernel references alone. For these workloads the majority of conflicts in are either within an application, or between the kernel and an application. In the first case, any conflicting page can be recolored. In the second case, the application's conflicting pages are recolored.

5.7 Open questions

In this chapter and in Chapter 4 I introduced two techniques for detecting cache conflicts, and I showed how the operating system can recolor pages in response to improve cache performance. A number of issues regarding modifications and extensions to these techniques remain, however.

A practical question is how to effectively distinguish capacity misses from conflict misses. Figure 5-5 shows that for a workload such as *nasa7-2* that suffers from capacity misses, the recoloring policies move the same page multiple times during a single run. This suggests that one promising way to avoid recoloring in the presence of capacity misses would be to simply throttle the policy when pages are being recolored too often.

Another issue concerns the parameter settings for the various remapping policies. The policies were tuned to provide a good tradeoff between policy overhead and cache miss rates for the programs under study, but were not subjected to analysis that would, for example, bound the total cost of cache management. While for the systems considered here, the parameter settings worked well, a more rigorous approach to setting parameter values might be useful as cache miss cost and page sizes change in future systems. For example, for very large pages, the high ratio of copying cost to cache miss cost could make the strategy of recoloring as soon as a conflict is detected counterproductive.

These two chapters use a two-way associative cache as an indicator of the opportunity to improve performance by eliminating conflict misses. However, dynamic mapping policies eliminate conflicts in fundamentally different ways from such a cache, because they allow greater associativity, and because the associativity is at the granularity of virtual memory pages rather than cache lines. It might be instructive to consider the performance of a hypothetical fully associative cache with optimal replacement, which would reveal conflict misses that could be eliminated with higher degrees of associativity. Alternatively, analyzing an offline dynamic mapping policy could provide some insight into the limits on performance improvement possible using an online dynamic policy.

Several questions arise concerning the practicality of implementing a CML buffer. For instance, as off-chip memory system implementations grow more complex in order to provide improved latency and bandwidth to the processor, it may well become increasingly difficult to interface an additional device to the memory bus without exacting a performance penalty. Another question is whether the LRU and HOT replacement policies could be implemented precisely in inexpensive logic. The time to maintain even a small sorted list in hardware might be longer than the memory access time, requiring an approximation to the replacement policies simulated in this chapter.

Finally, while the focus of these last two chapters has been on the effect of cache misses on uniprocessor performance, cache misses on multiprocessors due to coherence traffic can also result in significant performance degradation [Rosenblum et al. 95]. An intriguing question, then, is whether the information provided by a CML buffer could drive an application-independent page migration policy in the operating system to reduce coherence traffic. While this question motivated the work by [Sites 95] discussed in Section 2.2, it has not been studied quantitatively. Similarly, the information in the CML buffer could be used to provide feedback to the runtime layer of a distributed virtual memory system. In this case it might be more effective to monitor cache misses at a finer granularity than the virtual memory page size.

5.8 Conclusions

The CML buffer is a low-cost, low-overhead, self-contained device that unobtrusively monitors the memory system and provides information that enables the operating system to effectively manage cache resources. The CML buffer provides exactly the information the operating system needs to determine both when and where the cache is being used ineffectively – that is, when cache misses are occurring and which virtual addresses are responsible for the conflicts. In contrast, the memory system feedback used to drive an operating system policy in Chapter 4 is not always sufficient, because information about the frequency and location of cache misses are decoupled. The result is that a policy based on the CML buffer can eliminate cache misses at runtime more effectively than a policy that relies on standard hardware. While the CML buffer has a small hardware implementation cost, it requires no changes to the CPU or to other components of the memory system. Despite this low cost, it can be nearly as effective as more expensive hardware-only techniques for eliminating cache conflict misses.

Chapter 6

Using Virtual Memory to Improve the Cache and TLB Performance of Windows Applications

6.1 Introduction

In the preceding chapters I introduced operating system policies that automatically improve application performance by dynamically managing either the cache or the TLB to better satisfy application memory system resource requirements. The policies unobtrusively collect information about the relevant behavior of an application, relying on simple hardware to track cache and TLB misses. The policies then analyze this information to determine how to adjust virtual-to-physical mappings to either increase TLB coverage or to eliminate cache conflict misses. I demonstrated the benefits of the resulting dynamic virtual memory mapping policies through a combination of simulation and implementation, using a collection of Unix programs as benchmarks.

In this chapter I focus on two important open questions. First, how should the operating system manage virtual memory to transparently optimize *both* cache and TLB performance? Second, do the operating system policies for cache and TLB management proposed in this thesis improve the performance of Windows applications running on the x86 architecture in addition to Unix applications running on RISC architectures? I answer these questions using trace-driven simulation of the memory system behavior of Windows applications. I first consider the separate effects of the techniques introduced in the previous chapters, and find that Windows applications do in fact benefit from dynamic cache and TLB optimizations. I then measure the effect of a new virtual memory policy that is a simple composition of the two techniques, and find that this policy outperforms policies that use only one of the optimizations. For Windows/x86 applications this new policy reduces application cache miss rates by as much as 77% and TLB miss rates by as much as 99%.

6.1.1 Combining dynamic cache and TLB management

The first question that I address in this chapter, namely how to combine operating system policies for cache and TLB management, is challenging because the techniques introduced in this thesis make conflicting assumptions about page sizes.

- In Chapter 3 I showed how to improve TLB performance by selectively creating large pages at runtime. This technique is effective because increasing the page size increases the amount of memory mapped by a single TLB entry. This increase in TLB coverage in turn reduces TLB miss rates, improving overall application execution time.

- In Chapters 4 and 5 I showed how to improve cache performance through a technique that relies on relatively small pages. In particular, I showed how to change the mapping from virtual pages to physical pages in order to eliminate cache conflict misses. This technique works because in a system with a physically indexed cache, there is associativity in the mapping from virtual pages to cache pages. The degree of associativity is exactly the ratio of the cache size to the page size. As the page size grows, the likelihood that the operating system can find a set of non-conflicting pages to map the application's working set diminishes.

The difficulty in choosing a virtual memory page size to optimize both cache and TLB performance is illustrated by the simple example shown in Figure 6-1. The figure shows a system with a cache that is exactly twice as large as the virtual memory system base page size, and a TLB that holds just one virtual-to-physical translation. Consider an application that repeatedly alternates references to two virtual addresses separated by exactly the cache size. The top half of the figure shows a mapping that will result in good TLB performance: a single page covers both virtual addresses, so that the single TLB entry can map the entire working set. However, this large page must be aligned and contiguous in the physical address space, forcing the two active virtual addresses to map to the same line in the cache. The result is that every reference results in a TLB hit and a cache miss. The bottom half of the figure shows an alternative mapping that will result in good cache performance: separate small pages map the two active virtual addresses. The operating system can map the two virtual pages to physical pages that do not overlap in the cache, so that there will be no cache conflict misses. But now the single-entry TLB cannot map the entire working set. With this mapping, every reference results in a TLB miss and a cache hit.

This example is pathological, but it does illustrate the potential for conflict between the policies proposed thus far in the thesis. To determine whether such conflicts arise for real

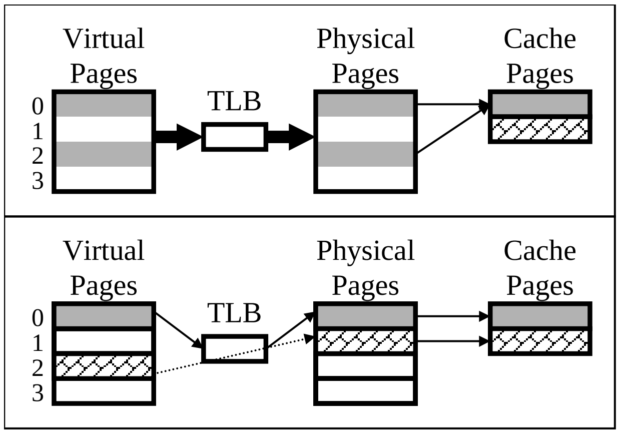


Figure 6-1 The dilemma in choosing page sizes. Each half of the figure shows a possible virtual to physical mapping for a program that actively uses two virtual base pages (the shaded regions of the "Virtual Pages" component of the figure). In a system with a single-entry TLB, the mapping in the top half of the figure (one large page) will optimize TLB performance but result in cache conflict misses, while the mapping in the bottom half of the figure (two small pages) will optimize cache performance but result in TLB capacity misses.

workloads, in this chapter I design a policy that is a simple combination of the policies for superpage construction and dynamic page mapping developed in Chapters 3 and 5. Each of the two components of the new policy dynamically adjusts page colors and sizes as though the other component were not active. The results of simulation show that this approach works well because in practice there is little contention between the two policy components: mappings changed by one component are most often not subsequently changed by the other.

6.1.2 Optimizing cache and TLB performance for Windows/x86 application

The second question that I address in this chapter is whether the optimizations for cache and TLB performance proposed in this thesis apply to personal computer applications. The relevance of a given optimization depends not just on the magnitude of performance improvement but on the relevance of the programs it benefits. The traces of Unix benchmarks running on RISC systems used in the previous chapters represent programs that are popular in the computer science research community, but do not accurately reflect the programs that are popular in the marketplace. Windows/x86 software, not Unix software, accounts for the vast majority of dollars spent on computer applications. Optimizations that benefit popular Windows applications have the potential to have far greater impact than optimizations that apply only to Unix workloads. While in many cases performance enhancements that serve Unix workloads well will also serve Windows workloads well, this is not always the case. Indeed, recent studies show that there can be significant performance differences between the traditional benchmarks used in the computer industry such as SPEC and commercial and desktop productivity applications [Perl & Sites 96, Chen et al. 96, Lee et al. 98].

To test whether the claims of my thesis apply to contemporary software, I present simulation results driven by address traces of Windows/x86 applications. The traces were collected using Etch [Romer et al. 97], a binary rewriting system for arbitrary Windows/x86 executables. The traces include two groups of programs: interactive applications such as Adobe Photoshop and Netscape Navigator, and traditional benchmarks drawn from the SPEC95 suite.

Using these traces, I show that the basic techniques developed in this thesis do benefit this new set of workloads. However, there are two differences between the two classes of programs considered in this chapter (interactive applications and SPEC benchmarks). First, for these programs, the optimizations result in greater benefit for the SPEC benchmarks than for the interactive applications. Second, for some of the interactive applications, the superpage construction policy as described in Chapter 3 has overhead that overwhelms the benefit due to reduced TLB miss rates. I show that adding a simple throttle that temporarily disables the policy when overhead is too high solves this problem. Despite these differences, these optimizations can provide significant performance gains for both classes of applications. For example, I measured reductions in the simulated cache miss rate for Windows Word of 74%, and reductions in the simulated TLB miss rate of the SPEC benchmark *go* of 92%.

6.1.3 Summary

In this chapter I show that:

- the performance benefits of dynamic operating system cache and TLB resource management on Unix/RISC workloads also apply to Windows/x86 applications;

- for some interactive Windows applications, the policy for superpage construction designed in Chapter 3 has high overhead, which can be addressed by modifying the policy to automatically suspend operation when overhead is above a threshold; and

- although the two optimization techniques have the potential to make conflicting page mapping decisions, in practice they rarely interact, so that a policy that both recolors and promotes pages is superior to policies that only recolor or only promote.

In summary, in this chapter I show that the operating system can dynamically adjust virtual-to-physical mappings to simultaneously manage cache and TLB resources, improving the performance of Windows/x86 applications.

The rest of this chapter

In the Section 6.2 I describe the methodology, workloads, and metrics that I use in this chapter. In Sections 6.3 and 6.4 I present measurements of the effect on Windows applications of the operating system policies developed earlier in the thesis that optimize cache and TLB performance separately. These measurements show that the cache and TLB behavior of Windows applications is both important to overall performance and amenable to dynamic optimization. In Section 6.5 I describe the design of a throttling mechanism that bounds the policy overhead of a dynamic superpage construction policy, which would otherwise increase execution time in some cases.

In Section 6.6 I introduce a simple policy that combines dynamic cache and TLB management. I then show how this policy performs for Windows applications, comparing its performance to the separate techniques as well as to a baseline system that performs neither optimization. In Section 6.7 I conclude.

6.2 Methodology

The experiments that I present in this chapter differ from those presented earlier in the thesis in three ways: I consider different applications; I use traces collected on a different architecture; and I evaluate a different policy for managing virtual memory. While these changes make direct comparisons with the results in earlier chapters difficult, they allow memory resource management policies to be evaluated in the context of contemporary and commodity software. I take three steps to better isolate the effects of these changes to the experimental framework. First, I use the traces to drive memory system simulations with the same parameters as those used in Chapters 3 through 5. Second, before designing and evaluating a combined policy for cache and TLB management, I re-evaluate the policies that I developed in the earlier chapters in the context of the new workloads. Third, I evaluate the policies using the same performance metrics as in the preceding chapters.

In the rest of this section I describe the experimental framework in more detail. I describe the workloads and how address traces were collected, and I review the simulated memory system parameters and the metrics used to evaluate performance.

6.2.1 Workloads and trace collection

The measurements in this chapter are based on memory address traces of ten programs running under Windows NT. The traces were collected for a different study of Windows application performance [Lee et al. 98]. Five of the programs are representative of personal computing applications, while five are drawn from the SPEC95 benchmark suite. Table 6-1 describes the programs.

The traces were collected by using Etch to rewrite executables and dynamically linked libraries with calls to instrumentation routines inserted before each instruction and before each load and store. As the rewritten program runs, the instrumentation routines are called and an address trace is written to a file. Details of the trace collection methodology and a description of its limitations can be found in [Lee et al. 98].

6.2.2 The simulated memory system

To predict the effect of TLB and cache optimizations on overall performance, I use the address traces to drive a memory system simulator. The simulator combines the cache and TLB simulators used in Chapters 3 through 5. The memory system parameters are based on those of a DEC 3000/500 Alpha workstation [Dutton et al. 92] with a 150 Mhz Alpha 21064 processor [Dig 92]. Except for the page size and TLB parameters, this is the same memory system that was simulated to collect cache performance data in Chapters 4 and 5. The simulated TLB is the same as that used to collect the TLB performance data in Chapter 3. Table 6-2 summarizes the memory system parameters.

6.2.3 Metrics

The metrics that I use to evaluate performance are the same as those used in Chapters 3 through 5. In particular, I measure memory cycles per instruction (MCPI), as well as several contributors to MCPI, and memory usage overhead. To review, these metrics are defined as follows.

- *MCPI (Memory cycles per instruction)*: The total number of cycles spent servicing cache misses, write buffer stalls, TLB misses, and implementing the cache and/or

Table 6-1 Windows/x86 benchmarks. The SPEC95 applications were compiled with Microsoft Visual C++ (MSVC) 5.0 using the makefiles from the distribution. The benchmarks ran on a Dual Pentium Pro 200 system running Windows NT Workstation 4.0 operating system, service pack 3. This table is taken from [Lee et al. 98]

Application, Instructions (x10^6)	Description
Interactive applications	
acrord32, 420	Adobe Acrobat Reader 3.0: Reader for portable document format (PDF) files. The benchmark loads acrobat.pdf (a 277 KB file) from the standard acrobat reader distribution, and navigates through the document three different ways: through the hyperlinks in the document itself, through the forward and back button provided by acrobat reader, and through a view of the document outline provided by acrobat reader. Finally, the benchmark searches for the word "buy" in the document before closing the program.
netscape, 97	Netscape Navigator 3.1 web browser. The benchmark opens four web pages: `www.cs.washington.edu`, `www.cnn.com`, `www.mtv.com`, and `www.washington.edu`. These pages were viewed on November 12, 1997. The java module for netscape was turned off because Etch does not handle the dynamically generated code generated by the java just-in-time compiler.
photoshp, 1,543	Adobe Photoshop 4.0 image editing package. The benchmark loads fruit.jpg (a 591 KB still-life photograph of fruit) from the standard distribution and applies the *color pencil*, *accented edges*, *diffuse glow*, and *add noise* photo filters to the image.
powerpnt, 121	Microsoft PowerPoint 7.0b slide preparation package. The benchmark loads in a 311 KB 18-page presentation (the presentation included five pages of graphs and six pages of figures in addition to text) in slide mode, scrolls through 3 pages, edits a figure, and continues scrolling through until the end of the document. The benchmark then goes into the outline mode and creates a new page and goes back into the slide mode to move text around. Finally, the benchmark goes into slide sorter mode and moves some slides around.
winword, 368	Microsoft Word 7.0 word processor. The benchmark simulates a user typing in seven paragraphs in an eight page document (document size is 29K). The benchmark then performs four search and replace commands on the document before saving a text version of the file. The interactive spell checker was turned on.
SPEC95 benchmarks	
compress, 403	SPEC95 version of the UNIX compress utility. The benchmark generates a file of a given size using a random number generator then compresses and decompresses the file. The measurements were for a file of size 136K.
gcc, 1,158	SPEC95 version of the GNU C compiler (version 2.5.3) which takes a preprocessed source file and converts these into Sparc assembly. The input to gcc was cccp.i from the test directory in the SPEC95 distribution.
go, 315	SPEC95 version of the game of go. The benchmark was run with a play level of 40 and a 10x10 board size.
perl, 2,013	SPEC95 version of the popular Perl interpreter (version 4.0.1.8). The benchmark was run with jumble.pl and jumble.in from the test directory in the SPEC95 distribution.
vortex, 2,147	SPEC95 version of a single-user object-oriented database transaction benchmark. The input to vortex was the database and schema provided in the test directory in the SPEC95 distribution.

Table 6-2 The parameters of the simulated memory system, based on a DEC 3000/500 Alpha workstation containing a 150 Mhz Alpha 21064 processor.

Page size	4 KB, . . . , 8 MB, in power of two multiples of 4 KB
Line size	32 bytes
First-level cache (L1) Policy Write buffer Miss penalty	8 KB instruction, 8 KB data virtually indexed direct-mapped write-through, read-allocate 4 entries 5 cycles
TLB	32-line ITLB, 32-line DTLB fully associative
Second-level cache (L2) Policy Miss penalty	512 KB, unified, physically indexed, direct-mapped, unified write-back, read/write-allocate 25 cycles

TLB management policy, divided by the number of traced instructions executed (excluding policy instructions).

- *Memory usage overhead*: The increase in memory consumed under a given TLB management policy relative to a policy that uses fixed-size 4 KB pages.

- *TLBMCPI (TLB management cycles per instruction)*: Cycles per instruction spent in service of the TLB misses and in implementing the TLB management policy.

- *L2 MCPI (Second-level cache cycles per instruction)*: Cycles per instruction due to second level cache misses and the cache management policy.

- *L2 Miss CPI (Second-level cache miss cycles per instruction)*: Cycles per instruction due to references that miss in the second-level cache, excluding references induced by the cache and/or TLB management policy.

6.3 The effect of dynamic virtual memory optimizations for cache performance on Windows applications

In this section I reconsider the effect of the *CML* policy introduced in Chapter 5 on cache performance in the context of Windows applications. I show that this policy improves the performance of almost all of these programs, including both interactive applications and SPEC benchmarks. As in Chapter 5, by dynamically remapping pages in response to observed cache miss behavior, *CML* can eliminate cache misses that occur when using a static mapping policy such as page coloring or bin hopping.

To review, *CML* relies on a simple hardware device, the CML buffer, which maintains a summary of per-page miss counts. When one of the counters reaches the *interrupt threshold*, the device interrupts the CPU. On receiving the interrupt, the operating system scans the CML buffer to find pages that may be suffering from conflicts. The operating system policy identifies any two or more pages of the same color (sharing the same cache page) with miss counts above a *recolor threshold* as conflicting. The system then recolors all but one of the conflicting pages of the same color.

Figures 6-2 and 6-3 show the effect of *CML* on the second-level cache performance of Windows applications. To focus attention on the effect of the mapping policy on second-level cache performance, the figures only show L2 MCPI. In the first figure the initial mapping policy is page coloring, while in the second it is bin hopping. For each application the figures show L2 CPI under three different strategies for cache conflict management: a direct-mapped cache with a static mapping policy (*DM*), a direct-mapped cache with the dynamic mapping policy (*CML*), and a hypothetical two-way associative cache with the same size and access time as a direct-mapped cache (*A2*). The figures include "CML CPI", which reflects the overhead introduced by the CML policy

These two figures lead to the following observations, which are consistent with those of Chapter 5:

- The performance of *CML* is generally better than that of *DM*, but not as good as *A2*.

- Neither static mapping policy is effective in avoiding conflict misses across a range of applications. Compared to *DM*, *CML* improves the L2 miss CPI of *winword*, *gcc*, *go*, and *vortex* by at least 0.04 with either initial mapping policy, and by as much as 0.38. This indicates that it is difficult to predict in advance which pages will result in cache conflicts at runtime.

- The overhead of *CML* is very small, contributing at most 0.005 CPI to application execution time. This overhead is less than the improvement in L2 miss CPI for all but one of the workloads (*compress* with bin hopping), resulting in a net improvement in overall performance. *CML* overhead is primarily due to the cost of copying pages.

- For the benchmarks considered in this chapter, *CML* almost never harms performance. While performance in L2 MCPI improved by as much as 0.38 with an initial mapping of page coloring (for *go*) and 0.09 with bin hopping (for *winword*), only in one case did performance suffer, with an increase in L2 MCPI of 0.001 (for *compress* with bin hopping). On a single issue machine with no sources of delay other than the L2 cache, these results would translate into improvements in execution time of 27% and 7.3% respectively, with no more than a 0.1% slowdown.

In addition, the measurements of *go* with an initial mapping of bin hopping reveal an effect not seen in Chapters 4 and 5: the high degree of associativity in the mapping from virtual pages to cache pages can allow performance improvements beyond what is possible with a two-way associative cache. In this one example *CML* actually eliminates more conflict misses than *A2*.

In summary, these measurements show that the conclusions of Chapter 5 apply to Windows programs as well as to Unix workloads. Dynamically remapping pages to eliminate cache conflicts improves performance relative to a static mapping policy across a broad range of applications, including interactive applications such as *photoshp* and *winword*, and SPEC benchmarks such *gcc* and *vortex*.

6.4 The effect of dynamic superpage construction on Windows application TLB performance

In this section I re-evaluate the *APPROX-ONLINE* policy that I developed in Chapter 3 in the context of Windows applications. I show that this policy still significantly reduces TLB miss rates, resulting in a net reduction in TLBMCPI of as much as 0.35 when compared to a system that uses fixed-size 4 KB pages. However, for three benchmarks the overhead of analyzing TLB misses and copying pages exceeds the benefit from reduced TLB miss rates, resulting in a net increase in TLBMCPI of as much as 0.10. In Section 6.5 I show how to modify the policy in order to bound this overhead.

Reviewing Chapter 3, the *APPROX-ONLINE* policy selectively constructs superpages at runtime where they are most likely to eliminate TLB misses, improving TLBMCPI while bounding the overhead due to copying pages. *APPROX-ONLINE* maintains a counter for each potential superpage in the system, reflecting the missed opportunity to avoid TLB misses had the superpage already existed. The counter is incremented on each TLB miss for any superpage that covers both the page that missed and a page already in the TLB.

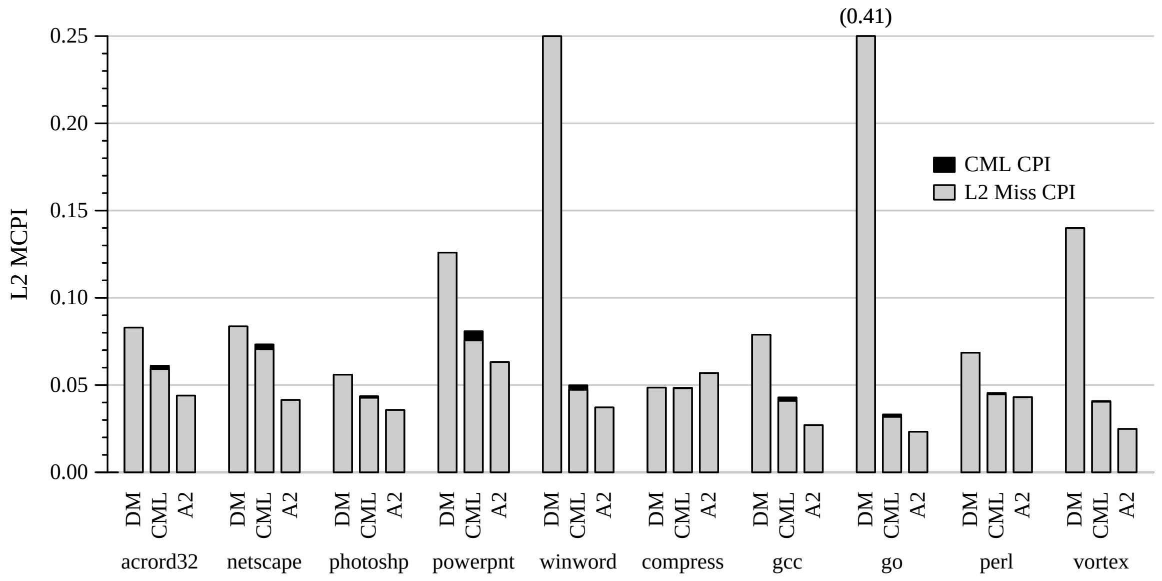


Figure 6-2 Effect of the CML buffer on Windows application performance with an initial mapping of page coloring. On a single-issue machine with no sources of delay other than the second-level cache, execution time is proportional to (1+L2 MCPI).

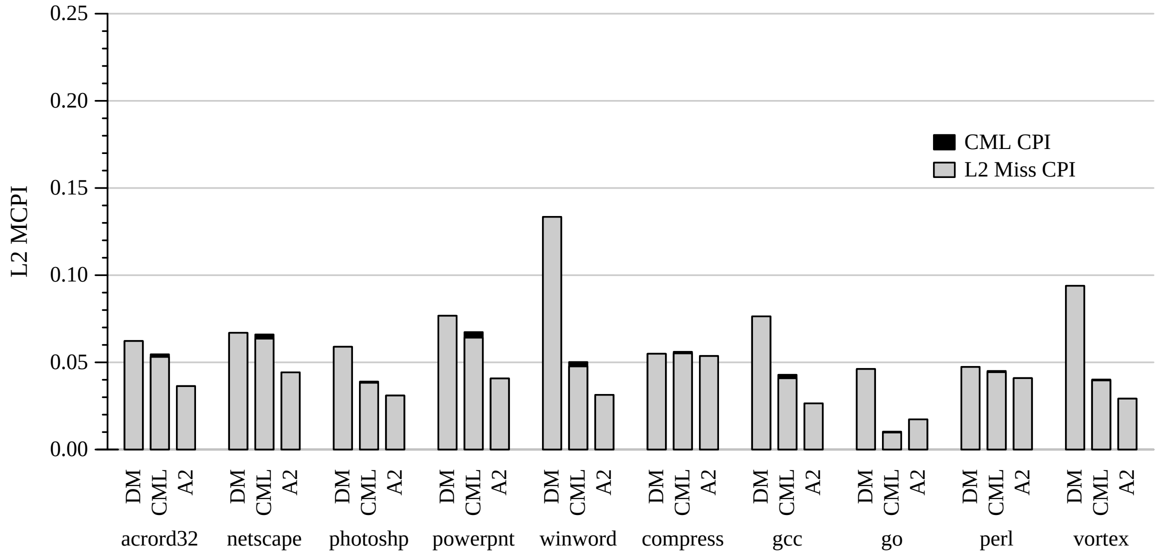


Figure 6-3 Effect of the CML buffer on Windows application performance with an initial mapping of bin hopping.

When the counter for a superpage reaches a threshold value (a parameter of the policy), the superpage is constructed. In the experiments in this chapter, the page placement policy used by *APPROX-ONLINE* when creating a superpage is to choose the cache offset of the target page by cycling through the available cache offsets for the given superpage size.

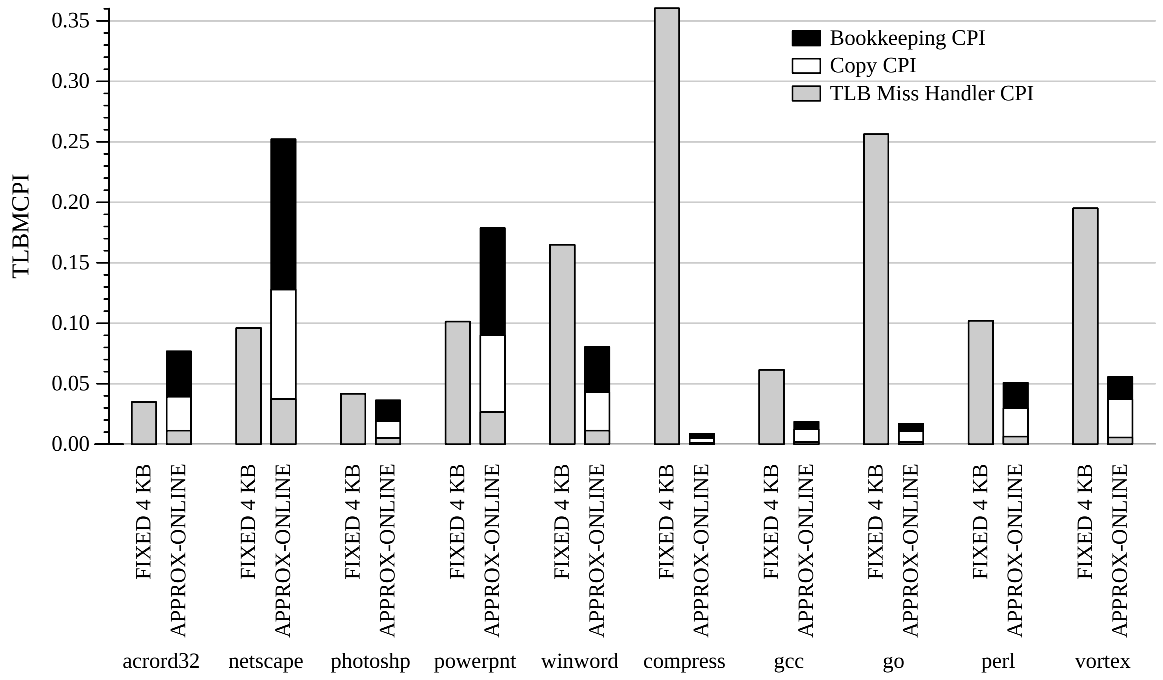


Figure 6-4 Effect of superpage promotion on Windows application TLB performance. For each benchmark two bars are shown, one for the baseline system with fixed size 4 KB pages, and one for the *APPROX-ONLINE* policy. TLBMCPI worsens (increases) for *acrord32, netscape,* and *powerpnt* due to the overhead of the online policy, but improves (decreases) for the remaining seven benchmarks.

Figure 6-4 compares the effect of *APPROX-ONLINE* on performance to a policy that uses fixed size 4 KB pages. The figure shows that *APPROX-ONLINE* reduces the number of TLB misses (reflected by the height of the "TLB Miss Handler CPI" component of the bars) for all of the programs. For seven of the ten programs this results in a net improvement in TLBMCPI. For the remaining three applications (*acrord32*, *netscape*, and *powerpnt*), however, *APPROX-ONLINE* results in a net *increase* in TLBMCPI. The cause of this increase is that the policy does not reduce TLB miss rates enough to balance the increased cost of each miss. In particular, the policy bookkeeping code increases the cost of a single TLB miss from 30 cycles to 130 cycles (a factor of 4.3), but decreases the TLB miss rate by only a factor of 2.6–3.8 for these three applications. In summary, *APPROX-ONLINE* is effective in eliminating TLB misses for Windows applications, but at the cost of increasing overall TLBMCPI for three benchmarks.

6.5 Bounding TLB management overhead by throttling

In the previous section I showed that the *APPROX-ONLINE* superpage management policy actually increases MCPI for three of the benchmarks used in this chapter. In this section I design and evaluate a refinement of *APPROX-ONLINE* that allows reductions in TLBMCPI of as much as 0.35 for these programs, while in the limit bounding the net increase in TLBMCPI to at most 0.02. (The maximum net increase in TLBMCPI is a configurable parameter of the policy.) My approach is to add a throttle to the policy that selectively disables the bookkeeping code in the TLB miss handler. I call the resulting policy *THROTTLE*.

6.5.1 Throttle design

An ideal throttle would enable bookkeeping code only for those TLB misses that will contribute to making good promotion decisions. In practice this is not possible – if the policy knew *a priori* which superpages would benefit TLB performance, it could construct them immediately. Instead, I sought to design a throttle that meets three practical goals:

1. the throttle should be simple, with just a few parameter settings;

2. when the promotion policy can detect and eliminate TLB misses with acceptable overhead, the throttle should be inactive; and

3. when the promotion policy does not result in improved performance, the throttle should place a bound on policy overhead.

I designed the throttle by focusing on the performance problem, namely high bookkeeping overhead. *APPROX-ONLINE* incurs high bookkeeping overhead when (1) the application has a high TLB miss rate, and (2) the policy does not eliminate enough TLB misses to justify the cost of bookkeeping. This suggests that a measurable symptom of poor performance is a persistently high TLB miss rate. To confirm this, I graphed the TLB miss frequency (TLB misses per instruction) as a function of time. I found that for the three applications that are served poorly by *APPROX-ONLINE*, the miss frequency exceeded 0.001 MPI for much of the programs' execution. For the remaining applications, though, the miss frequency was initially high but dropped below 0.001 MPI as superpages were created.

This observation motivates using a throttle that activates when the TLB miss frequency stays above 0.001 MPI. In particular, I introduce a new policy that I call *THROTTLE* which is like *APPROX-ONLINE* except as follows. Every w (window) instructions, the policy checks the TLB miss frequency for the window. If the miss frequency exceeds f (frequency), the throttle activates, suspending bookkeeping and promotion. The throttle remains active until enough instructions have executed to guarantee that TLB policy overhead is less than t (target) CPI. This strategy for bounding overhead is analogous to the page fault frequency approach for controlling thrashing in a virtual memory system [Wulf 69, Silberschatz & Galvin 97].

The throttle has just three parameters, so it meets the first of the design goals. The parameter t can be set to bound the overhead as desired so it meets the third goal. I show through simulation that the policy meets the second design goal.

6.5.2 Parameter selection and sensitivity

The results in this chapter are based on a throttle that bounds policy overhead to no more than $t = 0.02$ CPI. The throttle enforces this target by checking the miss frequency every $w = 10,000,000$ instructions, and suspends bookkeeping if the miss frequency exceeds $f = 0.001$ MPI. I explain here how I arrived at these parameter settings. I also describe how varying the parameters affects the performance of *THROTTLE* for a subset of the benchmarks: *netscape, powerpnt, winword, go*, and *vortex*.

The value of the target (t) parameter is setting is essentially arbitrary, depending how much overhead is deemed tolerable. I chose to set the maximal overhead to 0.02 CPI, so that in the limit the *THROTTLE* policy will not increase execution time by more than 2% on a single-issue machine when compared to a policy that does not promote. For *netscape, powerpnt*, and *go*, varying t between 0.005 and 0.03 CPI has little effect, changing TLBMCPI by less than 0.01. *Winword* and *vortex*, though, are more sensitive the setting of t. These two programs benefit from promotion, and so have the best performance when the throttle is inactive. A high setting of t effectively disables the throttle: for $t \geq 0.04$ *winword* and *vortex* had TLBMCPI within 0.02 of *APPROX-ONLINE*. A low setting of t results in the throttle activating prematurely: for example, setting $t = 0.005$ for *winword* results in almost the same performance as a policy that never promotes. For $0.015 \leq t \leq 0.025$, though, TLBMCPI varied by less than 0.02 for these two programs.

As I described earlier, I found that when *APPROX-ONLINE* is effective in reducing overall TLBMCPI, the TLB miss frequency eventually falls below 0.001 MPI, but otherwise the TLB miss frequency remains above 0.001 MPI. This motivated setting $f = 0.001$. I found that varying f between 0.00025 and 0.0015 resulted in a change in TLBMCPI of at most 0.02. When f is too high, though, the throttle never activates: for example, this occurs when $f \geq 0.003$ for *netscape* or $f \geq 0.002$ for the other four programs.

Finally, the window size w should be neither too small nor too large. If it is too small, the policy will be suspended before it can take action. If it is too large, the benefits of throttling may never be realized; or the policy may be suspended long after the behavior that resulted in high overhead has ceased. For example, with a window size of one, the policy may be suspended as soon as the program incurs its first TLB miss. At the other extreme, with a window size equal to the length of an independent phase of program execution, by the time the policy is suspended the cause of the problem will no longer be relevant. I found that setting $w = 10,000,000$ gave the policy an opportunity to construct beneficial superpages without allowing overhead to grow too large. I found that performance was relatively insensitive to the window size: varying w between $1,000,000$ and $20,000,000$ resulted in a change in TLBMCPI of at most 0.02.

6.5.3 Online superpage construction with throttling: performance

Figure 6-5 shows the effect on TLBMCPI of three strategies for TLB management: a policy that uses fixed-size 4 KB pages, the *APPROX-ONLINE* policy, and the *THROTTLE* policy described above. The figure simply adds the measurements of the *THROTTLE* policy to the results shown in Figure 6-4.

The figure shows that:

- Comparing *THROTTLE* to *FIXED 4 KB*, the superpage construction policy reduces TLBMCPI for seven of the ten applications, and increases TLBMCPI by less than 0.01 for two of the remaining three, indicating that the throttle is successful in bounding policy overhead.

- For *acrord32*, however, *THROTTLE* increases TLBMCPI by more than the target of 0.02. This is an artifact of the relatively short address trace, which covers only a few seconds of execution time. If *acrord32* ran longer, the throttle would remain active longer, reducing policy overhead.

- Comparing *THROTTLE* to *APPROX-ONLINE*, the addition of the throttle allows nearly the same improvement in performance for *photoshp*, *compress, gcc, go,* and

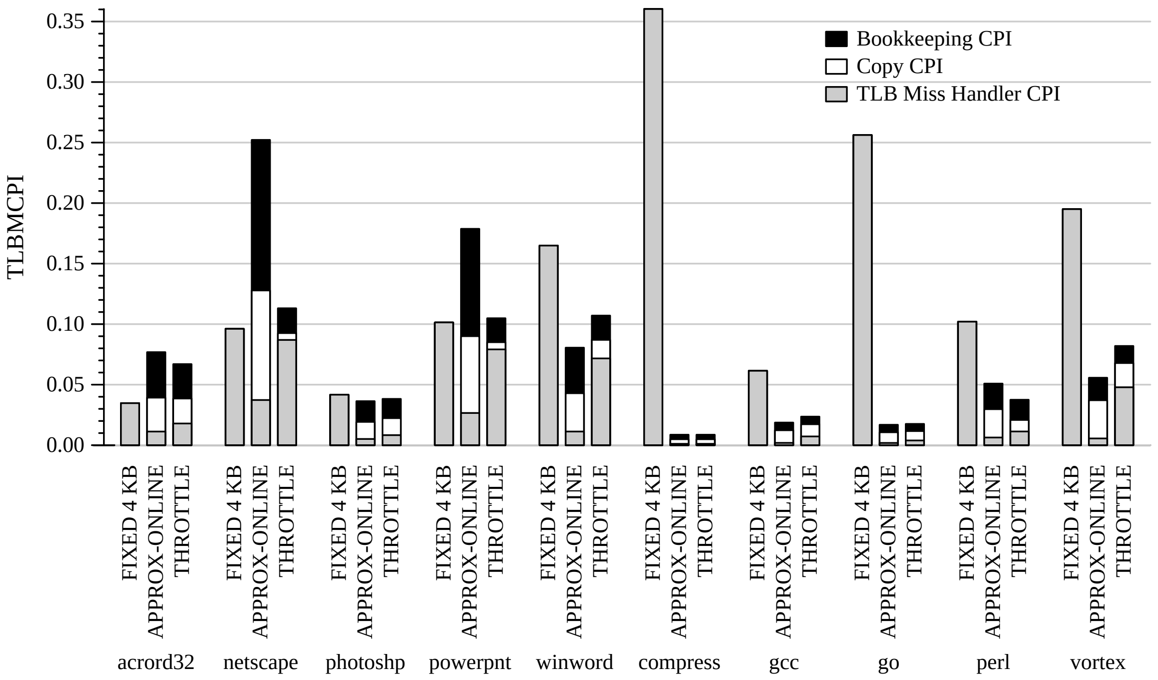


Figure 6-5 Effect of superpage promotion with and without throttling on TLB performance. For each application the effect of three policies on TLBMCPI is shown: a baseline policy that never promotes (*FIXED 4KB*), a superpage construction policy with no throttle (*APPROX-ONLINE*), and a superpage construction policy with a throttle (*THROTTLE*).

perl. For *winword* and *vortex* the throttle does prevent some beneficial superpages from being constructed, although there is still an overall reduction in TLBMCPI.

- Comparing the interactive applications to the SPEC benchmarks, the SPEC benchmarks typically have higher TLBMCPI and benefit more from superpage construction than the interactive applications.

6.5.4 THROTTLE*: summary*

While *APPROX-ONLINE* improves application performance by observing application behavior and dynamically creating superpages, the *THROTTLE* policy further improves performance by observing and modifying the behavior of the superpage promotion policy itself. By selectively disabling TLB monitoring and page copying, *THROTTLE* continues to improve the performance of applications that benefit from dynamic superpage construction, while bounding policy overhead for applications that do not. Compared to *DM*, the net effect is a reduction in TLBMCPI of up to 0.35 for seven of the ten applications, with an increase in TLBMCPI in the limit of at most 0.02.

6.6 Combined TLB and cache optimization

In this section I address the second main subject of this chapter: how to combine dynamic policies for managing cache and TLB resources. As I described in the introduction, the two policies have the potential to conflict: the superpage construction policy will tend to coalesce pages in physical memory in order to optimize TLB performance, while the dynamic mapping policy will tend to scatter pages in physical memory in order to optimize cache performance. Despite this potential for conflict, I show in this section that for the benchmark suite considered here conflicts are rare. This motivates the design of a new virtual memory mapping policy, *COMBINED*, that combines *THROTTLE* and *CML* as though they were independent. I use simulation to show that this policy is more effective in reducing MCPI than either of the separate policies.

6.6.1 COMBINED: A simple combined policy

The policy that I introduce here combines dynamic mapping policies that optimize cache and TLB performance without taking into account their interaction. The reason I use such a simple approach is that the policies are unlikely to collide in practice. This is illustrated by the data shown in Table 6-3. For those virtual pages copied by either *CML* or *THROTTLE*, the table shows the percentage of pages copied by *CML* alone, by *THROTTLE* alone, and by both. For all of the applications fewer than 5% of the pages are copied by both policies, suggesting that if the policies are both active simultaneously there will be little contention. The performance measurements below confirm that the policies rarely interact for the workloads studied in this chapter.

The *COMBINED* policy, then, creates superpages as *THROTTLE* would, and recolors pages as *CML* would. *COMBINED* chooses the cache offset of a target page when recoloring or promoting by cycling through the possible cache offsets for the target page size. When *CML* recolors a page that is a component of a superpage, the superpage is demoted, and only the base page is copied. Although this policy uses simple heuristics to choose when and where to copy pages, it should be able to adapt dynamically in case of a poor decision. For example, if a superpage is created on a physical page that results in cache conflicts, the CML buffer-based component of the policy should detect and correct the conflict.

Table 6-3 Percentage of virtual pages copied by *CML* alone, *THROTTLE* alone, or both policies, using an initial mapping of page coloring. Pages copied by neither policy are not included, so the percentages add to 100.

Benchmark	Percentage of pages copied by		
	THROTTLE	*CML*	Both
acrord32	80	19	1
netscape	95	3	1
photoshp	72	26	2
powerpnt	94	2	3
winword	86	12	3
compress	72	27	0
gcc	26	73	1
go	42	53	5
perl	86	13	1
vortex	49	51	0

6.6.2 Combined cache and TLB optimization: evaluation

To evaluate the performance of *COMBINED*, I measured the simulated effect of the policy on TLBMCPI and L2 MCPI, including the overheads of cache management (CML CPI) and TLB management (Bookkeeping CPI and Copy CPI). Figures 6-6 and 6-7 show the results of these measurements, using initial mappings of page coloring and bin hopping. For comparison, the figures also show the effect on TLBMCPI and L2 MCPI of three alternative policies: a baseline system that neither recolors nor promotes pages (*DM*), *CML*, and *THROTTLE*. Policies that have lower MCPI will result in reduced execution time.

The figures show that

- *COMBINED* reduces MCPI relative to *THROTTLE* for all of the benchmarks except one (*compress* with bin hopping, for which MCPI increases by less than 0.01).

- The improvement of *COMBINED* over *THROTTLE* occurs because *THROTTLE* introduces cache conflicts (*e.g. winword* and *perl* with bin hopping), or because *THROTTLE* ignores existing cache conflicts (*e.g. perl* and *powerpnt* with page coloring), or both (*e.g. go* with page coloring). Adding a dynamic remapping policy to *THROTTLE* allows *COMBINED* to eliminate these cache conflicts.

- *COMBINED* improves performance relative to *CML* for all of the programs except the three that do not benefit from the *THROTTLE* superpage construction policy

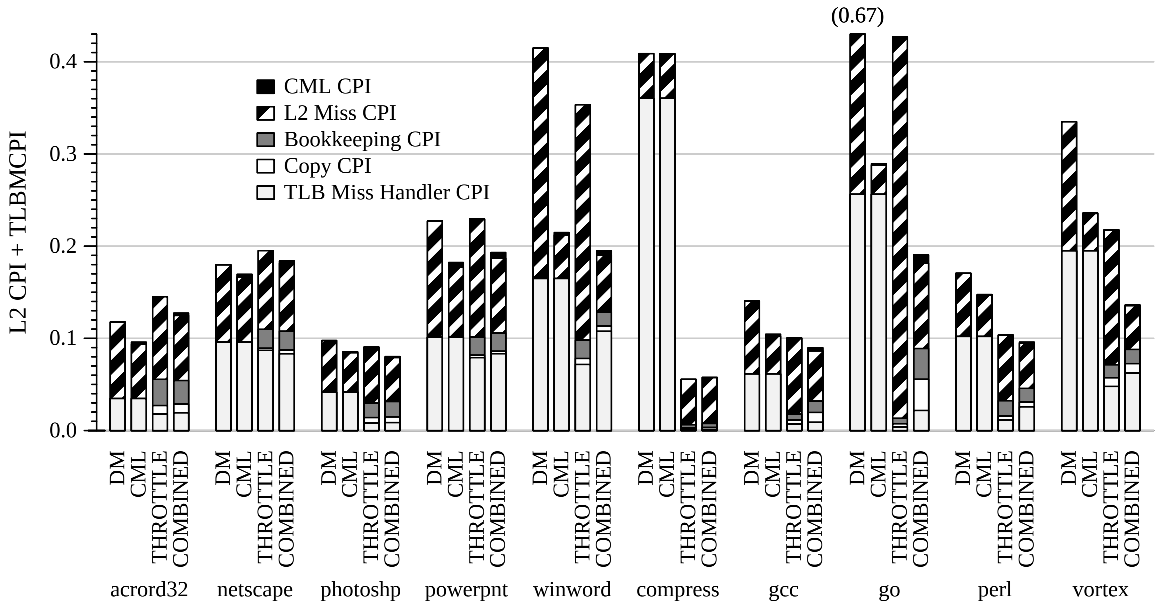


Figure 6-6 Effect of *COMBINED* on cache and TLB performance with an initial mapping of page coloring. This figure shows the sum of L2 MCPI and TLBMCPI for each of four virtual memory mapping policies: *DM*, which neither recolors nor promotes pages; *CML*, which only recolors pages; *THROTTLE*, which only promotes pages; and *COMBINED*, which both recolors and promotes pages.

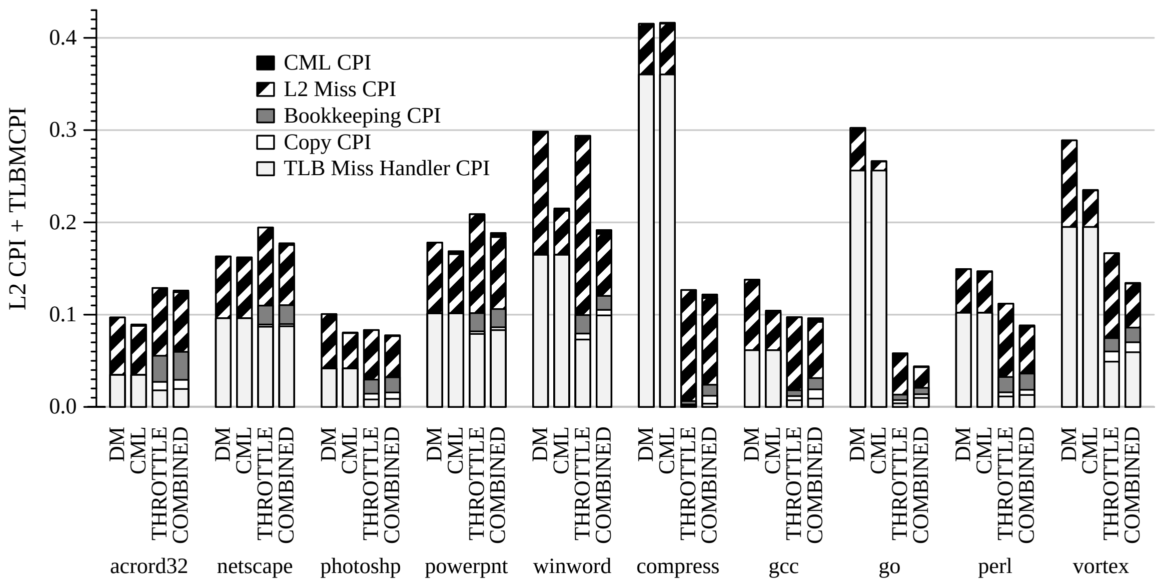


Figure 6-7 Effect of *COMBINED* on cache and TLB performance with an initial mapping of bin hopping. This figure shows the sum of L2 MCPI and TLBMCPI for each of four virtual memory mapping policies: *DM*, *CML*, *THROTTLE*, and *COMBINED*.

(*acrord32*, *netscape*, and *powerpnt*). The improvement of *COMBINED* over *CML* occurs simply because *CML* does not reduce TLB miss rates.

- CML CPI and Copy CPI do not increase perceptibly when *CML* and *THROTTLE* are combined, except for *go* with page coloring, *compress* with bin hopping, and *gcc* with either initial mapping. Despite the increase in overhead, *COMBINED* still has the best performance of the four policies for these benchmarks.

The increase in copying cost for *go* with page coloring suggests that some pages are being copied by both the TLB management and cache management components of *COMBINED*. To measure the extent of this interaction, I counted the number of times each page was recolored multiple times by *CML* and by *COMBINED* when using an initial mapping of page coloring. Tables 6-4 and 6-5 show for each application the percentage of recolor operations due to virtual pages that were recolored once, twice, etc. by *COMBINED* and by *CML*. The tables show that for most of the programs any given page is generally copied

Table 6-4 Percentage of recolor operations due to virtual pages that were copied once, twice, etc. by *COMBINED* with an initial mapping of page coloring.

Benchmark	Number of pages recolored	Percentage of pages copied n times				
		$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n \geq 5$
acrord32	114	50	21	13	7	9
netscape	29	83	7	10	0	0
photoshp	150	78	9	4	5	3
powerpnt	99	47	26	6	4	16
winword	208	34	20	17	4	25
compress	10	70	0	30	0	0
gcc	441	24	23	12	10	32
go	364	7	3	2	2	87
perl	225	57	10	3	4	27
vortex	121	26	10	17	17	31

at most twice. However, for *go* with *COMBINED*, 87% of the pages are copied 5 or more times, indicating that some pages are being alternately promoted and recolored. The result is that for *go* the MCPI due to page copying is 0.03 higher with *COMBINED* than with *THROTTLE*. Although *COMBINED* improves the overall performance of *go* by 0.48 MCPI, it should be possible to improve performance slightly further by restricting the copying rate of the policy.

Table 6-5 Percentage of recolor operations due to virtual pages that were copied once, twice, etc. by *CML* with an initial mapping of page coloring.

Benchmark	Number of pages recolored	Percentage of pages copied n times				
		$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n \geq 5$
acrord32	90	60	18	13	9	0
netscape	32	78	13	9	0	0
photoshp	131	88	5	5	3	0
powerpnt	83	49	22	4	19	6
winword	115	68	21	8	3	0
compress	10	80	20	0	0	0
gcc	265	49	36	10	5	0
go	51	75	20	6	0	0
perl	134	80	9	0	0	11
vortex	50	66	20	6	8	0

6.6.3 The effect of increasing memory latency

The simulation results presented here assume that a second-level cache miss costs 25 processor cycles. As memory access time increases relative to processor cycle time, the number of cycles per cache miss will increase. The effect of increased memory latency can be roughly estimated by scaling the bars in Figures 6-6 and 6-7. For example, to gauge the effect of a four-fold increase in cache miss cost on the cache performance of *COMBINED*, the L2 Miss CPI, CML CPI, and Copy CPI components of each bar should be stretched by a factor of four. (This makes the slightly pessimistic assumption that CML CPI and Copy CPI are dominated by the memory system effects of page copying.) For example, for a program that benefits only slightly from recoloring with low miss costs, such as *gcc* with an initial mapping of page coloring, higher miss costs would have the net effect of increasing the benefit of *COMBINED* relative to *DM* from 0.05 to 0.08 MCPI. For a program that has a large benefit even with low miss costs, such as *winword* with page coloring, the effect of increasing the simulated miss penalty is even higher: the benefit of *COMBINED* compared to *DM* would increase from 0.22 to 0.75 MCPI.

6.6.4 Combined TLB and cache optimization: summary

The simulation results in this section focus on the performance improvement possible with the policy *COMBINED*. My conclusion is that *COMBINED* improves upon the performance of a baseline system that neither promotes nor recolors, and also improves upon the performance of systems that only promote or only recolor. Figures 6-8 and 6-9 support this conclusion, showing the improvement in MCPI of *CML*, *THROTTLE*, and *COMBINED* relative to *DM*. The figures divide the effect on MCPI into improvement of cache performance and improvement of TLB performance.

The figures show three bars for each application, corresponding to improvements in L2 MCPI, TLBMCPI, and total MCPI. For each bar, the top of the bar reflects the improvement in MCPI for an "ideal" policy that attains the combined benefit of *CML* and *THROTTLE* as though the two policies did not interact. The top of the dark portion of each bar shows the actual improvement in MCPI achieved by *COMBINED*. Thus the ideal improvement in TLBMCPI is that of *THROTTLE*; the ideal improvement in L2 MCPI is that of *CML*; and the ideal improvement in total MCPI is the sum of the two. For example, Figure 6-8 shows that with an initial mapping of page coloring, *COMBINED* improves the performance of *go* by 0.47 MCPI, short of the ideal improvement of 0.62 MCPI. More of the improvement came from cache optimization (0.31 MCPI) than from TLB optimization (0.16 MCPI).

The two figures reveal essentially the same trends. I limit the discussion here to Figure 6-8, which shows:

- *COMBINED* results in a reduction in MCPI of as much as 0.62 for eight of the ten benchmarks, and for the remaining two benchmarks (*acrord32* and *netscape*) increases MCPI by less than 0.01.

- Both the cache and TLB optimizations contribute to this improvement in performance.

- As a result, for all but these two benchmarks, *COMBINED* improves performance by more than *CML* or *THROTTLE* alone.

- For six of the benchmarks the total improvement in MCPI is within 0.03 of the improvement possible with an ideal policy that can combine the optimizations implemented by *CML* and *THROTTLE* at no cost.

In summary, these measurements show that the memory system optimization techniques developed in this thesis can be combined in a straightforward way to benefit a wide class of applications. Both programs that have improved TLB performance with *THROTTLE* (*e.g. winword, compress, go,* and *vortex*) and programs that have improved cache performance with *CML* (*e.g. powerpnt, winword, go,* and *vortex*) have lower MCPI with *COMBINED* than with *DM*. Despite the potential for contention illustrated by the pathological

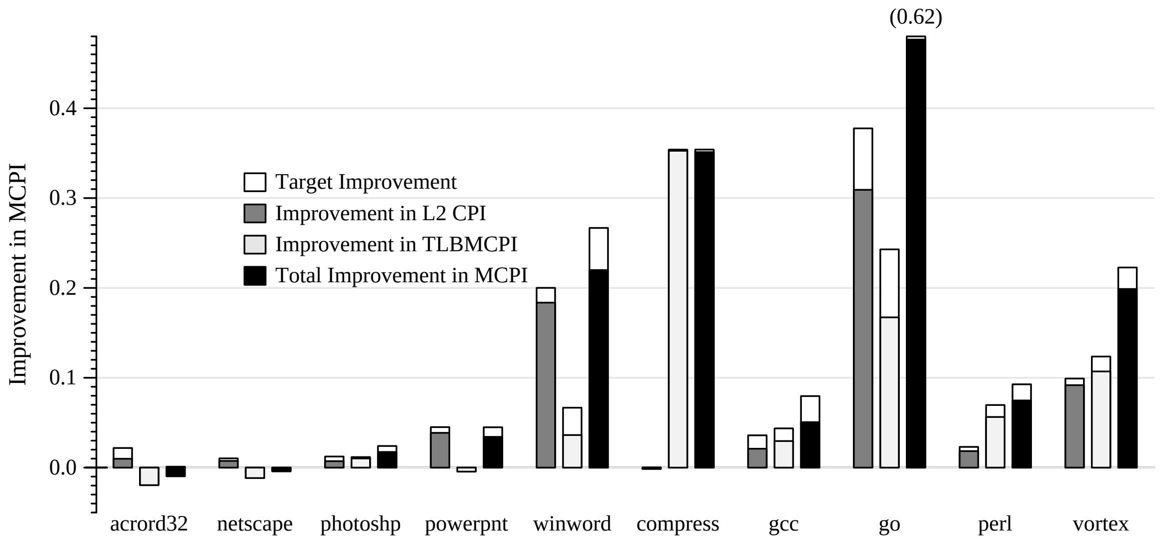


Figure 6-8 The effect of combined cache and TLB optimization on MCPI for Windows applications, with an initial mapping of page coloring. For each bar, the top of the bar reflects the improvement in MCPI of an ideal policy that can improve cache and TLB performance independently, while the top of the dark portion of the bar shows the actual improvement in MCPI achieved by *COMBINED*.

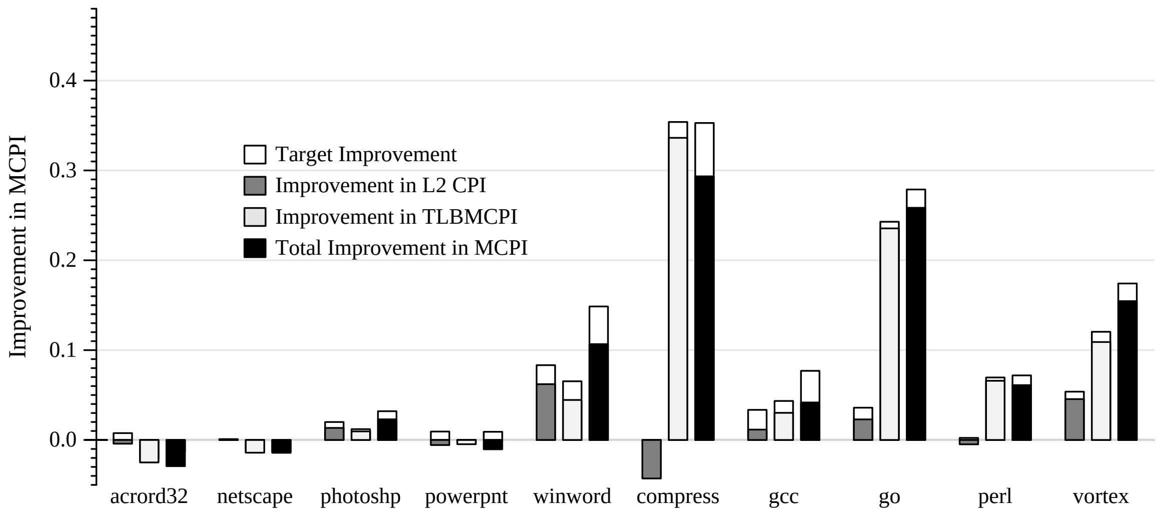


Figure 6-9 The effect of combined cache and TLB optimization on MCPI for Windows applications, with an initial mapping of bin hopping. For each bar, the top of the bar reflects the improvement in MCPI of an ideal policy that can improve cache and TLB performance independently, while the top of the dark portion of the bar shows the actual improvement in MCPI achieved by *COMBINED*.

example given in the introduction to this chapter, the two techniques in fact complement one another.

6.7 Conclusions

In this chapter I provide the final demonstration of my thesis that the operating system can dynamically observe and optimize application memory system behavior. I extend the results of the thesis in two ways. First, I show that the cache and TLB optimizations developed and evaluated in the preceding chapters also apply to memory reference streams characteristic of Windows applications. Second, I show that the two optimizations can be combined to create a new policy, *COMBINED*, that outperforms either of the separate policies.

Although the simulation results in this chapter show that *COMBINED* has the potential to significantly improve the performance of commodity software, there are two significant obstacles to attaining these benefits on commodity *hardware*. First, the policy has a greater impact on programs drawn from from the SPEC benchmark suite than on widely-used interactive applications. Second, the policies developed in this thesis rely on hardware that is not available on current implementations of the x86 architecture. The first of these limitations may reflect small working sets of the interactive benchmarks as tested, or may require additional refinements to the policy design. The second limitation, though, suggests a mismatch between the requirements of these operating system memory resource management policies and the mechanisms available on the most popular commodity processor.

In summary, then, the results in this chapter reveal an opportunity to significantly improve the performance of programs compiled for the popular Windows/x86 platform. For example, the policy developed in this chapter reduces the simulated MCPI of Microsoft Word by as much as 0.22 MCPI. On a idealized single-issue machine with no sources of delay other than the TLB and the second-level cache, this would translate into an improvement in execution time of 15%. However, realizing such gains in practice will require either new implementations of the x86 architecture or additional research to map these techniques onto existing x86 implementations.

Chapter 7
Conclusions

In this thesis I show how to transparently improve the memory system performance of commodity software running on commodity hardware. Because the techniques that I develop improve the performance of the memory hierarchy, their impact on execution time will grow as the gap between memory access time and CPU cycle time grows. Because the techniques do not change the program's interface to the memory system, they can potentially improve the performance of any program without imposing any software development cost on application vendors. Because the techniques do not constrain the architect's implementation of the CPU or the memory system, they can improve the performance of programs running on any hardware platform that provides appropriate mechanisms to allow the operating system to observe and control memory system behavior.

The memory system optimizations that I introduce in this thesis manage cache and TLB resources automatically, without relying on user-level hints or profile-based feedback. As a result they make no imposition on users or on application developers. In contrast to profile-based optimizations, which require a separate profiling run of a program before performing optimization, dynamic operating system optimizations perform both steps while a program runs, improving program performance immediately. This approach also allows the optimizations to benefit programs whose memory system behavior changes as they run or changes depending on the programs' inputs.

The contributions of this thesis are the design and evaluation of three specific operating system policies that dynamically observe memory system behavior and manipulate virtual-to-physical mappings to manage application cache and TLB resource usage. Specifically:

- I introduce an operating system policy that dynamically and selectively creates superpages, increasing TLB coverage and hence reducing TLB miss rates and execution time. This example illustrates the value of good policy design: by carefully balancing the overhead of copying pages with the benefit of reduced TLB misses, the policy can reduce overall execution time despite increasing the cost of each individual TLB miss by more than a factor of four. This example also illustrates the value of good hardware feedback and control mechanisms: the policy relies on precise information about the cause of each TLB miss, and on precise control of the virtual memory page size.

- I introduce operating system policies that dynamically remap pages to eliminate cache conflicts in a direct-mapped physically indexed cache, reducing cache miss rates and again reducing execution time. I develop both a policy that detects conflicts using information available on current systems (Snapshot-Miss), and a policy that detects conflicts using a proposed simple hardware monitor (CML). Snapshot-Miss illustrates the value of relying on stock hardware: I implemented Snapshot-Miss and demonstrated that it speeds the execution of real, not simulated, programs. CML illustrates the value of good feedback: simulations show that by using the additional information provided by a hardware monitor, CML is more effective in improving application performance than Snapshot-Miss.

- I show that these two techniques – dynamic operating system cache and TLB optimizations – can be combined effectively to provide improvements that exceed the performance of either technique alone. I further show that these techniques can improve the performance of programs compiled for the popular Windows/x86 personal computer platform. This example provides a reminder of the importance of focusing on commodity applications: the impact of even small performance enhancements for a installed base of popular software is potentially much higher than performance enhancements for traditional benchmarks that are rarely used in practice.

Together, these contributions show that the operating system can in fact effectively manage the memory hierarchy to improve the performance of existing programs running on existing processors. This is by no means a new idea: for example, the operating system already transparently manages physical memory as a cache of virtual memory. The high cost of page faults justifies careful operating system policies designed to minimize the number of page faults. Similarly, as the cost of cache and TLB misses grows, the opportunity to use software to better manage the use of cache and TLB resources will also grow. This thesis shows that the performance penalty for poor memory resource management is already high enough to justify operating system intervention, and that some existing systems already provide enough hardware support to implement these policies.

The results in this thesis suggest that hardware and operating system designers should co-operate in the design and use of mechanisms that allow software to observe and control the behavior of the memory system. Some recent system design trends are encouraging in this regard:

- Researchers at Digital have proposed hardware support for collecting detailed and accurate samples of program behavior at run-time with low-overhead [Dean et al. 97]. While this support is intended to drive off-line optimizations [Anderson et al. 97], the same mechanisms could be used to drive dynamic optimizations.

- Operating system designers at Hewlett-Packard [Subramanian et al. 98] and SGI [Ganapathy & Schimmel 98] have implemented the virtual memory extensions needed to support multiple page sizes. These mechanisms would enable implementation of the policy for superpage management introduced in this thesis.

- Numerous architectures provide flexible support for a wide range of page sizes [Hunt 95, Heinrich 96, Dig 97, Sun 97].

- Finally, modern CPUs include increasingly detailed hardware performance counters. For example, the Intel Pentium and Pentium Pro allow software to count a wide variety of hardware events [Int 97].

However, further examination of current implementations of the Intel x86 architecture shows that it would be difficult to apply the ideas in this thesis to this popular platform:

- The AMD K6 includes no hardware performance counters.

- The Intel Pentium, despite its wide array of counters, does not allow software to count data TLB misses.

- The x86 handles TLB misses in hardware, preventing software from analyzing the cause of these misses.

- The x86 supports just two page sizes (4 KB and 4 MB), limiting the ability of software to influence TLB performance.

This thesis indicates that the absence of flexible hardware support for software observation and optimization of memory system behavior can result in suboptimal memory system performance. I believe that this support is lacking in current x86 implementation because of the misperception that memory system performance problems can only be solved with specialized, costly revisions to hardware, applications, or compilers. In this thesis I show that in fact hardware feedback can provide information that allows the operating system to resolve memory system performance problems immediately and at low cost. My hope is that in the future computer architects and operating system designers will work together to define memory system interfaces that allow the operating system not just to measure, but to fix, memory system performance problems.

Bibliography

[Agarwal & Pudar 93] 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 Symposium on Computer Architecture*, pages 179–190. IEEE, May 1993.

[Anderson et al. 97] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[Austin & Sohi 96] T. M. Austin and G. S. Sohi. High-Bandwidth Address Translation for Multiple-Issue Processors. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, pages 158–167. IEEE, May 1996.

[Bala et al. 94] K. Bala, F. Kaashoek, and W. Weihl. Software Prefetching and Caching for Translation Buffers. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 243–254. USENIX Assoc., November 1994.

[Bershad et al. 94] B. N. Bershad, J. B. Chen, D. Lee, and T. H. Romer. Avoiding Cache Misses Dynamically in Large Direct-Mapped Caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170. ACM, October 1994.

[Blanck & Krueger 92] G. Blanck and S. Krueger. The SuperSPARC Microprocessor. In *Digest of Papers. COMPCON Spring 1992*, pages 136–141, February 1992.

[Borg et al. 89] A. Borg, R. Kessler, G. Lazana, and D. Wall. Long Address Traces From RISC Machines: Generation and Analysis. WRL Research Report 89/14, Digital Equipment Corporation Western Research Laboratory, 1989.

[Bray et al. 90] B. K. Bray, W. L. Lynch, and M. J. Flynn. Page Allocation to Reduce Access Time of Physical Caches. Technical Report CSL-TR-90-454, Stanford University, 1990.

[Bugnion et al. 96] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-Directed Page Coloring for Multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 244–255. ACM, October 1996.

[Chaiken & Agarwal 94] D. Chaiken and A. Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 314–324. IEEE, April 1994.

[Chambers 93] C. Chambers. The Cecil Language: Specification and Rationale. Technical Report 93-03-05, Department of Computer Science and Engineering, University of Washington, March 1993.

[Chen & Bershad 93] J. B. Chen and B. N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 120–133. ACM, December 1993.

[Chen 93] J. B. Chen. Software Methods for System Address Tracing. In *Proceedings of the 4th Workshop On Workstation Operating Systems*, pages 178–185. IEEE, October 1993.

[Chen 94] J. B. Chen. Memory Behavior For An X11 Window System. In *Proc. Winter 1994 USENIX Conference*, pages 189–200. USENIX Assoc., January 1994.

[Chen et al. 92] J. B. Chen, A. Borg, and N. P. Jouppi. A Simulation Based Study of TLB Performance. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 114–123. IEEE, May 1992.

[Chen et al. 96] J. B. Chen, Y. Endo, K. Chan, D. Mazieres, A. Dias, M. Smith, and M. Seltzer. The Measured Performance of Personal Computer Operating Systems. *ACM Transactions on Computer Systems*, 14(1):3–40, February 1996.

[Chilimbi et al. 98] T. Chilimbi, J. Larus, and M. Hill. Cache-Conscious Pointer-Based Data Structures. Submitted for publication, 1998.

[Chiueh & Katz 92] T. Chiueh and R. H. Katz. Eliminating the Address Translation Bottleneck for Physical Address Cache. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 137–148. ACM, 1992.

[Custer 93] H. Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.

[Dean et al. 97] J. Dean, J. Hicks, C. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th Annual Symposium on Microarchitecture*, pages 292–302. IEEE, December 1997.

[Denning 70] P. J. Denning. Virtual Memory. *Computing Surveys*, 2(3):153–189, September 1970.

[Dig 89] ULTRIX Documentation Group, Digital Equipment Corporation. *ULTRIX Documentation Overview for RISC Processors*, 1989.

[Dig 92] Digital Equipment Corporation. *DECchip 21064-AA Microprocessor, Hardware Reference Manual*, October 1992.

[Dig 97] Digital Equipment Corporation. *Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual*, February 1997.

[Dutton et al. 92] T. Dutton, D. Eiref, H. Kurth, J. Reisert, and R. Stewart. The Design of the DEC 3000 AXP Systems, Two High-Performance Workstations. *Digital Technical Journal*, 4(4):66–81, 1992. Special Issue.

[Ganapathy & Schimmel 98] N. Ganapathy and C. Schimmel. General Purpose Operating System Support for Multiple Page Sizes. In *Proceedings of the USENIX 1998 Annual Technical Conference*. USENIX Assoc., June 1998. To appear.

[Grove et al. 95] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-Guided Receiver Class Prediction. In *OOPSLA'95 Conference Proceedings*, pages 108–123. ACM, October 1995.

[Harty & Cheriton 92] K. Harty and D. R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197. ACM, 1992.

[Hauck & Borriello 97] S. Hauck and G. Borriello. An Evaluation of Bipartitioning Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):849–866, August 1997.

[Heinrich 96] J. Heinrich. *R10000 Microprocessor User's Manual—Version 2.0*. MIPS Technologies Inc., Mountain View, California, 1996.

[Hennessy & Patterson 95] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition*. Morgan Kaufmann, Palo Alto, CA, 1995.

[Hill 87] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD dissertation, University of California at Berkeley, Computer Sciences Division, November 1987.

[Hill 88] M. D. Hill. A Case for Direct-Mapped Caches. *IEEE Computer*, 21(12):25–40, December 1988.

[Hookway & Herdeg 97] R. J. Hookway and M. A. Herdeg. DIGITAL FX!32: Combining Emulation and Binary Translation. *Digital Technical Journal*, 9(1):3–12, 1997.

[Horowitz et al. 96] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, pages 260–270. IEEE, May 1996.

[Hosking & Moss 93] A. L. Hosking and J. E. B. Moss. Protection Traps and Alternatives for Memory Management of an Object Oriented Language. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 106–119. ACM, December 1993.

[Hunt 95] D. Hunt. Advanced Performance Features of the 64-bit PA-8000. In *Digest of Papers. COMPCON '95*, pages 123–128. IEEE, March 1995.

[IBM 95] IBM. *Motorola PowerPC 603e RISC Microprocessor Technical Summary*, 1995.

[Int 97] *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Manual*. Mt. Prospect, Illinois, 1997.

[Jamrozik et al. 96] H. A. Jamrozik, M. J. Feeley, G. M. Voelker, J. E. II, A. R. Karlin, H. M. Levy, and M. K. Vernon. Reducing Network Latency Using Subpages in a Global Memory Environment. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 258–267. ACM, October 1996.

[Jouppi 90] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 364–373. IEEE, May 1990.

[Kane & Heinrich 92] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Karlin et al. 88] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive Snoopy Caching. *Algorithmica*, 3(1):70–119, 1988.

[Keller 96] J. Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. Slides from presentation at Microprocessor Forum, October 1996.

[Kessler & Hill 92] R. Kessler and M. D. Hill. Page Placement Algorithms for Large Real-Indexed Caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.

[Khalidi et al. 93] Y. A. Khalidi, M. Talluri, M. Nelson, and D. Williams. Virtual Memory Support for Multiple Page Sizes. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 104–109. IEEE, October 1993.

[Knuth 73] D. E. Knuth. *The Art Of Computer Programming, 2nd Edition, Volume 1*. Addison Wesley, Reading, Massachusetts, 1973.

[Lam et al. 91] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74. ACM, April 1991.

[LaMarca 96] A. G. LaMarca. *Caches and Algorithms*. PhD dissertation, Department of Computer Science and Engineering, University of Washington, 1996.

[Lee et al. 98] D. Lee, P. Crowley, J.-L. Baer, T. Anderson, and B. Bershad. Execution Characteristics of Desktop Applications on Windows NT. In *Proceedings of the 24th Annual Symposium on Computer Architecture*. IEEE, 1998. To appear.

[Lynch 93] W. L. Lynch. *The Interaction of Virtual Memory and Cache Memory*. PhD dissertation, Computer Systems Laboratory, Stanford University, November 1993. Technical Report CSL-TR-93-587.

[McMurtry 86] L. McMurtry. *Leaving Cheyenne*. Texas A&M University Press, College Station, Texas, 1986.

[Mogul & Borg 91] J. C. Mogul and A. Borg. The Effect of Context Switches on Cache Performance. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84. ACM, April 1991.

[Mogul 93] J. Mogul. Big Memories on the Desktop. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 110–115. IEEE, October 1993.

[Perl & Sites 96] S. E. Perl and R. L. Sites. Studies of Windows NT Performance Using Dynamic Execution Traces. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 169–184. USENIX Assoc., October 1996.

[Przybylski et al. 88] S. A. Przybylski, M. Horowitz, and J. L. Hennessy. Performance Tradeoffs in Cache Design. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 290–298. IEEE, May 1988.

[Ramakrishnan et al. 93] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL Deductive Database System. In *Proceedings of ACM SIGMOD Internation Conference on Management of Data*, pages 167–176. ACM, May 1993.

[Romer et al. 94] T. H. Romer, D. Lee, B. N. Bershad, and J. B. Chen. Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 255–266. USENIX Assoc., November 1994.

[Romer et al. 95] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, pages 176–187. IEEE, June 1995.

[Romer et al. 97] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the Usenix Windows NT Workshop*, pages 1–8. USENIX Assoc., 1997.

[Rosenblum et al. 95] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 285–298. ACM, December 1995.

[Silberschatz & Galvin 97] A. Silberschatz and P. B. Galvin. *Operating System Concepts, 5th Edition*. Addison Wesley, Reading, Massachusetts, 1997.

[Singhal & Goldberg 94] A. Singhal and A. J. Goldberg. Archictectural Support for Performance Tuning: A Case Study on the SPARCcenter 2000. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 48–59. IEEE, April 1994.

[Sites 95] R. L. Sites. Method and Apparatus for Cache Miss Reduction by Simulating Cache Associativity. United States Patent No. 5,442,571, August 1995.

[Smith 82] A. J. Smith. Cache Memories. *ACM Computer Surveys*, 14(3):473–530, September 1982.

[Srivastava & Eustace 94] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the 1994 ACM Symposium on Programming Languages Design and Implementation*, pages 196–205. ACM, June 1994.

[Subramanian et al. 98] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath. Implementation of Multiple Pagesize Support in HP-UX. In *Proceedings of the USENIX 1998 Annual Technical Conference*. USENIX Assoc., June 1998. To appear.

[Sun 97] Sun Microelectronics. *UltraSPARC-II Data Sheet: Second Generation SPARC v9 64-Bit Microprocessor With VIS*, July 1997.

[Talluri & Hill 94] M. Talluri and M. D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–182. ACM, October 1994.

[Talluri 95] M. Talluri. *Use of Superpages and Subblocking in the Address Translation Hierarchy*. PhD dissertation, Computer Sciences Department, University of Wisconsin, August 1995.

[Talluri et al. 92] M. Talluri, S. Kong, M. D. Hill, and D. Patterson. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 415–424. IEEE, May 1992.

[Talluri et al. 95] M. Talluri, M. D. Hill, and Y. A. Khalidi. A New Page Table for 64-bit Address Spaces. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 184–200. ACM, December 1995.

[Taylor et al. 90] G. Taylor, P. Davies, and M. Farmwald. The TLB Slice – A Low-Cost High-Speed Address Translation Mechanism. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 355–363. IEEE, May 1990.

[Uhlig et al. 94] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest, and R. Brown. Design Tradeoffs for Software-Managed TLBs. *ACM Transactions on Computer Systems*, 12(3):175–205, August 1994.

[Wahbe et al. 93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, December 1993.

[Wall 92] D. W. Wall. *Code Generation — Concepts, Tools, Techniques*, chapter Systems for Late Code Modification, pages 275–293. Springer-Verlag, 1992.

[Wang & Baer 91] W.-H. Wang and J.-L. Baer. Efficient Trace-Driven Simulation Methods. *ACM Transactions on Computer Systems*, 9(3):222–241, August 1991.

[Wheeler & Bershad 92] B. Wheeler and B. N. Bershad. Consistency Management for Virtually Indexed Caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 124–136. ACM, October 1992.

[Wood 86] D. A. Wood. An In-Cache Address Translation Mechanism. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 358–365. IEEE, June 1986.

[Wulf 69] W. A. Wulf. Performance Monitors for Multiprogramming Systems. In *Proceedings of the Second ACM Symposium on Operating Systems Principles*, pages 175–181. ACM, 1969.

[Zhang et al. 97] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System Support for Automatic Profiling and Optimization. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 15–26. ACM, December 1997.

Vita

Theodore Haynes Romer was born on October 24th, 1963 in Northampton, Massachusetts. He attended Haverford College in Haverford, Pennsylvania from 1981 to 1983. He worked as a schoolbus driver for the Eanes Independent School District in Westlake Hills, Texas from 1984 to 1988. He attended the University of Texas in Austin, Texas from 1989 to 1991, receiving his B.S. degree with Highest Honors and Special Honors in Computer Sciences in 1991. He attended the University of Washington in Seattle, Washington from 1991 to 1998, receiving his M.S. degree in Computer Science and Engineering in 1995 and his Ph.D. degree in Computer Science and Engineering in 1998.

Although Ted used to *drive* the bus, now he's just a passenger.