The Design and Evaluation of In-Cache Address Translation

David A. Wood

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

February 1990

This research was funded by DARPA contract number N00039-85-C-0269 (SPUR), an NSF Presidential Young Investigator award to Randy H. Katz, and an IBM Predoctoral Fellowship. Additional funding came from the California MICRO program, in conjunction with Texas Instruments, Xerox, Honeywell, and Phillips/Signetics.

The Design and Evaluation of In-Cache Address Translation

by

David A. Wood

Abstract

In this dissertation we study in-cache address translation, a new approach to implementing virtual memory. In-cache translation combines the functions of the traditional translation lookaside buffer with a virtual address cache. Rather than dedicating hardware resources specifically to hold pagetable entries, in-cache translation lets the pagetable share the regular cache with instructions and data. By combining the two mechanisms we simplify the memory system design, and potentially reduce the cycle time. In addition, by eliminating the translation lookaside buffer, we simplify the translation consistency problem in multiprocessors: the operating system can use the regular data cache coherency protocol to maintain a consistent view of the pagetable.

Trace-driven simulation shows that in-cache translation has better performance than many translation lookaside buffer designs. As cache memories grow larger, the advantage of in-cache translation will also increase. Other simulation results indicate that in-cache translation performs well over a wide range of cache configurations.

To further understand in-cache translation, we implemented it as a central feature of SPUR, a multiprocessor workstation developed at the University of California at Berkeley. A five-processor prototype convincingly demonstrates the feasibility of this new mechanism. In addition, a set of event counters, imbedded in the custom VLSI cache controller chip, provides a means to measure cache performance. We use these counters to evaluate the performance of in-cache translation for 23 workloads. These measurements validate some of the key simulation results.

The measurements and simulations also show that the performance of in-cache translation is sensitive to pagetable placement. We propose a variation of the algorithm, called inverted in-cache translation, which reduces this sensitivity. We also examine alternative ways to support reference and dirty bits in a virtual address cache. An analytic model and measurements from the prototype show that the SPUR mechanism has the best performance, but that emulating dirty bits with protection is not much worse and does not require additional hardware. Finally, we show that the miss bit approximation to reference bits, where the bit is only checked on cache misses, performs better than true reference bits, which require a partial cache flush.

For my parents,
Roger Charles Wood
Ann Elizabeth Wilson Wood

Acknowledgments

Graduate school has been one of the best periods of my life. The dynamic environment created by the SPUR project inspired the work presented in this dissertation. I would like to thank professors Dave Patterson, Randy Katz, John Ousterhout, and Dave Hodges for making SPUR possible.

Many people contributed to SPUR and to the ideas studied in this dissertation; George Taylor, Joan Pendelton, Mark Hill, Susan Eggers, Garth Gibson, and Scott Ritchie all played important roles. Scott Ritchie collaborated on the initial analysis of in-cache translation, and Walter Beach did the original layout for the address translation datapath. Susan Eggers, Garth Gibson, and Deog-Kyoon Jeong all made major contributions to the design and implementation of the SPUR cache controller chip. Shing Kong and Daebum Lee designed and built the CPU chip, and Doug Johnson, while visiting from Texas Instruments, helped fill in the holes with his years of experience. Ken Lutz provided the stabilizing force, and real-world experience, that helped make SPUR a working system.

John Ousterhout's Sprite team also provided invaluable assistance. Mike Nelson and Mendel Rosenblum ported Sprite to the SPUR prototype. Mendel, in particular, spent many hours debugging subtle problems, making possible the measurements in this dissertation. John Hartman took over the torch from Mike and Mendel, and continued to keep SPUR running despite numerous operating system changes. Brent Welch has been with the group from the beginning, always ready to answer my operating system questions.

George Taylor, Jim Larus, and Ben Zorn wrote the SPUR Lisp system and the Barb simulator, which I used to generate the SPUR address traces. Dick Sites at Digital Equipment Corporation and Anant Agarwal at Stanford (now at MIT) provided the ATUM traces, and Joe Hull and Rollie Schmidt provided the Synapse traces. Mark Hill wrote DineroIII, which I extended to support in-cache address translation.

I would like to thank my advisor Randy Katz, for 6 years of encouragement, advice, and support; I know he never thought I would take this long to finish. My second reader, John Ousterhout, provided numerous suggestions that improved this dissertation. And I would like to thank Ron Wolff, my third reader, for reading yet another cache dissertation. Several other people read earlier drafts and helped improve the final product; my thanks to Susan Eggers, Garth Gibson, Mark Hill, Corinna Lee, and Jane Doughty.

The social aspects of graduate school were also very important. Mark Hill, Gaetano Borriello, Gregg Foster, Garth Gibson, Jim Larus, Susan Eggers, Corinna Lee, and the rest of the lunch bunch provided a break from the office and a chance to drink Caffe Lattes.

Ken Lutz taught me new appreciation for power tools, and introduced me to the evils of home improvements. Tamaroa, my sailboat, provided frequent escapes from the stresses of school; without her I wonder if I would have kept my sanity.

Finally, I would like to thank my family for all their years of support. My parents raised me to value education; as a second-generation computer scientist, I have grown up assuming that I would earn a doctorate. My wife, Jane Doughty, provided years of encouragement and both emotional and financial support. She put up with my long hours and lost weekends when they were truly necessary, but forced me to keep school in perspective. Without her, graduate school would not have been nearly as enjoyable. And, last, I want to thank my son, Alexander Neil Wood-Doughty, who provided the incentive to finally finish; unlike his father, Alex only slipped his schedule by 3 days.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Oxford English Dictionary defines *architecture* as "the art or science of constructing edifices for human use". Analogous in function to its namesake, *computer architecture* pertains to the design of computer systems for human use. Both fields combine aesthetics and utility: a pleasing form is always desirable as long as it provides the required functionality. Extending this comparison slightly further, one also sees that in both disciplines the practitioner, or architect, must fulfill the needs of the user within the constraints of the current technology. Because the user's requirements and the current technology change over time, however, the architectural traditions and technical designs within these two fields wax and wane; many disappear entirely to perhaps eventually re-emerge in a somewhat different form.

With respect to the architecture of edifices, a primary example of this developmental flow is the shift of the western world from an agrarian society to first an industrial economy, and then the current service economy which brought with it the need for large, high-density office buildings and residences. New technologies have fostered this change: steel girders and reinforced concrete allow the construction of modern high-rises and office buildings.

Though born in the 20th century, computer architecture has also seen substantial changes in requirements and technology during its short history. Once computers existed only in the domain of scientists. Now most new automobiles contain at least one, and many families have one in their home. New applications have been made possible through new technologies; first transistors and then integrated circuits have increased the speed and reduced the cost of computers by many orders of magnitude.

As computer technology and uses evolve, the architect faces continuing challenges: i.e., new cost/performance trade-offs necessitate new ideas and inspire new designs. During this last decade, for example, very large scale integrated circuit technology (VLSI) has increased the speed of logic faster than the speed of memory. This trend makes microprogramming less attractive than hardwired control, leading to reduced-instruction-set computers (RISC). This new approach, although not without its critics, is now widely accepted for high-performance processors.

The changing cost/performance trade-offs also affect memory system design: as processors get faster, it becomes harder for memory to keep up. *Cache memories*, small high-speed buffers that hold the most recently-used data and instructions, provide a way to bridge this gap. Even a small cache services a large fraction of the processor's requests, improving overall system performance. Rapidly increasing memory densities and declining prices make it feasible to have large caches, which can handle most memory requests. Once found only on multi-million dollar mainframes, caches now appear on nearly every new microprocessor.

Memory systems also include virtual memory, one of the most important ideas in computer architecture. Virtual memory provides each program with a large, independent address space, hiding details of the physical memory system and of other programs from the user. Invented nearly 30 years ago, virtual memory has persisted as a feature of almost all modern computer systems.

But changing technology provides new opportunities for efficient and economical implementations of virtual memory. Traditionally, computers have implemented *virtual address translation*, the mapping from an abstract virtual memory to a machine's physical memory, with a special-purpose cache called a *translation lookaside buffer*. But as processors get faster the translation lookaside buffer often limits the cycle time, degrading overall performance. Some computers implement *virtual address caches* to prevent this bottleneck. By accessing the cache with virtual addresses, these systems only access the translation lookaside buffer on cache misses, thereby permitting a faster cycle time.

In this dissertation, we extend virtual address caches one step further, and eliminate the translation lookaside buffer entirely. We investigate a radically new approach to virtual address translation, called *in-cache address translation*, which combines the functions of the traditional translation lookaside buffer with the virtual address cache. By integrating the two mechanisms we improve performance by reducing the cycle time and the total number of cache and translation misses. In addition, we believe that in-cache translation simplifies the memory system design, reducing design and implementation costs.

1.1 Contributions of this Dissertation

Architects sometimes design structures that cannot be built or are not cost effective. Similarly, computer architects occasionally fall into the same trap. To help avoid these problems, we believe in an experimental approach that combines design, analysis, prototype implementation, and evaluation, as illustrated in Figure 1.1. Changes in technology and user requirements inspire new designs, promising new features, better performance, lower cost, or reduced complexity. We analyze these characteristics using analytic models and simulation techniques, as appropriate, helping to refine the designs and understand their behavior. But analytic models and simulations often require simplifying assumptions, which degrade the accuracy of their results. Similarly, without implementation, significant design errors and performance trade-offs may remain hidden. To minimize these problems, we implement research prototypes to prove the correctness of the designs and provide performance

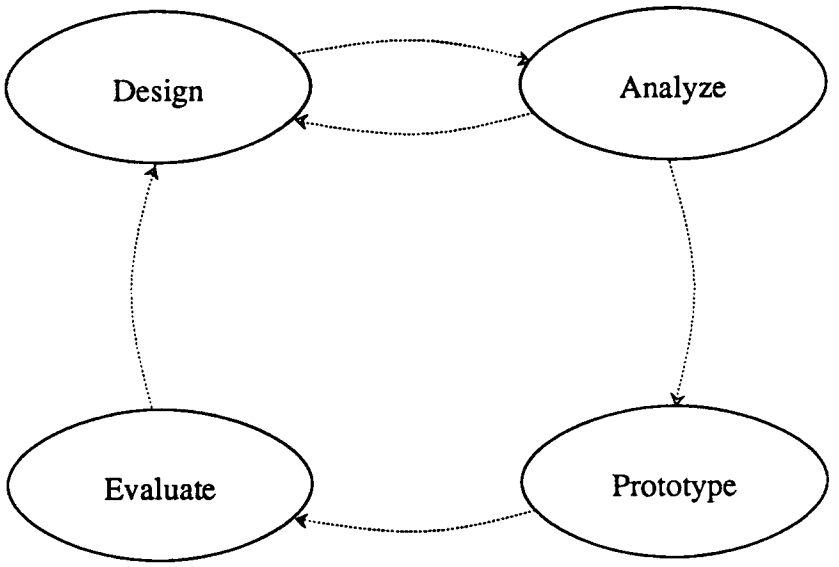


Figure 1.1: Experimental Research Philosophy

test-beds. Once operational, we evaluate a design by measuring the performance of real workloads running on the prototype. Finally, we use the implementation experience and measurement results to further refine the design and propose alternatives. By combining implementation experience with the accuracy of measurements and the flexibility of simulation, we are able to fully understand the advantages and limitations of new architectural ideas.

We have applied this experimental research philosophy to in-cache address translation, and present the results in this dissertation. We analyze the performance of in-cache translation using trace-driven simulation. We use 14 address traces to determine when in-cache translation performs better than traditional translation lookaside buffers. We show that of the 11 translation lookaside buffers we examine, only 2 of the largest have better performance for the SPUR cache configuration, and these are no more than 1% better. In-cache translation performs more than 5% better than the worst of the buffers. These figures assume the cache access time is constant, however, virtual address caches often have faster access times. When we factor in this implementation-dependent factor, in-cache translation has the potential for significantly better performance than many translation lookaside buffers.

We test the sensitivity of these results to the SPUR cache parameters. We find that in-cache translation has superior performance when the cache contains 64 kilobytes or more (assuming 32-byte blocks, direct-mapped). For smaller caches, the interference between

translation information and the processor's instructions and data degrades the performance of in-cache translation below that of typical translation lookaside buffers. We also show that the performance is generally insensitive to other cache parameters such as block size, associativity, page size, and miss penalty. Thus in-cache translation performs better than translation lookaside buffers over a wide range of cache organizations.

To further understand this new mechanism, we implemented it as a central feature of SPUR, a multiprocessor workstation developed at the University of California at Berkeley. This implementation convincingly demonstrates that in-cache translation works: a five-processor system currently provides a test-bed for multiprocessor operating system research. We show that the implementation is simple, requiring only limited chip resources, and discuss several important performance trade-offs discovered during the prototyping phase.

To permit a thorough evaluation of in-cache translation's performance, we included a set of imbedded performance counters in the SPUR cache controller chip. Using these counters, we measure the performance of the cache and translation algorithm for a large set of important workloads. The results concur with the simulation results, with a few interesting exceptions. One difference illustrates the sensitivity of in-cache translation's performance to the placement of the data structure that holds the mapping information. In addition, this data structure, an array, inefficiently uses memory for sparsely populated address spaces. We show that replacing the data structure with a hashtable solves these problems. This variation, called *inverted in-cache translation*, performs better than the original scheme in many cases, and no more than 1% worse in all cases.

Finally, we examine some of the supporting issues related to in-cache translation and virtual address caches. Most systems support reference and dirty bits to help the operating system optimize its utilization of physical memory. These status bits are traditionally stored in the translation lookaside buffer along with the mapping information. But since in-cache translation eliminates the translation lookaside buffer, we must use a different approach. We examine several alternative ways to maintain dirty bits, including a new scheme that we implemented in the SPUR workstation. We use an analytic model and measurements from the SPUR prototype to compare the performance of the alternatives. We show that the SPUR approach has the best performance, but that emulating dirty bits with protection is not much worse and requires no additional hardware. We also show that the *miss bit approximation* to reference bits, where the bit is only checked on cache misses, performs better than true reference bits, which require a partial cache flush when the bit is cleared. Both these latter results apply to all virtual address cache designs, rather than just in-cache translation.

In summary, in-cache translation is a new approach to virtual address translation, which combines the functions of the translation lookaside buffer with the virtual address cache. By integrating the two mechanisms we improve performance by reducing the cycle time and the total number of cache and translation misses. In addition, we believe that eliminating a separate translation lookaside buffer simplifies the memory system design, reducing both the design and implementation costs. Our results convincingly show that in-cache translation

is an attractive alternative to translation lookaside buffers over a large region of the design space.

1.2 Dissertation Outline

The outline of this dissertation follows the experimental research process described in the previous section. Except for Chapter 2, which provides background material, each chapter encompasses one or more research phases, as illustrated in Figure 1.2.

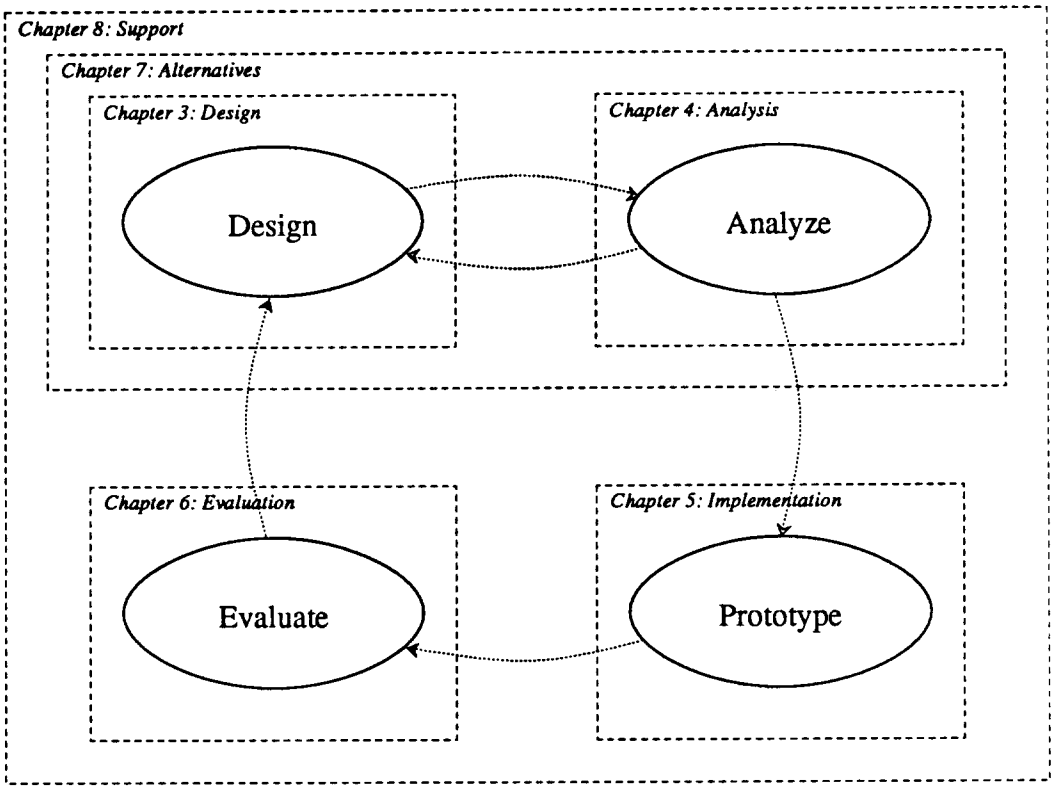


Figure 1.2: Dissertation Outline

The next chapter provides background and terminology used throughout the dissertation. We discuss the primary issues in cache and virtual memory design, and lay the groundwork for the design of in-cache translation. In addition, this chapter contains a brief summary of related research.

In Chapter 3, we examine alternative designs for virtual address translation. We show the evolution of cache design that inspired in-cache translation, and describe the translation algorithm in general terms. Then we describe several important design decisions, followed

by the specific realization of in-cache translation used in SPUR, which differs slightly from the general form of the algorithm.

In Chapter 4, we analyze the performance of in-cache translation using trace-driven simulation. The baseline analysis examines the performance of in-cache translation for the SPUR cache parameters, and compares these results to simulations of 11 translation lookaside buffers. Then we test the sensitivity of these results by varying the cache parameters over a wide range. We examine the effect of cache size, block size, associativity, page size, and miss penalty on the performance of in-cache translation.

In Chapter 5, we describe the implementation of in-cache translation in the SPUR prototype. We examine the implementation trade-offs, and discuss optimizations that could improve performance in a more aggressive implementation. This chapter also describes the set of built-in counters that enable performance measurements of the memory system for running programs.

In Chapter 6, we evaluate the performance of in-cache translation on the SPUR prototype, using the imbedded performance counters. We demonstrate the power of this evaluation methodology, but also show that careful planning is not always sufficient. In our analysis, we discovered that two important events were excluded from the original set of counters. A small amount of additional hardware eliminated the problem, permitting us to measure the performance accurately.

In Chapter 7, we consider a variation of in-cache translation that solves two problems with the original algorithm. We describe the new algorithm and compare its implementation complexity to the original scheme. We compare the performance of the two schemes using trace-driven simulation. In Chapter 8, we examine the problem of supporting reference and dirty bits in a virtual address cache. We use an analytic model and measurements from the SPUR prototype to evaluate the alternative approaches. Finally, we summarize our work in Chapter 9, and suggest areas for future research.

Chapter 2

Preliminaries

In this chapter, we introduce background material and terminology used throughout the rest of the dissertation. We summarize the basic issues in cache design, and review the fundamental concepts of virtual memory and address translation. We briefly describe the SPUR project, which provided a framework for this research, then survey the most relevant previous works. Readers familiar with the field may wish to skip directly to Chapter 3.

2.1 Cache Design Basics

A *cache memory* is a high-speed associative memory that holds recently used instructions and data. When the processor requests a data (or instruction) word, which is called a *reference*, the hardware checks the cache. If the requested data resides in the cache, the reference *hits*, otherwise the reference *misses* and a hardware controller fetches the data from memory, and stores it in the cache possibly replacing some other data. The controller determines hits and misses by comparing the requested address to an *address tag* associated with the data word.

Several parameters affect the probability that a reference misses. The *cache size* specifies how many bytes of instructions and data are contained within the cache. The *block size* specifies how many bytes of instructions and data are associated with each address tag; a block is also the unit of data fetched on a cache miss[1].

To limit cost and complexity, we usually partition the cache into fixed size *sets*. Part of the address, called the *index*, selects which set may contain a particular memory block, as illustrated in Figure 2.1. Traditionally, both the cache index and address tag are extracted from the physical address. However, as we discuss in Chapter 3, the trend towards faster processors makes *virtual address caches*, which form the index and tag from the virtual address, more attractive. The number of blocks in a set is called the *degree of associativity*.

[1] A block may be divided into sub-blocks, or sectors, but we do not consider that generalization in this dissertation.

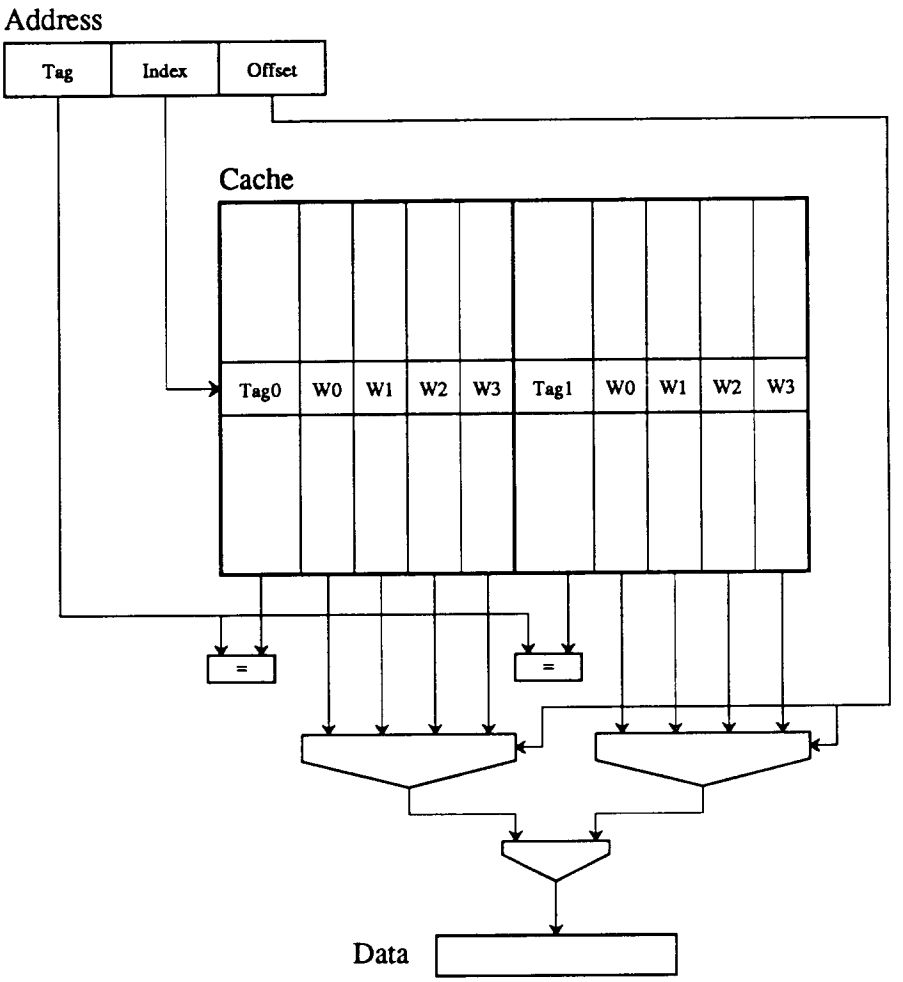


Figure 2.1: Basic Cache Organization

This figure illustrates a 2-way set-associative cache. The *index* selects the *set*, which contains two *blocks*. The *address tag* from the requested address is compared with the tags stored with each block. The *offset* selects the desired word from the block.

When the cache has only one set, we call it *fully-associative.* When each set contains at least two blocks, we call it *set-associative.* When each set contains only one block, we call it *direct-mapped.* These three parameters: cache size, block size, and associativity specify the configuration of the cache.

Several policy decisions affect the cost and performance of the cache. For example, on processor writes the data can either be written directly to memory (*write-through*) or written into the cache (*write-back*). We assume a write-back policy because it generally provides higher performance. Similarly, when we make room in the cache for a new block, we replace the least-recently-used block of the selected set. Finally, we assume that the cache only fetches blocks requested by the processor (demand fetch), rather than fetching them in advance (prefetch).

2.2 Virtual Memory Basics

Virtual memory provides the programmer with the illusion of a single large contiguous address space. The operating system supports this abstraction by shuffling data between physical memory and secondary storage. *Address translation* is the process of mapping the virtual address space provided to the programmer to the physical address space supported by the memory system. This process is generally broken into two parts: the translation algorithm and the hardware acceleration needed to improve performance.

In the most common translation algorithms, the virtual address space and physical memory are broken into fixed-size units called *pages.* A data structure called the *pagetable* contains an entry for each active virtual page. A *pagetable entry (PTE)* contains the address of the physical page that currently holds the virtual page. Figure 2.2 illustrates the translation process.

In most systems, the pagetable is conceptually an array. However, because the virtual address space is generally very large (2^{32} bytes), it is impractical to allocate the entire array in physical memory. Instead, most systems implement a sparse array using a *multi-level pagetable* data structure, similar to the well-known B$\star$-tree except that each node in the tree is an array. We descend the tree from the root to the leaves, using bits from the virtual address to index into each node. A valid bit indicates whether an entry in the node points to another page of pagetable entries.

Since all real systems have a pagetable with at least two levels, the translation algorithm requires an additional two references for each processor reference. Since such high overhead is impractical, most systems accelerate the performance by using a special-purpose cache called a *translation lookaside buffer (TLB),* which holds pagetable entries rather than instructions or data.

A translation lookaside buffer is accessed with only the virtual page number, the high-order bits of the virtual address, rather than the entire address like a regular data cache. If we find the pagetable entry in the translation lookaside buffer, then we avoid accessing the

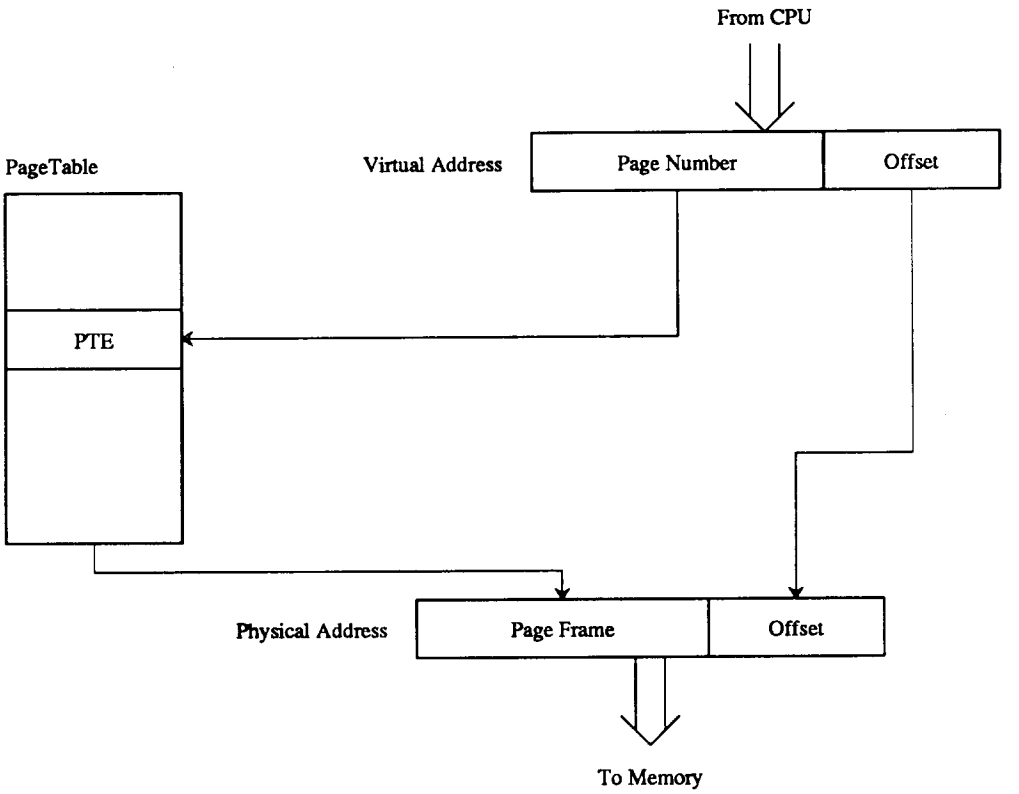


Figure 2.2: Conceptual Address Translation Mapping

The high-order bits of the virtual address form the virtual page number. Conceptually, the pagetable is an array indexed with the virtual page number. Each pagetable entry contains a physical page number and several status bits.

full pagetable in main memory. Since the translation lookaside buffer is much faster than memory, it greatly improves the performance of address translation.

A pagetable entry (and a translation lookaside buffer entry) contains several status flags as well as the physical address of the page. One flag indicates whether the virtual page is *valid*, i.e., stored in physical memory. The *protection* bits specify which types of references (reads, writes, and instruction fetches) are permitted to the page. Usually the system supports two other flags, called the *reference* and *dirty bits*, which indicate whether a page has been referenced or modified, respectively, while in memory. The operating system uses these bits to help determine which pages are actively used and should thus be retained in memory.

2.3 Performance Analysis Basics

In performance analysis, we want to determine how adding a particular feature, or changing the configuration of a cache, affects the performance of a target workload running on a target system. The most accurate evaluation technique is to build the target system and measure its performance under the target workload. However, since we obviously cannot afford to implement every idea, we often use models and simulations to estimate the performance. Analytic models, such as mean-value analysis and Markov chains, often have closed-form solutions and are the fastest and easiest techniques to use. Unfortunately, to achieve a model with a tractable solution usually requires making assumptions that affect the accuracy of the results. For models without a closed form, we can use distribution-driven simulation to estimate the solution. While this approach requires fewer assumptions, we still must assume that the input is accurately characterized by a tractable distribution.

Trace-driven simulation is the most common technique for studying cache performance. In this approach, we record a snap-shot of a processor's execution in the form of an address trace. This trace contains a representation of the actual execution of a real program. The cache simulator uses this trace as input, thus the simulation accurately measures the cache's performance for the workload captured by the trace (we discuss the limitations of trace-driven simulation in Chapter 4).

The performance of cache and translation lookaside buffer organizations is traditionally evaluated using the *miss ratio*, defined as the number of references that miss divided by the total number of references. This metric is independent of implementation details, and therefore permits an abstract comparison between alternatives.

The *effective access time* accounts for the difference in time required to handle cache hits and misses. It is defined as the total number of cycles needed to execute a program (trace) divided by the total number of references. Because it models the number of cycles needed for each reference, the effective access time more accurately characterizes performance. The additional accuracy comes at the expense of generality, since this metric requires additional assumptions about the implementation. We discuss these metrics further in Chapter 4.

2.4 SPUR: A Context for this Research

We conducted this research as part of the SPUR multiprocessor workstation project at the University of California at Berkeley[40]. The SPUR workstation, illustrated in Figure 2.3, is a shared-memory multiprocessor computer developed as an experiment in parallel processing. Up to 12 processors communicate with main memory over a common bus. Large direct-mapped caches, 128 kilobytes with 32-byte blocks, reduce each processor's demand on memory. The *Berkeley Ownership* coherency protocol[49] maintains a consistent image across all the caches by invalidating read-only blocks when necessary.

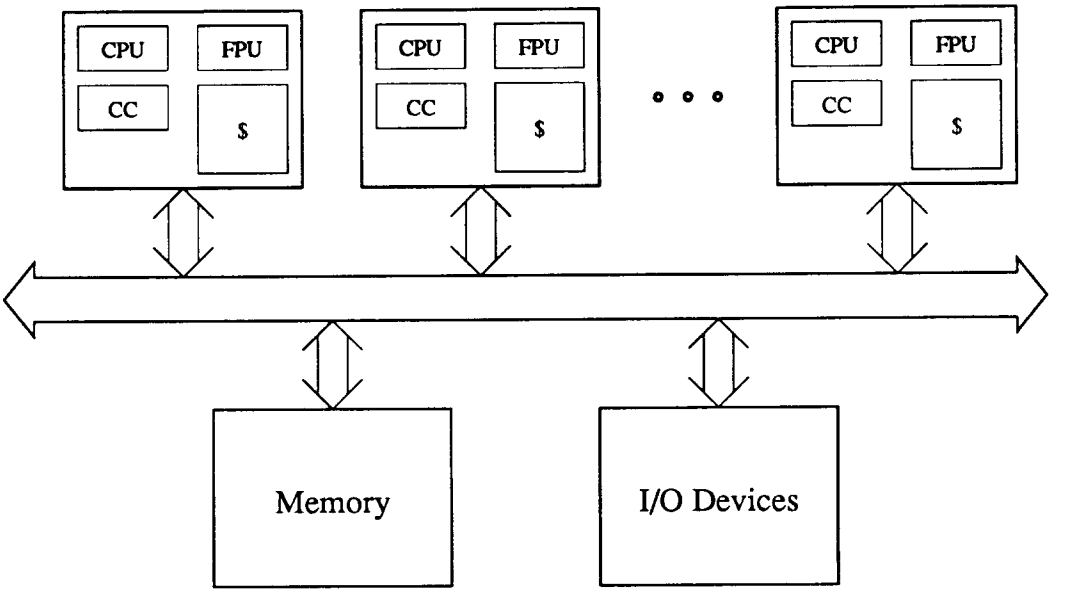


Figure 2.3: Organization of SPUR Workstation

Each processor contains three custom VLSI chips: a central processing unit (CPU), a floating point unit (FPU), and a cache controller (CC). The CPU chip is a RISC-like processor with 32-bit instructions and 8 register windows. Tagged data and special instructions provide efficient support for Lisp programs. The floating point unit implements the IEEE standard, including extended precision, without using microcode. The cache controller chip manages the cache and memory system, implementing the Berkeley Ownership protocol and in-cache address translation. We describe more details of the SPUR implementation in Chapter 5.

The goals and constraints of the SPUR workstation inspired the design of in-cache translation. In addition, the prototype design provided an opportunity to implement and experiment with the algorithm. Now that we have working SPUR machines running the Sprite network operating system[62], we can also measure the performance of in-cache translation for a large set of real programs.

2.5 History and Related Work

Both virtual memory and caches have existed for many years and have been the subject of many publications. In this section, we review the history of these innovations and survey the previous publications that most closely pertain to our work. First, we review virtual memory and address translation, then cache memories, and finally translation consistency.

Virtual memory first appeared in 1961, implemented in Atlas, a system developed jointly by the University of Manchester and the Ferranti Corporation[29]. Its one-level store presented programs with a virtual memory independent of the machine's underlying hardware[50]. It divided the 16-kilobyte main memory into 32 page frames, each with a *Page Address Register* containing the current virtual page number.

The Multics project at MIT generalized the pure paging of Atlas into the now familiar paged segments[11], directly affecting the design of the GE-645 [36] and the IBM 360/67[10]. The pioneering work of the Multics project, introducing major concepts such as protection domains, shared memory, and multi-level pagetables, has affected all subsequent virtual memory research.

Virtual memory is a fundamental architectural concept, with important performance ramifications. Most of the research has focussed on page replacement strategies: a 1978 bibliography included over 300 entries[75], and many more papers have been written since. Denning's classic survey[23] and subsequent retrospective[24] provide excellent background reading.

In this dissertation, our interest in virtual memory focuses primarily on address translation and the hardware needed to accelerate its performance. Atlas placed its acceleration hardware close to the memory, providing a fully-associative lookup of the virtual address with one content-addressable memory word per page frame. Variations on this approach have been proposed several times[53, 86, 87, 52, 51, 84], but have never been widely accepted. In Atlas, implemented using discrete transistors and core memory, the content-addressable memories were inexpensive relative to main memory. But because content-addressable memories have not scaled as rapidly as MOS dynamic RAM, this technique has not proven economically viable for modern systems.

Instead, most systems place their acceleration hardware close to the processor. The GE-645 introduced an associative memory, now commonly called a translation lookaside buffer, to hold the most recently used translation entries[36]. Nearly all modern systems that support virtual memory use a translation lookaside buffer[32]. We discuss translation lookaside buffers more fully in Chapter 3.

Traditionally, translation lookaside buffers have been managed using hardware or microcode. But the success of reduced instruction set computers (RISC), which only implement performance-critical functions in hardware and leave infrequently-used functions to software, has prompted a re-evaluation of this design decision. The MIPS R2000[48], AMD 29000[1], and Regulus[73] architectures all handle translation lookaside buffer misses in software. In addition to handling misses in software, reference and dirty bits must also be

supported in software, using the protection bits. Berkeley UNIX pioneered this approach, emulating reference bits on the VAX 11/780[8, 9].

The performance of translation lookaside buffers has not been widely studied. Schroeder studied the performance of the GE-645's associative memory, using a single frequency counter to measure live workloads[72]. More recently, there have been several studies that examine translation lookaside buffer performance in implementations of the VAX architecture. In the first, Satyanarayanan and Bhandarkar used trace-driven simulation with user-mode address traces to examine the effects of buffer size and associativity[71]. In the second, Clark and Emer use trace-driven simulation and hardware measurements to study the performance of the VAX 11/780 translation buffer[20]. Among their results, they show that the translation lookaside buffer accounts for 4% of the execution time. Finally, in a measurement study of the VAX 8800, Clark, Bannon, and Keller show that 6% of all cycles are consumed by the translation lookaside buffer[19]. In another recent study, Alexander, Keshlear, and Briggs used traces obtained with a hardware monitor to study the performance of different buffer configurations[5]. All of these performance studies focus on conventional designs, rather than innovative approaches like in-cache translation.

Cache memories have appeared in most high-performance systems since the IBM System/360 Model 85 in 1968[10]. Because of their potential to improve system performance, the design of cache memories has been a fertile source of research. A recent bibliography[79] includes over 400 entries, on topics ranging from optimal block size to cache coherency protocols. Smith's survey of cache design is the most comprehensive paper[77], serving as excellent background for this dissertation. Rather than summarize the entire field, we discuss only the recent results that have direct bearing on this dissertation.

Agarwal has examined the effects of operating systems and multiprogramming using trace-driven simulation[4, 3, 2]. He quantifies earlier observations by other researchers that operating systems have worse cache performance than most application programs, and that multiprogramming effects severely degrade performance. In addition, he has developed a new address trace generation technique, which modifies a machine's microcode, and gathered a number of high quality traces. These traces are now widely available and we use them in this dissertation.

Hill has examined the effect of set-associativity on cache performance. He shows that increasing associativity also increases the cache access time, which makes direct-mapped caches perform better in many cases, despite having higher miss ratios[42, 41]. Przybylski generalizes these results, showing that cycle time should always be considered when determining an optimal cache configuration[67, 66]. In addition to confirming Hill's results on associativity, he shows that increasing the cache size may improve performance even if it degrades the cache access time.

The rapid increases in processor speed have made virtual address caches more popular in recent years; they now appear in a number of commercial machines[83, 27, 55]. However, because virtual address caches make sharing data more difficult, discussed at length in Chapter 3, they have appeared primarily in research machines. The Manchester MU6-G

employed a virtual address cache, using a segmentation scheme that allows sharing of system segments[54, 87, 86]. The MU6-G does not support sharing between user address spaces, and performs address translation at main memory, as did Atlas. Knapp and Baer propose a similar approach, also placing translation at main memory[51, 52]. Their system maps a *program virtual address* to a global *system virtual address*; programs share data by using the same system virtual address.

The Xerox Dragon's virtual address cache stores both the virtual and physical address tags in content addressable memory[57]. The CPU accesses the virtual address tags on each reference; the physical tags are checked on cache misses to resolve synonyms. The physical tags are also used to maintain cache consistency across all the processors. Together, the two sets of tags serves as a small translation lookaside buffer. Unfortunately, this design relies on content-addressable memory, which does not scale well to large caches.

Goodman proposed a similar design using set-associative caches[37]. The virtual and physical tags contain small pointers to cross-reference corresponding entries. In general, this approach does not allow full utilization of the cache, since the two sets of tags are indexed with different bits, causing spurious conflicts. Wang, Baer, and Levy extend this idea to a cache hierarchy: with a first-level virtual cache, a second-level physical cache, and cross-reference pointers in both[90]. Simulation results indicate excellent performance.

Most of these systems use a separate translation lookaside buffer to produce the physical address on a cache miss. The Regulus processor reserves a portion of its cache for pagetable entries[73]. On a cache miss, the hardware looks for the appropriate pagetable entry in the translation region, trapping to software on a cache miss[56].

The VMP system adds an additional twist to virtual address caches, by handling misses in software[17]. The hardware handles cache hits, generating faults on cache misses. A local memory provides code and data for the miss handler. However, performance measurements show that even with a cache of 256 kilobytes with 128 byte blocks, misses occur frequently enough that the 200-300 cycle miss penalty significantly degrades performance[16]. Instead, they propose implementing "simple" cache misses in hardware.

Shared-memory multiprocessors suffer from the well-known cache consistency problem. Since a translation lookaside buffer is just a special-purpose cache, a multiprocessor which uses one per processor suffers from a *translation lookaside buffer consistency problem.* When one process changes a pagetable entry, it must take some action to insure a consistent image across the entire system.

Several research systems have tried to eliminate this problem by maintaining only a single copy of each pagetable entry. The Xerox Dragon[57] and MIPS-X-MP[82] systems have proposed using centralized translation units that are only accessed on cache misses. However, this shared resource quickly becomes a bottleneck[84]. Teller proposes that address translation be shifted to the memory controller, similar to Atlas, the MU6-G, and Knapp's proposed machine, discussed above. However, this approach requires associative memories larger and faster than commercially available.

The most popular solution, implemented in several commercial systems[31, 26, 88], has

been to use interprocessor interrupts and software to resolve inconsistencies. In general, these algorithms require the initiating processor to interrupt the other processors, causing them to invalidate or update their translation lookaside buffers.

Thompson, et al., describe an algorithm for a multiprocessor based on the MIPS R2000[88]. The algorithm uses *lazy devaluation* to reduce overhead by deferring consistency operations as long as possible.

Rashid, et al., propose a more general algorithm for the Mach multiprocessor operating system[68], which runs on many different machines. The *TLB shootdown algorithm* maintains translation consistency by interrupting only those processors that have had access to the pagetable entries being changed[12]. They use measurements from a 15 processor system to show that the overhead is usually less than 1% for a system of this size, and scales linearly with the number of processors.

Tzou uses an analytic model and event-driven simulation to study the performance of three algorithms, including the MACH shootdown algorithm[89]. He finds that these algorithms do not scale well to many processors and message-passing operating systems. Instead, he proposes operating system mechanisms to tolerate translation inconsistency. These mechanisms have been implemented in the DASH experimental operating system[6].

2.6 Summary

This chapter introduced background information used throughout the dissertation. We described the basic issues of cache design, virtual memory design, and performance analysis, defining the terminology used in later chapters. The SPUR workstation provided a framework our research. Finally, we reviewed the history of caches and virtual memory, and summarized the recent publications that most closely relate to our work.

Chapter 3

Design

In this chapter, we describe the central focus of the dissertation: the in-cache address translation mechanism. This novel approach evolved during the design of the SPUR workstation, in an attempt to simplify both hardware and software complexity. Beginning with the first section, we motivate the design by examining alternative ways to combine address translation and caches in the same system. The advantages and disadvantages of each approach are qualitatively compared, with examples from commercial machines and research prototypes. This discussion leads to a key observation: that we can combine the functions of the translation lookaside buffer with a virtual address cache. We argue in the second section that integrating the functions into a single unit can improve performance and decrease hardware cost and complexity.

The third section considers the general case of the translation algorithm. By mapping the pagetable into the virtual address space, we can keep pagetable entries in the virtual address cache. On a cache miss, the controller uses a simple address calculation to generate the virtual address of the pagetable entry. With the address, the controller accesses the cache as if it were a translation lookaside buffer.

Section 3.4 discusses the problems with using in-cache translation in a writeback cache. Because the translation algorithm brings pagetable entries into the cache, a naive implementation can cause deadlock. We discuss alternative solutions that prevent this problem.

Next, we discuss the translation consistency problem. Since a translation lookaside buffer is simply a special-purpose cache, multiprocessor computers that have a translation lookaside buffer per processor have an additional cache consistency problem. We argue that in-cache translation simplifies this problem by letting the operating system maintain translation consistency using the regular cache coherency mechanism.

Finally, we describe the SPUR realization of in-cache translation, which differs slightly from the general algorithm. SPUR uses a global virtual address space to prevent synonyms, defined in the next section. The mapping from the process virtual space to the global virtual space permits an optimization in the translation algorithm.

3.1 Placement of Address Translation

In computers with virtual memory, address translation occurs between the processor and main memory. When we introduce a cache into the memory hierarchy, we must decide where translation occurs relative to the cache. As illustrated in Figure 3.1, we can perform translation at a number of alternative positions: before the cache, in parallel with the cache, after the cache, or even at the memory controller. Each approach has a number of advantages and disadvantages, which we examine in turn.

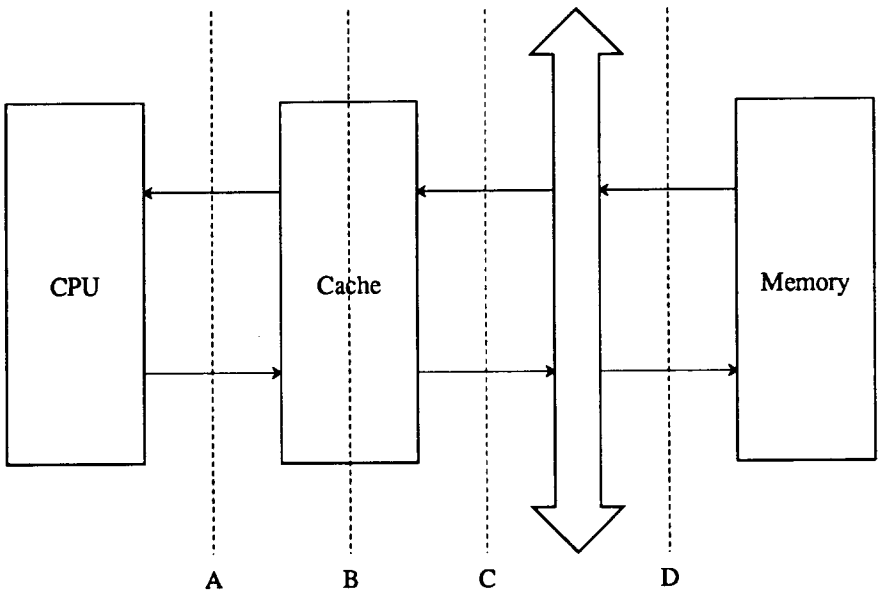


Figure 3.1: Alternative Address Translation Placements

This figure illustrates the possible locations for address translation in a bus based system. The most common placements are (A) and (B), either before the cache or in parallel with it. Some systems place translation behind the cache, at position (C). Finally, translation can be performed at the memory controller, as shown by (D).

3.1.1 Sequential Address Translation

The simplest scheme, used in many commercial machines, translates a virtual address to a physical address before accessing the cache. Figure 3.2 illustrates this approach, referred to as *sequential address translation*. When the processor issues the virtual address, the translation lookaside buffer performs the mapping to the corresponding physical address. The cache derives both its index and tag from this translated address.

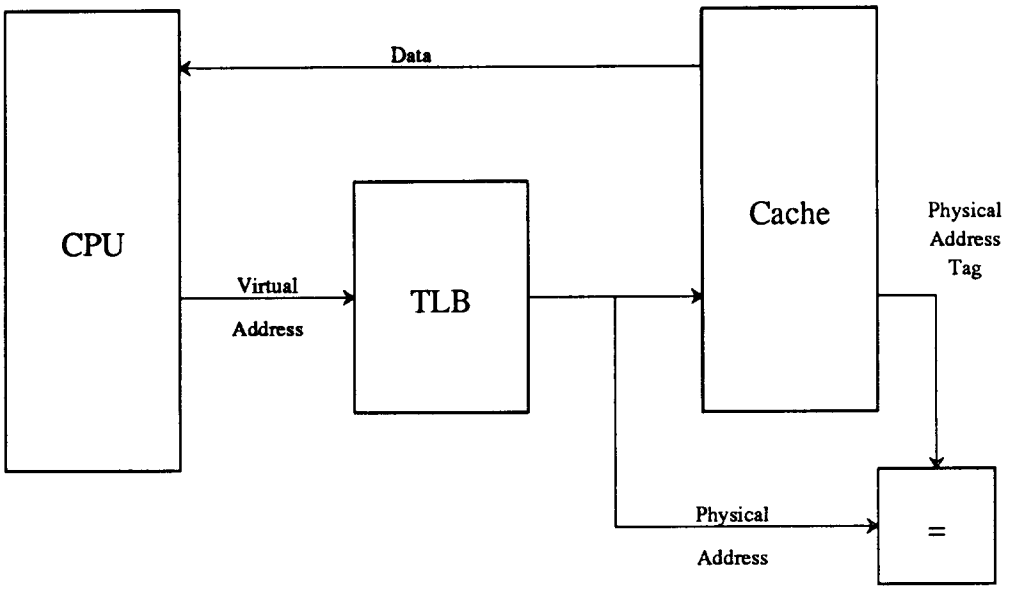


Figure 3.2: Sequential Address Translation

This figure illustrates sequential address translation, where the translation lookaside buffer is placed before the cache. The virtual address is translated to a physical address before beginning the cache access; this allows both the cache index and tag to come from the physical address.

The sequential operation of the cache and translation lookaside buffer leads to this design's simplicity. However, it also causes its greatest problem. Since each cache access requires translation, the minimum cache access time requires two RAM reads: first for translation, and then for the cache. Since the cache access time often determines a processor's critical path, especially for RISC processors, sequential address translation limits the processor's cycle time.

Pipelining the cache access, by performing translation in one stage and accessing the cache in the next, eliminates this problem. However, introducing an additional pipeline stage has its own performance problems. The additional stage increases the load latency, which in turn increases the frequency and duration of pipeline interlocks (or increases the number of delay slots for systems without hardware interlocks). An additional stage may also increase the branch latency, the number of potentially wasted cycles whenever the processor changes its flow of control. Pipelining the cache access minimizes the cycle time at the expense of additional cycles.

In some pipelines, it may be possible to merge translation into an earlier pipe stage. For example, the MIPS R2000 integrates the translation lookaside buffer onto the processor chip and allocates half a pipeline stage for translation[48]. This leaves an entire pipeline stage to access the cache RAMs. However, the translation lookaside buffer was the most difficult part of the design, and required complex self-timed circuits to achieve the desired performance[22]. Translation for the instruction stream requires a separate "miniTLB", containing only two entries, because a full translation would significantly increase the cycle time. Despite the complexity, design effort, and fancy circuitry, the translation lookaside buffer still limits the cycle time of this chip.

3.1.2 Parallel Address Translation

High-performance machines have traditionally eliminated the performance problems of sequential translation by performing the translation lookaside buffer lookup in parallel with the cache. Figure 3.3 illustrates this scheme, called *parallel address translation*. The processor sends the virtual address to both the cache and translation unit, which then operate in parallel. The cache extracts the index from the virtual address, and uses it to read from the selected set. In parallel with the cache access, the translation lookaside buffer performs translation and generates the physical address. We detect cache hits by comparing the physical address tags stored with each cache line with the physical address from the translation lookaside buffer.

Since we access the translation lookaside buffer in parallel with the cache tag access, the critical path contains only a single RAM access. For small caches, parallel translation is simple to implement: we merely use the virtual address to form the cache index. However, large caches must solve the virtual address synonym problem.

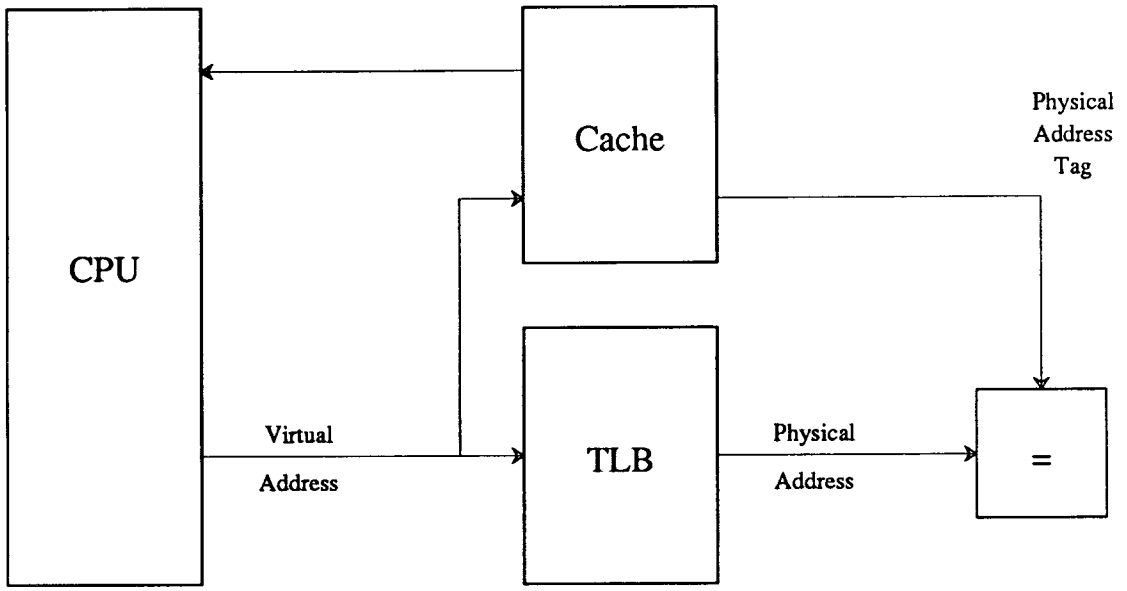


Figure 3.3: Parallel Address Translation

This figure illustrates parallel address translation, where the translation lookaside buffer is placed in parallel with the cache. The virtual address is sent to both the translation lookaside buffer and the cache, allowing parallel access. This organization permits a faster cycle time than sequential translation.

Virtual Address Synonyms

A virtual address synonym, or alias, exists when two virtual addresses map to the same physical address [77], as illustrated in Figure 3.4 (a). Synonyms generally arise when different processes share code or data, for example, through the use of memory-mapped files or shared memory. In caches indexed with virtual addresses, a problem arises when two synonyms for the same physical address index into different cache lines, as illustrated in Figure 3.4 (b). If we do not take action, then two copies of the same data reside in different locations in the cache simultaneously. If the processor then modifies one of these synonyms, the two copies become inconsistent, potentially causing incorrect program behavior.

Clearly, the cache must prevent or detect this multiple-copy problem. The traditional solution, used in many mainframe computers, restricts the cache configuration so that a block always maps to the same cache line, regardless of its virtual address. We can guarantee this only if the cache index is the same in both the virtual and physical addresses, which is the case if the page size times the associativity of the cache is not less than the cache size. But for large caches to meet this restriction they must generally have a high degree of associativity. For example, a 128-kilobyte cache with 4-kilobyte pages requires the cache to be 32-way set-associative. Unfortunately, 32-way set-associative caches are too expensive for most mainframe computers, much less for multiprocessor workstations like SPUR.

To reduce the associativity, some systems change other parameters. The DEC MultiTitan increased the page size to 64 kilobytes, allowing parallel translation with a direct-mapped cache of the same size [47]. The data portion of the IBM 3081's 64-kilobyte cache is only 4-way set-associative, but the address tags are 16-way set-associative[38]. On each access, the controller guesses which bank contains the data: if the controller guesses correctly the reference completes in a single cycle, if it guesses wrong, an additional penalty occurs to change banks. Yet another alternative is to restrict the placement of pages in physical memory, increasing the number of bits that are the same between the virtual and physical memories[76]. In a similar scheme, the Sun-3 and Sun-4 architectures[15, 83] limit the cache to be direct-mapped and at most 128 kilobytes. The operating system restricts virtual address synonyms to be the same modulo 128 kilobytes. This guarantees that synonyms always map to the same cache set[1]. Unfortunately, none of these solutions is completely desirable: large pages decrease the utilization of physical memory, the IBM reprobing scheme is still too expensive for a workstation, the set-associative memory scheme increases the paging overhead, and the Sun approach limits the associativity of the cache.

Another class of solutions calls for detecting when a data block already resides in the cache under another address. One approach uses an analog of the translation lookaside buffer to provide the reverse translation, appropriately named a *reverse translation buffer*[77]. On a cache miss, we use the reverse translation buffer to check whether the data already resides in the cache; if so, we move it to the new location. Goodman proposes a specific variation of this scheme that combines the cache coherency functions with the

[1]The cache must be direct-mapped because it is virtually addressed, as discussed below.

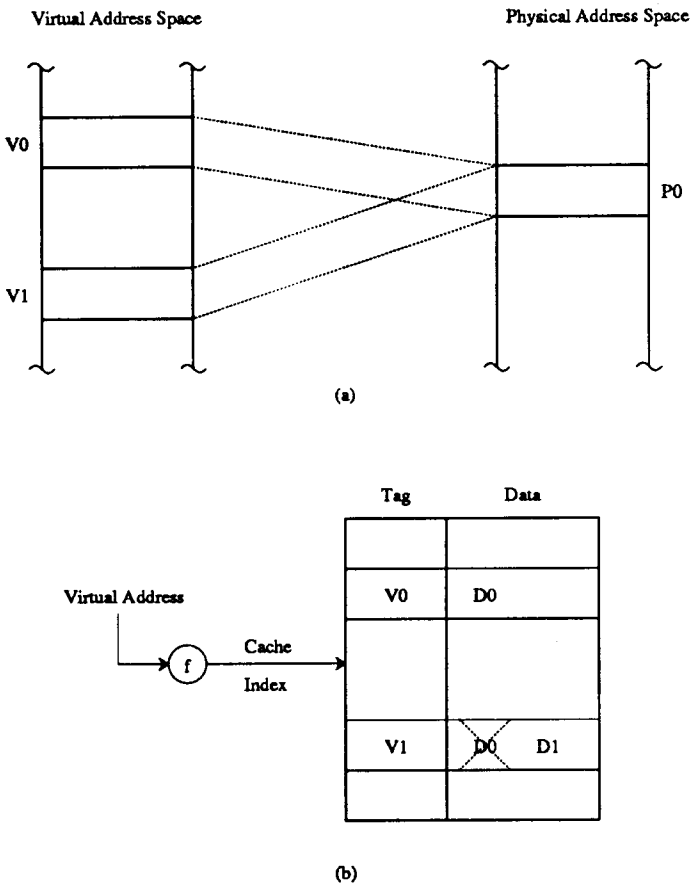


Figure 3.4: Virtual Address Synonyms

The upper figure illustrates a virtual address synonym, or alias, which exists when the operating system allows two virtual addresses to map to the same physical address. This can cause a problem in a cache indexed with virtual addresses because the two synonyms may map to different cache lines, as the lower figure illustrates. If two copies of the data are allowed to reside in the cache simultaneously, then the two copies may become inconsistent, leading to erroneous results.

reverse translation capability[37].

Another solution requires each physical page to have only a single virtual address: eliminating synonyms and their problems[51]. The operating system must enforce this restriction, since the hardware does not otherwise guarantee correct results. Depending upon the hardware support, this approach may either restrict the way programs share data, or impose a significant performance penalty for unrestricted sharing.

Both of the last two solutions eliminate all restrictions on the cache configuration, making them attractive choices for workstation designs. Regardless of the specific solution, once synonyms are dealt with, address translation can occur in parallel with the cache access, without contributing to the critical path. For this reason, parallel address translation is used in many high-performance machines.

3.1.3 Virtual Address Cache

But once we have solved the synonym problem, we can consider a third placement for the translation lookaside buffer. This approach, called a virtual address cache, places the buffer behind the cache, as illustrated in Figure 3.5. The processor sends the virtual address to the cache, which forms both the index and the tag directly from the virtual address. The address tag stored with each cache block is derived from the virtual address, rather than the physical address as in the other schemes. The cache detects hits by comparing the virtual address to the virtual address tag. The translation lookaside buffer performs its function on cache misses, when the physical address is needed to access main memory.

Virtual address caches are as fast as physical address caches that use parallel translation, since in both cases only a single RAM access lies on the critical path. But with a virtual address cache, we only access the translation lookaside buffer on cache misses, so we can afford to build it from slower, less expensive components. Similarly, a high translation lookaside buffer miss ratio has less effect on performance. Thus a virtual address cache makes the design of the translation lookaside buffer much less important. Virtual address caches have been implemented in many machines, particularly in workstations like the Sun-3, Sun-4 and Apollo DN4000.

3.1.4 Translation at the Memory Controller

All of the preceding translation schemes use a translation lookaside buffer at each processor. But since translation lookaside buffers are just special-purpose caches, they pose an additional coherency problem when used in a multiprocessor. For example, when the operating system invalidates a page, it must potentially invalidate an entry in every translation lookaside buffer. This invalidation requires either special hardware support or complex software that interrupts each of the processors[12].

An alternative is to centralize the translation information at the memory controller. In this scheme, processors send virtual addresses across the bus, rather than physical addresses.

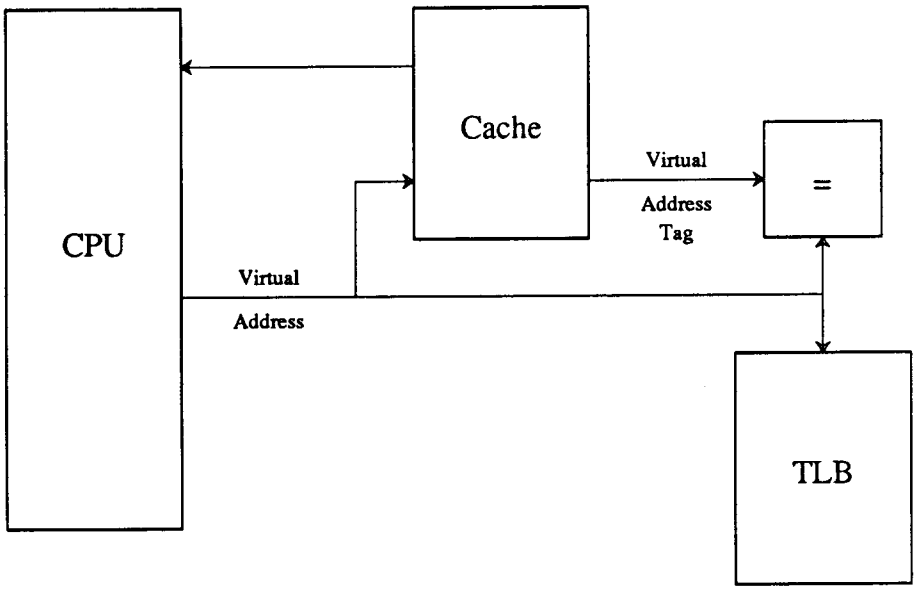


Figure 3.5: Virtual Address Cache

This figure illustrates a virtual address cache, which creates both the cache index and tag from the virtual address. The physical address is still needed to access main memory, but translation is only required on cache misses, rather than every reference.

The memory controller(s) translate the address, and access the requested page.

Atlas, the first virtual memory machine, used a variation of this approach[29, 50]. The Atlas memory controller performed a fully-associative hardware search to locate the matching page. Unfortunately, in today's technology, associative memory costs too much and performs too slowly to scale to large memories. The MU-6G used a centralized translation lookaside buffer rather than a fully-associative search. In her dissertation, Knapp proposes a similar design.

Despite the advantages of centralizing the translation information, the cost and complexity of this approach has prevented it from being implemented in commercial systems.

3.2 The Case for In-Cache Address Translation

Centralizing address translation simplifies the translation coherency problem by eliminating the translation lookaside buffer from each processor. While other problems prevent this centralized approach from being attractive, it suggests that we consider other alternatives that do not require translation lookaside buffers.

As described above, virtual address caches do not require fast translation lookaside buffer implementations because we only reference them on cache misses. But the primary function of a translation lookaside buffer is to reduce the average access time to pagetable entries. This apparent contradiction suggests the following question: since we already have a large general-purpose cache, why do we need a much smaller, special-purpose cache for pagetable entries?

Several simulation studies have shown that separate caches for instructions and data generally have worse performance than unified caches of the same total capacity[77, 2]. These studies show similar results for split user/kernel caches as well. We conjecture that the same result holds for having separate translation lookaside buffers and data (and instruction) caches. By combining the functions into a single unified cache, we should improve overall performance.

This is the fundamental observation behind in-cache address translation. Rather than have a small special-purpose cache just for pagetable entries, we can keep the entries in the large general-purpose virtual address cache just like ordinary data. When we need to translate an address, we simply look for the pagetable entry in the regular cache. Since the cache only holds part of the pagetable at any one time, it may not contain the necessary entry. But because this cache is much larger than a translation lookaside buffer, there is a high probability that we will find the pagetable entry in the cache, even though it must compete for space with instructions and data. If we do not find the entry, then we must access main memory, just like a translation lookaside buffer miss.

In-cache address translation has several advantages over the other translation schemes. First, it integrates the functions of the translation lookaside buffer with the data and instruction cache, resulting in simpler logic and fewer components in most technologies. Reducing

implementation cost is always important in commercial systems. But with rapid improvements in technology, reducing the design time is very important, because simpler systems can be designed and tested more quickly. By the time a complex system is ready for sale, it must often compete with simpler systems built using the next generation technology. By simplifying the design of address translation, in-cache translation helps reduce the design time.

A second advantage comes from improved performance, as described in Chapter 4. Just as unified instruction and data caches have lower miss ratios than split caches, in-cache translation has a lower total miss ratio for large caches. Because data caches are much larger than translation lookaside buffers, usually by 2 or 3 orders of magnitude, in-cache translation generally has superior performance.

Finally, in-cache address translation simplifies the translation consistency problem, to be described in Section 3.5. By eliminating the translation lookaside buffer, in-cache translation eliminates this additional cache consistency problem. The regular data cache consistency mechanism maintains a consistent image of the translation information.

3.3 The In-Cache Translation Algorithm

In-cache translation works because the virtual address cache filters the reference stream; cache hits proceed immediately and only the less frequent cache misses require address translation. But when a cache miss does occur, the controller must generate the physical address needed to access main memory. Rather than look in a special-purpose cache for the pagetable entry, the controller looks in the general-purpose data cache.

But how does the controller find the pagetable entry in the cache? To solve this problem, we place the pagetable at a well known location in the virtual address space. The pagetable is organized as a large array, contiguous in virtual space. The page number of the virtual address (the high-order bits) forms the index into the pagetable. Given the base address of the pagetable, the address computation, illustrated in Figure 3.6, requires a simple field extract and addition. Using this constructed address, the controller looks for the pagetable entry in the virtual address cache.

In the simplest and most common case, the controller finds the pagetable entry in the cache and completes translation. The controller first checks the status bits to detect special handling and exceptions, such as non-cacheable pages and page faults. If no exceptions arise, it extracts the physical page number from the pagetable entry and constructs the physical address. Finally, the controller sends the physical address to main memory and completes the miss processing.

The more interesting case arises when the pagetable entry is not in the cache. The controller must bring the pagetable entry into the cache before processing the miss on the target address. This requires that we determine the physical address of the pagetable entry so we can fetch it from main memory.

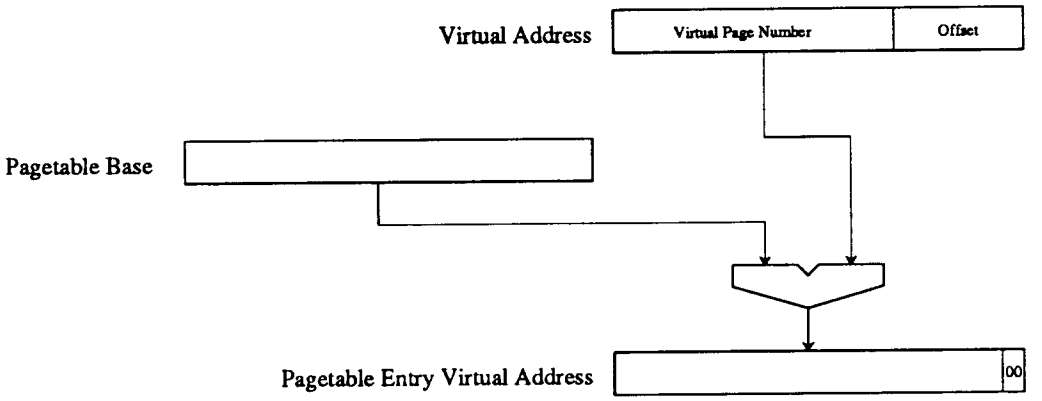


Figure 3.6: Pagetable Entry Virtual Address Calculation

This figure illustrates the computation of the pagetable entry virtual address from the virtual address of the requested word. Since the pagetable is implemented as an array in the virtual address space, the address computation is simply the base plus an offset. The page number of the virtual address, the high-order bits, is shifted right and added to the pagetable base address.

Because the pagetable resides in the virtual space, and since it maps the entire space, *a portion of the pagetable maps the pagetable itself*. In other words, a region of the pagetable contains the physical address mapping of the entire pagetable. We call this region of the pagetable the second-level, or *root*, pagetable.

This self-referential mapping means that we treat the pagetable entry miss just like an ordinary miss. Given the virtual address of a pagetable entry, we determine the virtual address of *its pagetable entry* (the root pagetable entry for the target address) using the same address mapping described above. Using this address, the controller looks for the root pagetable entry in the cache. If it is there, the controller translates the (first-level) pagetable entry address, and brings the entry in from main memory. Then translation for the target address proceeds as before.

However, when we look for the root pagetable entry, it may not be in the cache. Since the root pagetable entry resides in virtual space, it must in turn be mapped by another pagetable entry, called the third-level, or root^2 (root-squared), pagetable entry. If this entry is in the cache, we can use it to bring in the root pagetable entry and complete translation.

Clearly, the root^2 pagetable entry may not be in the cache either, and this translation process could recurse forever. However, at each level the root^N pagetable decreases in size by a factor of the page size over the pagetable entry size (a factor of 1024 in SPUR). Eventually, we get to a level where the pagetable consists of a single pagetable entry, which maps the page that contains it. This pagetable, called the root* (root-star) pagetable, terminates the recursion. Further application of the mapping continues to generate the address of the

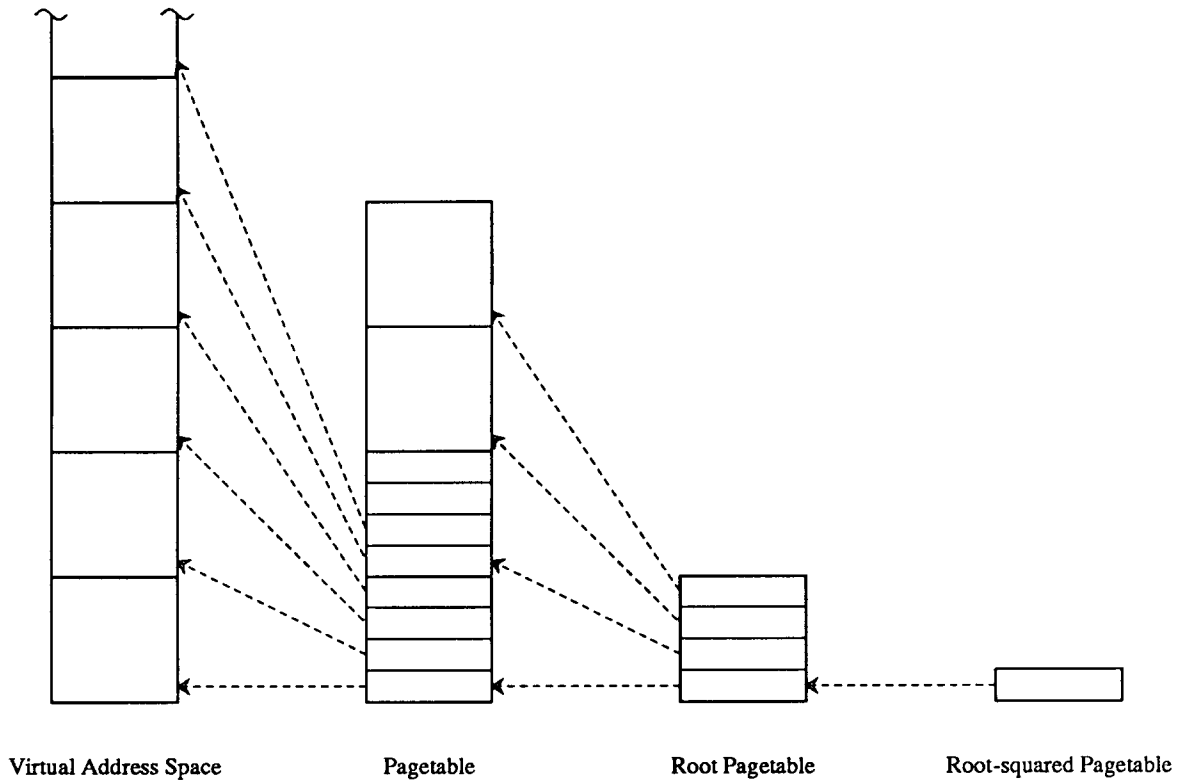


Figure 3.7: Self-Referential Pagetable Mapping

This figure illustrates a self-referential pagetable with 3 levels. We have broken out the different levels of the pagetable to emphasize the hierarchical mapping, but it is important to realize that the pagetables are merely a subset of the virtual address space and that each higher-level pagetable is a subset of the lower levels. In this example, the virtual address space consists of 16 pages, each containing 4 words. The pagetable requires 16 pagetable entries, one word each, for a total of 4 pages. Thus the pagetable consumes the lower quarter of the virtual address space in this example. The root pagetable is the portion of the pagetable that maps itself; this requires 4 entries, or one page. The root-squared pagetable contains only one entry, the root* pagetable entry, which maps the root pagetable.

root* pagetable. Figure 3.7 illustrates the pagetable and its self-referential behavior.

```
/* Global State */
virtAddr pteBaseVA;     /* Base of pagetable in virtual space */
physAddr rootPA;        /* Physical address of root-star pagetable */

/* Translate target virtual address */
Translate(targetVA)
{
      if (targetVA == PteVA(targetVA)) {
            /* End recursion, terminate with physical address */
            ReadDataFromMemory(rootPA);
      }
      else if (Hit(PteVA(targetVA))) {
            /* Pagetable entry is in the cache, go get the target */
            ReadDataFromMemory(TargetPA(PteVA(targetVA)))
      }
      else {
            /* Miss, pagetable entry is not in cache */
            Translate(PteVA(targetVA));
      }
}

/* Compute pagetable entry virtual address from target virtual address */
PteVA(targetVA) {
      return(pteBaseVA + VirtPageNumber(targetVA)<<2 )
}
```

Figure 3.8: Pseudo-Code for Translation Algorithm

When the controller fails to find the root* pagetable entry in the cache, it must terminate the recursion by fetching the entry directly from main memory. The controller does this by storing the physical address of the root* pagetable entry in a special register. Figure 3.8 shows the pseudo-code for the translation algorithm.

3.4 Translation on Writebacks

A virtual address cache requires translation whenever it accesses main memory. However, in addition to bringing data into the cache, we must also write modified blocks back to

memory when they are selected for replacement[2]. Although we would like to use the same translation algorithm in both cases, this can lead to deadlock for some cache configurations.

Deadlock may occur because writing a dirty block back to memory requires an access to the corresponding pagetable entry. If the pagetable reference misses and all blocks in the selected set are also dirty, then the controller must make room for the pagetable entry by first writing another dirty block back to memory. This in turn requires an access to the second dirty block's pagetable entry. This recursive procedure can easily result in a cyclic dependency, resulting in deadlock. For example, consider the extreme case when the cache is full of dirty data blocks: since every block is dirty and none of the pagetable entries currently reside in the cache, the controller cannot fetch the pagetable entries needed to write the dirty blocks back to memory.

We can prevent deadlock if the cache has a degree of associativity greater than the depth of recursion in the pagetable. If the depth of recursion in the pagetable is N, then we can prevent deadlock by guaranteeing that there are always N clean blocks in each set. Unfortunately, this solution is difficult to implement and may cause many more writebacks than otherwise necessary. More importantly, this solution precludes the low-associativity caches implemented in most modern systems.

Having a separate *write buffer* does not solve the deadlock problem. High-performance systems often include write buffers to reduce the latency of cache misses: the controller copies the dirty block to the buffer before bringing the requested block in from memory. This reduces the latency since the processor continues execution in parallel with the writeback.

At first glance, a write buffer seems to solve the problem by providing a place to put dirty blocks while we access the pagetable entries. However, to prevent deadlock, the write buffer must be nearly as large as the cache itself. The worst case condition arises when the cache is full of dirty data blocks. For each block written back, N additional blocks must be added to the write buffer. This forces the write buffer to hold approximately $\frac{N-1}{N}|C| + N$ blocks, where $|C|$ is the cache size. Since N is at least 2, a write buffer is clearly impractical.

Alternatively, we can provide a *read buffer* as a place to hold pagetable entries that would otherwise cause writebacks. Since we can guarantee that entries in the read buffer are always clean, it must only hold enough entries to permit a full translation (N). The disadvantage of this approach is that it makes cache coherency more complex (as does a write buffer) because the cache controller must detect references to blocks in the read buffer as well as in the cache. In addition, a read buffer does not improve performance in the same way as a write buffer, so the additional hardware cost and complexity is less easily justified.

A read buffer works because it provides an alternative place to put pagetable entries when they would otherwise cause cyclic writebacks. We can achieve the same goal by simply not caching pagetable entries in this case. Caches usually provide non-cacheable pages to support memory-mapped I/O devices. In this alternative, the controller bypasses the cache

[2]We assume a writeback cache throughout this dissertation, although the same arguments extend to write-through caches.

when a pagetable reference would cause a cyclic writeback. The main drawback of this approach is that we lose some performance by not caching all pagetable entries.

The final solution to the deadlock problem is to save the translation when the block is brought into the cache. This entails maintaining a separate set of physical address tags, along with the virtual address tags used to access the cache. When bringing a block into the cache, the controller saves the physical address in this second set of tags. When writing a block back to memory, the controller simply uses the saved physical address. Since writebacks do not require translation, deadlock cannot occur.

This solution has two drawbacks. First, the physical address tags require additional fast storage, although they do not need to be as fast as the virtual address tags since they are only accessed on writebacks. However, if we use the physical address tags to maintain cache coherency, as Goodman proposes[37], then they are already necessary. The second drawback is that since we cache physical addresses in addition to virtual addresses, we effectively cache the virtual/physical mapping. This makes translation consistency somewhat more difficult to maintain, as discussed in the next section.

Of these alternatives, caching the physical tags and selectively not caching pagetable entries are the most attractive choices. In the SPUR workstation, we decided to cache the physical tags because this simplified the hardware design: although the tags require more transistors than a read buffer, they are more regular (merely a RAM array) and require less control.

3.5 Translation Consistency

Maintaining a consistent view of the virtual/physical mapping causes two basic problems. In a multiprocessor with a translation lookaside buffer per processor, each processor may have copies of the same pagetable entry. This is called the *translation lookaside buffer consistency problem* [84], and is directly analogous to the data cache consistency problem. When one processor modifies a pagetable entry it makes other copies in the translation lookaside buffers out of date. In systems with virtual address caches, each cache block duplicates part of the translation information. This problem, called the *virtual cache consistency problem* [51], arises when modifying a pagetable entry makes the translation information stored with each cache block out of date. Uniprocessors suffer from this problem as well as multiprocessors.

Few multiprocessors provide hardware explicitly to maintain consistency across their translation lookaside buffers. Instead, the operating system must use interprocessor interrupts to notify processors when they must take action. The Mach *shootdown* algorithm [12] requires that the operating system track which processors have had access to a particular page. When a processor changes a pagetable entry, it interrupts only those processors that may have the entry in their translation lookaside buffers; the interrupted processors respond by invalidating the entry. The performance of this algorithm decreases linearly as the number of processors increases, posing a potentially serious performance problem for large multiprocessor systems[89, 70].

In-cache translation eliminates the translation lookaside buffer consistency problem by eliminating the translation lookaside buffer. Under in-cache translation, pagetable entries reside only in the data cache, so the regular cache consistency mechanism guarantees that all processors see the current mapping. This is a major advantage of in-cache translation.

However, since in-cache translation relies on a virtual address cache, we must still solve the virtual cache consistency problem. In a single processor system, this entails either invalidating or updating blocks from the changing page. For example, to change the protection of a page, the processor updates the pagetable entry then flushes or updates each block from the page that already resides in the cache. This guarantees that all future references see the new protection[3].

```
/*
** Update protection on page and force all other processors to see change
*/
UpdatePageProt(pageVA, prot)
virtAddr pageVA;
hwProt   prot;
{
      int i;
      virtAddr pteVA;

      pteVA = GetpteVA(pageVA);
      pteVA->prot = prot;
      for (i=pageVA; i < pageVA+PAGE_SIZE; i = i + BLOCK_SIZE) {
          FlushBlock(i);       /* Guarantee a cache miss */
          ReadPrivateBlock(i);
      }
}
```

Figure 3.9: Pseudo-Code for Updating Page Protection

To extend this approach to a multiprocessor, we must be able to update or invalidate blocks in other caches. But this is exactly the function of the regular cache consistency protocol. For example, SPUR's Berkeley Ownership is an invalidation protocol[49], meaning that when one cache wants to write a data block, it obtains a private copy by invalidating all other copies. The protocol uses two special bus operations: *ReadForOwnership* reads a block

[3]Not all protection changes must be seen immediately. Changes that increase the permissions, i.e., read-only to read-write, can often be deferred without problems. The trap handler need only detect that the violation occurred unnecessarily.

and makes it private by invalidating all other copies, *WriteForInvalidation* changes a shared block to private by invalidating other copies. To update a page's protection in SPUR, the initiating processor obtains ownership of each block using the protocol's *ReadForOwnership* bus operation. This guarantees that the other processors must access the pagetable entry, and hence the new translation information, before regaining access to the block. Figure 3.9 shows the pseudo-code for this routine.

The performance of the Mach shootdown algorithm is a function of the number of processors that must be interrupted; thus the algorithm becomes less efficient for large systems. Under in-cache translation, the performance is a function of the number of blocks that must be invalidated. But measurements from Mach show that usually only a few pages, and hence only a few cache blocks, are affected; three of the four workloads average less than 1.3 pages per shootdown, and the fourth averages only 2.3 pages[12]. Since the number of pages affected is a function of the application, we do not expect it to change as the number of processors increases. Therefore, the translation consistency overhead of in-cache translation should scale to larger systems better than translation lookaside buffer shootdown schemes.

In-cache translation simplifies the translation consistency problem because it allows the processor that makes the change to guarantee the update, with minimal interference to other processors. With a translation lookaside buffer, unless the hardware provides additional bus operations to remotely invalidate pagetable entries, the operating system must interrupt other processors to guarantee they see the update.

3.6 SPUR: A Realization of In-Cache Translation

The SPUR workstation implements a version of in-cache translation[91, 92], which differs slightly from the general form described earlier. The main difference is a performance optimization made possible by SPUR's solution to the synonym problem. SPUR supports a single, segmented virtual address space, called the *global virtual address space*, that is shared by all processes. The global virtual address space consists of 256 segments, each one gigabyte, for a total of 256 gigabytes of virtual storage, requiring 38 bits of address. A processor has only 32 bits of address, called the *process virtual address*, so it can only access four segments at one time. The top two bits of the process virtual address select one of four *global segment* registers, as illustrated in Figure 3.10. Because segments are large, we form the cache index from the process virtual address. This allows the global segment lookup to proceed in parallel with the cache RAM read, since we only use the high-order bits in the tag comparison.

pping mechanism permits processes to share data at the segment level; if two processes share any portion of a segment, they share the entire segment. By restricting sharing in this way, the Sprite operating system[62] guarantees that processes always access shared data using the same global virtual address. Thus SPUR allows processes to share data without permitting virtual address synonyms.

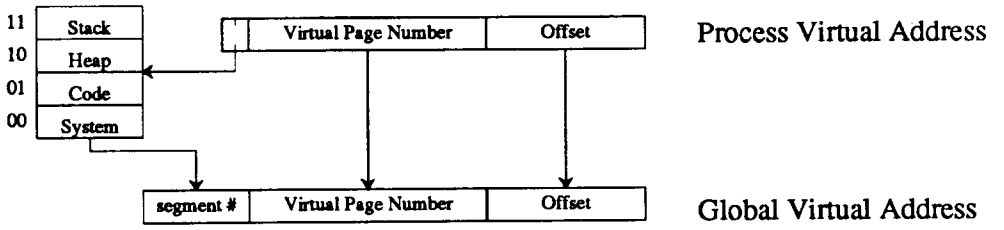


Figure 3.10: Formation of Global Virtual Address

This figure illustrates the formation of SPUR's global virtual address from the process virtual address. The high-order two bits select one of four global segment registers. We concatenate the 8-bit global segment number with the remaining bits to form the global virtual address. The Sprite operating system reserves one segment to the kernel, and uses the other three for the process's code, stack, and heap.

By software convention, the 4 segments usually contain system code and data, user code, stack, and heap. All processes share the single system segment, processes executing the same program share a code segment, and processes that share data do so by sharing a heap segment. We designed SPUR to use this simple sharing model, unfortunately, experience has shown that the model is too simple. Processes often want a private heap segment in addition to a shared heap segment. And Sprite must occasionally reference another process's segment; but since all 4 segments are always in use, Sprite must unmap one of them to gain access to the other process. This requires careful and difficult coding to prevent references to the unmapped segment. Therefore, in retrospect, SPUR should have provided at least 6 global segment registers (of course, 8 is the logical choice for implementation reasons).

The segmentation of the SPUR address space permits an optimization to the translation algorithm. The global virtual address space is 256 gigabytes, but each process only has direct access to 4 gigabytes. The root pagetable for the entire space consumes 256 kilobytes of virtual space[4], but each process only needs access to 4 one-kilobyte regions. In SPUR, we associate a *root pagetable map* register with each global segment register, to contain the physical address of the corresponding region of the root pagetable, as illustrated in Figure 3.11. If a root pagetable entry does not reside in the cache, we can directly compute its physical address and access main memory. This eliminates all references to the root^2 and root* pagetable entries that would otherwise be necessary. Thus by adding 4 registers to the cache controller, we improve the performance of the translation algorithm.

A second optimization simplifies the computation of the pagetable entry virtual address.

[4]Assuming SPUR's 4-kilobyte pages and 4-byte pagetable entries.

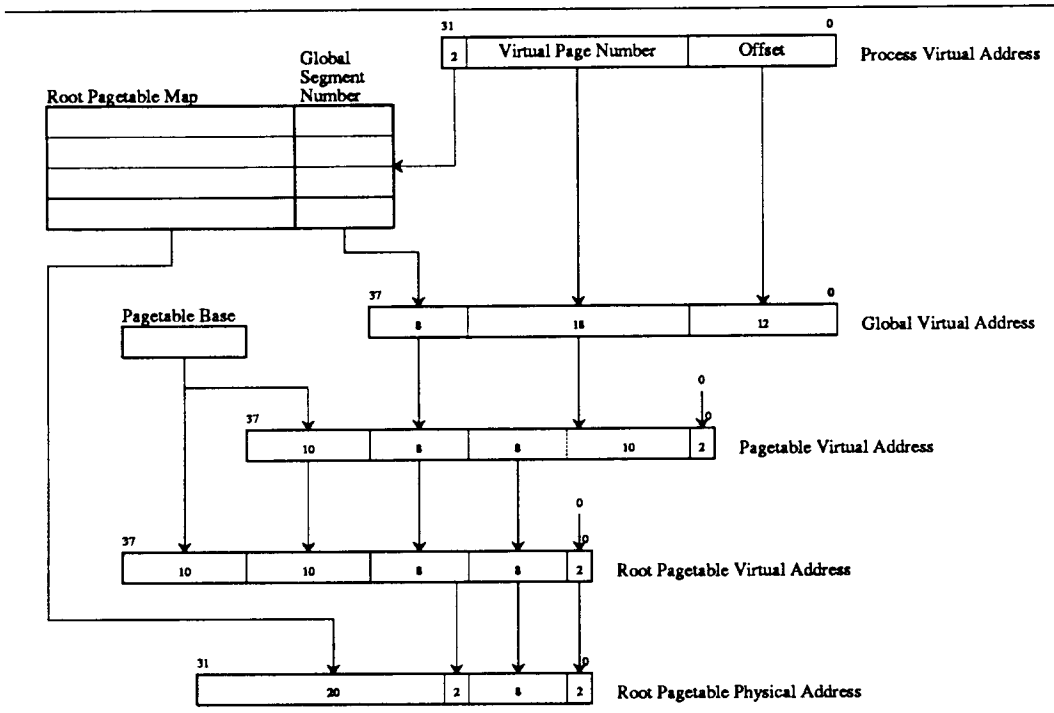


Figure 3.11: SPUR In-Cache Translation Process

This figure illustrates the formation of addresses in the SPUR realization of in-cache address translation. The process virtual address is mapped to the global virtual address using the global segment registers. The pagetable virtual address is computed by shifting the global virtual address right by 10 bits and concatenating it with the pagetable base address. This process is repeated to form the root pagetable virtual address. The root pagetable map registers point directly to the physical memory containing the portions of the root pagetable that correspond to the current global segments. A root pagetable entry physical address is formed by contatenating the contents of a root pagetable map register with the low 12 bits from the root pagetable virtual address.

SPUR restricts the pagetable to be aligned on a 256-megabyte boundary in global virtual space. This eliminates the address addition described in Section 3.3, and simplifies the address computation to a field-extract and concatenate operation. This executes much faster than a long addition, and requires fewer transistors to implement.

3.7 Summary

In this chapter we have introduced the in-cache address translation algorithm, showing how it integrates the functionality of the translation lookaside buffer with the virtual address cache. We argue qualitatively that by merging the two units we reduce the cost and complexity of the implementation. Also, since the virtual address cache is much larger than any realistic translation lookaside buffer, the performance of in-cache translation should be superior.

We have described the general form of the translation algorithm, explaining the simple address calculation the cache controller performs to compute the virtual address of the pagetable entry. This recursive procedure uses a self-referential pagetable structure, terminating when either it finds a pagetable entry in the cache, or does not find the root* pagetable entry in the cache. When the later condition occurs, the cache controller takes the physical address of the root* pagetable entry from a special register and accesses memory directly.

We have discussed the problems of translation consistency, and have shown how in-cache translation simplifies them. Because the pagetable entries only reside in the regular instruction and data cache, the operating system can use the semantics of the cache coherency protocol to maintain a consistent image of the translation information.

Finally, we describe the specific realization of in-cache translation used in SPUR, which differs slightly from the general algorithm. Rather than recurse all the way to the root* pagetable, SPUR terminates translation after two levels (the root pagetable). This requires 4 registers to contain the physical addresses of the appropriate regions of the root pagetable, rather than only a single entry. We show in Chapter 4 that the higher levels of the pagetable have much larger miss ratios.

Chapter 4

Analysis

In this chapter we analyze the performance of in-cache address translation by asking the same questions normally asked about conventional caches and translation lookaside buffers. How often does the cache or buffer miss? How many cycles does an average reference require? What is the overhead of address translation? How do these results change for different cache and buffer parameters?

We also look closely at how in-cache translation affects the cache. How frequently do we find the pagetable entries in the cache during translation? How often do pagetable entries kick out useful instructions and data?

To put these results in context, we compare the performance of in-cache translation to conventional caches with translation lookaside buffers. We show that in-cache translation has superior performance, on average, over a wide range of cache and translation lookaside buffer configurations. The conventional approach is only superior for very large buffers, at least 256 entries and 2-way set-associative, small caches, 32 kilobytes or less, or when the memory access time is very long, at least 20 cycles. If the cycle times are equal, then the performance differences are usually less than 5%. However, since a virtual address cache can reduce the cycle time by as much as 50%, in-cache translation can have substantially better performance than a translation lookaside buffer with a physical address cache.

This chapter contains three major sections. Section 4.1 describes the trace-driven simulation methodology, including the metrics and address traces. Section 4.2 presents an analysis of in-cache translation, using the parameters of the SPUR workstation as the baseline configuration. It shows that pagetable entries consume only 2.0% of the cache, on average, and that address translation consumes only 2.2% of the cycles. In comparison to 11 typical translation lookaside buffer configurations, only two of the largest buffers have better average performance.

Section 4.3 tests the sensitivity of these results to the baseline cache parameters. We examine the performance of in-cache while varying the cache size, block size, associativity, page size, and miss penalty. The performance is most sensitive to cache size: in-cache only performs well for caches of at least 64 kilobytes. However, since large caches are now

commonplace, this is not a major limitation[1]. The performance is much less sensitive to the other parameters, indicating that in-cache translation performs better than translation lookaside buffers for many cache configurations.

4.1 Simulation Methodology

Trace-driven simulation is still the preferred method to study cache and memory system behavior [77]. Analytic models have not yet proven general enough nor accurate enough to replace simulation, despite the obvious advantages of reducing the analysis time. Measurements of real implementations provide the most accurate results, but the cost and complexity of implementation generally limits the evaluation to a single existing cache configuration. Trace-driven simulation accurately models the cache operation for the workload captured in the address traces. The flexibility of simulation allows us to study many cache configurations, as well as entirely new designs like in-cache address translation.

However, trace-driven simulation is not without its limitations. Simulations require large amounts of CPU time and disk storage. These resource requirements limit simulations to the equivalent of at most a few seconds of actual execution time. This introduces a significant risk that the address traces do not accurately characterize the intended workload. Careful trace selection, discussed in Section 4.1.2, reduces this risk, and improves the quality of the results. Furthermore, we are able to validate some of the key results in Chapter 6, using measurements from the SPUR prototype.

4.1.1 Metrics

Traditionally, caches have been evaluated using the *miss ratio*, defined as the fraction of references that miss in the cache:

$$m_{cache} = \frac{\text{the number of cache misses}}{\text{the number of references}} \tag{4.1}$$

The miss ratio is easy to compute, depending only upon the cache organization and workload. Since it is independent of specific implementation details, such as cycle time, the miss ratio facilitates comparisons between different cache organizations.

However, the miss ratio can be a misleading indicator of total system performance. Two different caches may have the same miss ratio but very different performance if the *miss penalties*, that is, the time to bring a block of data from memory into the cache, are different. The *effective access time* takes this factor into account:

$$t_{eff} = \text{effective access time}$$

[1]Many systems, including SPUR, use multiple levels of caches with a small cache or instruction buffer close to the processor. However, the second-level caches are large, usually containing at least 64 kilobytes.

$$= t_{cache} + m_{cache} t_{memory} \tag{4.2}$$

where m_{cache}, t_{cache}, and t_{memory} are the cache miss ratio, the cache access time, and the average memory access time, respectively. The average memory access time includes the memory latency and transfer time, overhead to perform housekeeping functions, plus the cost of writing a dirty cache block back to memory when necessary:

$$t_{memory} = t_{readblock} + f_{writeback} t_{writeback} \tag{4.3}$$

where $t_{readblock}$ is the time to get a block from memory and update the cache, $f_{writeback}$ is the fraction of blocks that must be written back to memory, and $t_{writeback}$ is the time to write a block back to memory.

In addition to the basic cache metrics, we also need metrics to measure the performance of address translation. Since a translation lookaside buffer is just a special-purpose cache, its performance is commonly measured by its miss ratio:

$$m_{tlb} = \frac{\text{the number of translation lookaside buffer misses}}{\text{the number of references}} \tag{4.4}$$

The effective access time must also include the delays due to address translation. For most of the analysis, we assume that the translation lookaside buffer access overlaps with the cache access and does not degrade the cycle time. With this assumption, the effective access time must include the penalty of missing in the translation lookaside buffer, as well as in the cache:

$$t_{tlb} = t_{cache} + m_{tlb} t_{tlbmiss} + m_{cache} t_{memory} \tag{4.5}$$

where m_{tlb} and $t_{tlbmiss}$ are the translation lookaside buffer miss ratio and miss penalty respectively. To simplify our simulations, we assume that translation lookaside buffer misses do not interfere with the cache. Since this interference does occur in most real systems, t_{tlb} is a slightly optimistic estimate of the effective access time.

To measure the performance of in-cache address translation, we need some additional metrics. First, the *total miss ratio* for in-cache translation includes both misses to instructions and data and misses to pagetable entries, normalized to the number of processor references:

$$m_{incache} = \frac{\text{number of cache misses, including pagetable entry misses}}{\text{number of processor references}} \tag{4.6}$$

We exclude *references* for pagetable entries from the denominator so that $m_{incache}$ is directly comparable to m_{cache}. To reduce confusion between the similar names of the two metrics, we refer to m_{cache} as m_{ideal} for the remainder of the chapter (similarly, t_{eff} is called t_{ideal}). We can think of m_{ideal} as the total miss ratio for a cache with *ideal* address translation, which has no affect on the miss ratio or effective access time.

We also need a metric that lets us compare in-cache translation against m_{tlb}. One

possibility is the pagetable entry miss ratio:

$$m_{pte} = \frac{\text{number of pagetable entry misses}}{\text{number of references}} \tag{4.7}$$

But this metric does not include misses to instructions and data that were knocked out of the cache by pagetable entries. These *collisions* are also a cost of in-cache translation. The collision miss ratio is defined as:

$$m_{coll} = \frac{\text{number of misses to instructions and data knocked out by pagetable entries}}{\text{number of references}} \tag{4.8}$$

Then we can define the *translation miss ratio* for in-cache address translation as:

$$m_{trans} = m_{pte} + m_{coll} \tag{4.9}$$

An equivalent definition is the difference between the total miss ratio for in-cache, and the ideal cache miss ratio:

$$m_{trans} = m_{incache} - m_{ideal} \tag{4.10}$$

where we assume that no pagetable entries ever enter an ideal cache.

It is also useful to have metrics for each type of cache reference: CPU references, first-level pagetable entries, and root pagetable entries. The fraction of CPU references that miss in the cache under in-cache translation is:

$$m_{cpu} = \frac{\text{number of CPU misses under in-cache}}{\text{number of references}} \tag{4.11}$$

$$= m_{ideal} + m_{coll} \tag{4.12}$$

This includes those references that would have missed in an ideal cache plus those misses caused by bringing pagetable entries into the cache. Similarly, the first-level pagetable miss ratio is:

$$\begin{aligned} m_{upte} &= \frac{\text{number of first-level pagetable misses}}{\text{number of first-level pagetable references}} \\ &= \frac{\text{number of first-level pagetable misses}}{\text{number of CPU misses}} \end{aligned} \tag{4.13}$$

where the subscript *upte* is an abbreviation for *user* pagetable entry, a synonym for first-level pagetable entry. Finally, the miss ratio for the second-level, or root pagetable entries, is:

$$\begin{aligned} m_{rpte} &= \frac{\text{number of second-level pagetable misses}}{\text{number of second-level pagetable references}} \\ &= \frac{\text{number of second-level pagetable misses}}{\text{number of first-level pagetable misses}} \end{aligned} \tag{4.14}$$

To calculate the effective access time for in-cache translation, we must include the penalties for pagetable entry and collision misses, weighted by their frequencies. The equation for the effective access time is:

$$\begin{aligned} t_{incache} &= t_{cache} + m_{cpu}(1 - m_{upte})(t_{memory} + t_{upte-hit}) \\ &\quad + m_{cpu}m_{upte}(1 - m_{rpte})(2t_{memory} + t_{rpte-hit}) \\ &\quad + m_{cpu}m_{upte}m_{rpte}(3t_{memory} + t_{rpte-miss}) \end{aligned} \tag{4.15}$$

where $t_{upte-hit}$, $t_{rpte-hit}$, $t_{rpte-miss}$ are the overheads of translation given that: the first-level pagetable entry is in the cache, the first-level pagetable entry is not in the cache, but the second-level cache is, and neither the first-level nor the second-level pagetable entries are in the cache. Comparing this metric to t_{tlb} and t_{ideal} shows how well in-cache translation performs compared to a translation lookaside buffer scheme and versus optimal translation.

Most cache performance studies assume that t_{cache} is constant for all cache configurations. But Hill and Przybylski have independently shown that this simplifying assumption leads to erroneous conclusions of the optimal cache organization[42, 41, 67, 66]. For example, many studies have shown that increasing the associativity usually decreases the miss ratio, implying that higher associative caches achieve better performance. However, with higher associativity the cache requires more logic to transmit the data back to the processor, increasing the cache access time t_{cache}. Hill and Przybylski show that the increase in t_{cache} often outweighs the decrease in the miss ratio m_{cache}, leading to a higher effective access time and hence lower system performance.

Cache access time clearly plays an important role in determining the performance of address translation. For example, sequential address translation obviously requires a longer cycle time than in-cache translation. However, determining the cycle time for a particular organization makes the analysis dependent upon the implementation technology. To make our results more general, we calculate the metric

$$r_{breakeven} = \frac{t_{incache}}{t_{tlb}} \tag{4.16}$$

which is the ratio of the effective access times. This unitless metric facilitates implementation-dependent comparisons between in-cache translation and translation lookaside buffers. If an implementation of in-cache translation runs with cycle time $c_{incache}$, then a translation lookaside buffer must run with a cycle time of $r_{breakeven}c_{incache}$ to achieve equal performance[2]. If implementing a translation lookaside buffer would increase the access time above $r_{breakeven}c_{incache}$, then the performance of in-cache translation would be superior. With this metric we can determine which approach has the best performance for a particular implementation technology.

[2]This derivation assumes that the miss penalty scales with the cache access time. This introduces a small error, since the memory latency should remain constant. However, since $r_{breakeven}$ is usually close to one, the error is hidden by the quantization of the latency into a fixed number of cycles.

4.1.2 Address Traces

A trace-driven simulation is only as good as the address trace used as input. A simulation reflects the miss ratio of a target system only as accurately as the trace reflects its instruction set, operating system, and workload. Ideally, we would collect traces from a previous version of the target system, then use the simulation results to design the next generation. This approach produces the highest quality traces and hence the best simulation results. However, while this strategy works well for incremental improvements of an established product-line, it does not help design completely new machines.

Instead, we must predict the performance of a new machine using traces from existing systems, or from instruction-level simulations of the target machine. In either case, the traces may not accurately reflect the behavior of the target system and frequently suffer from at least one of the following problems:

- *Lack operating system references:* The most common and most important shortcoming is when a trace lacks operating system references. Agarwal [3] has shown that the operating system accounts for up to 50% of all references, and frequently has a higher miss ratio than application programs.

- *Lack multiprogramming effects:* A related problem arises when the trace only includes references from a single program. Many previous studies have shown that context switching seriously degrades the cache miss ratio.

- *Short length:* The addresses captured in a trace often represent only a fraction of a second of actual execution time. This often results from a lack of CPU time or storage when running the simulation. Sometimes the resource limitation arises during trace generation, as with the ATUM traces discussed below[4]. Short traces present a problem when simulating large caches, because cold-start behavior (discussed below) distorts the simulation results.

- *Different instruction set:* Another problem exists when using traces from one architecture to predict the performance of another. For example, consider using a Motorola MC68000 trace to predict the cache miss ratio of a RISC processor. Since the MC68000 is a 16-bit machine with variable instruction and data size, its reference pattern is quite different from a 32-bit RISC processor[28].

- *Different workload:* Different programs have different cache performance. Smith has shown that different traces can have several orders of magnitude difference in the miss ratio for the same cache [78].

- *Different compilers:* The compiler on the traced machine may differ significantly from the compiler on the target machine. The quality of the optimizer and code generator affects the size of basic blocks. Reducing the size of basic blocks decreases the number of instructions that must be executed, but may increase the miss ratio.

Trace Group	Number of Traces	Architecture / Operating System	System References	Multi-programming	Average Length
ATUM_MP	3	Vax-11 / VMS and Ultrix	yes	yes	1.6M
ATUM_UP	2	Vax-11 / VMS	yes	no	1.5M
SPUR	6	SPUR / Sprite	no	no	5M
Synapse	3	MC68000 / Synthesis	yes	yes	5M
Total	14	n/a	n/a	n/a	53M

Table 4.1: Address Trace Characteristics

This table summarizes the characteristics of the four groups of address traces: ATUM_MP, ATUM_UP, SPUR, and Synapse. Three of the four groups include operating system references, and two of the four contain multiprogramming behavior. The 14 traces combined contain 53 million references.

- *Different Operating System:* Some operating systems are more efficient than others. Agarwal showed that 20% of the references occurred in the system mode under VMS, while 50% occurred in system mode under Ultrix[2].

None of the traces used in this analysis is completely free of all the problems listed above. However, to minimize the bias of any one of these limitations, we use four sets of traces, each with different characteristics.

ATUM_MP These traces were gathered on a DEC VAX using the ATUM microcode technique[4]. These traces each consist of 4 samples of approximately 400,000 references, which are concatenated, or *stitched*, together to form a longer trace, as discussed below. The *Mul2* trace consists of SPICE, a circuit simulator, and ALLOC, a microcode address allocator, running concurrently under VMS. The *Mul8* trace consists of Spice, Alloc, a FORTRAN compile, a PASCAL compile, an assembler, a string search, a numerical benchmark (JACOBI) and an octal dump[4] running concurrently under VMS. The *Savec* trace consists of the C compiler and two other tasks running concurrently under Ultrix[3]. All three traces include both system references and multiprogramming effects.

ATUM_UP These traces were also gathered with the ATUM microcode technique, but these workloads exhibit very little multiprogramming behavior. Each of the traces also

Group	Trace	Length (1000s)	Percent Instructions	Number of Blocks	Number of Pages
ATUM_MP	mul2	1507	54%	14027	929
	mul8	1628	50%	20422	1549
	savec	1530	52%	7424	277
ATUM_UP	dec0	1439	51%	6625	537
	forf	1569	52%	14403	757
SPUR	rsim	5000	91%	7864	116
	rsim2nd5M	5000	88%	10749	187
	slc	5000	83%	20915	622
	slc2nd5M	5000	84%	18418	576
	weaver	5000	88%	6082	477
	weaver2nd5M	5000	88%	4713	265
Synapse	devel	5000	66%	6042	271
	prod5M	5000	63%	16969	497
	prod2nd5M	5000	62%	17964	561

Table 4.2: Address Trace Data

This table summarizes basic data from the address traces. The number of unique 32-byte blocks and 4-kilobyte pages referenced in the trace, listed in the last two columns, indicate the size of the workload captured by the trace.

consists of 4 concatenated samples. *Dec0* is a trace of DECSIM, a behavioral level simulator, simulating some cache hardware. *Forf* traces the execution of a FORTRAN compile. Both these traces include system references (VMS), but have few context switches, and hence little multiprogramming behavior.

SPUR These user-mode only traces were generated by *Barb* [95], the SPUR instruction level simulator. There are two 5 million reference samples from each of 3 different Lisp applications, for a total of 6 traces. *Weaver* and *Weaver2nd5M* trace a VLSI routing program that runs on top of the OPS5 rules system [46]. *Rsim* and *Rsim2nd5M* trace the switch-level circuit simulation of an adder [85]. *Slc* and *Slc2nd5M* trace the SPUR Lisp compiler compiling several of the Gabriel benchmarks[33]. These traces include neither operating system references nor multiprogramming behavior. In addition, the Lisp compiler lacks an instruction scheduler[3], increasing the average basic block size and tending to decrease the miss ratio for caches with large blocks[95]. The advantage of these traces is that they are long, they use the SPUR instruction set, and the applications are large Lisp programs.

Synapse These traces were gathered from a uniprocessor Synapse N+1 [30], a Motorola MC68000 based multiprocessor, using a hardware tracing system. The *Devel* trace is taken from a synthetic software development workload, consisting of an edit, compile, debug sequence. The *Prod5M* and *Prod2nd5M* traces are from a synthetic transaction processing workload (TP1[7]) running on the Synapse relational database system. The traces are 5 million references long and include both system references and multiprogramming effects.

Table 4.1 summarizes the trace characteristics, and Table 4.2 summarizes basic trace data such as trace length and number of unique cache blocks. The 14 traces contain a total of 53 million references.

4.1.3 Start-up Behavior

Trace-driven simulation suffers from a problem known as *cold-start behavior*[25] or *start-up distortion*[2]. This problem occurs because while a cache is empty at the start of a simulation, it is rarely empty in a real system. Thus the simulated miss ratio is greater than the true miss ratio until the cache fills up. For short traces and large caches this distortion dominates the miss ratio, introducing significant error.

Easton and Fagin address this problem by defining two miss ratios for fully-associative caches[25]. The *cold-start miss ratio* is measured from an initially empty cache; the *warm-start miss ratio* is measured only after the cache has filled. By eliminating the cold-start

[3]The SPUR CPU, like most RISC processors, implements delayed branches and loads. Initially the compiler generates NOP instructions in the delay slots of these instructions, since the compiler itself only performs machine-independent optimizations. A separate program called an instruction scheduler should reorganize the code to eliminate unnecessary NOPs.

misses, this second miss ratio eliminates the start-up distortion. However, while fully-associative caches fill quickly, large set-associative caches take arbitrarily long, requiring prohibitively long address traces.

Agarwal proposes a more general definition of warm-start for set-associative caches: the cache is warm when it has either filled up or the initial working set of the workload is resident[2]. The initial working set is determined by locating the knee in cumulative cold-miss curve. He shows that even large caches fill quickly for uniprogramming traces, requiring only 25,000 references. However, multiprogramming traces require much longer, requiring as many as 600,000 references for 256 kilobyte caches.

The SPUR and Synapse traces, containing 5 million references each, are long enough to compute warm-start miss ratios, using Agarwal's definition. The ATUM traces, on the other hand, consist of samples containing roughly 400,000 references each. These samples are too short to compute warm-start miss ratios for multiprogramming traces and large caches. Agarwal proposes concatenating the samples together to form a longer *stitched* trace, rather than simulating the samples independently.

We show in Appendix A that stitched traces approximate the true miss ratio better than simulating the samples independently, for 75% of the simulations. As the cache size increases, stitching becomes more important: for caches larger than 64 kilobytes the stitched traces are better in over 90% of the simulations. However, we also show that even stitched traces have a large relative error for large caches: the error averages 20% for 256 kilobyte caches.

Since trace stitching produces smaller errors than simulating the samples independently, we construct each of the ATUM_UP and ATUM_MP traces by concatenating 4 samples. This makes the shortest trace contain over 1.4 million references, allowing us to compute warm-start miss ratios. We approximate warm-start by warming up the cache with the first 500,000 references, and computing the miss ratio from the remaining references.

4.1.4 Averaging

To summarize results, we use the unweighted arithmetic mean of the miss ratios (effective access times)

$$mean = \frac{1}{N}\sum_{i=1}^{N} m_i \tag{4.17}$$

where m_i is the miss ratio for trace i. This is the correct mean assuming that each trace is equally weighted. If we wished to relax this assumption, we would use the weighted arithmetic mean.

To see that we should use the arithmetic mean rather than either the geometric or harmonic mean, consider a model workload **W** consisting of N traces w_i, each with miss ratio m_i. If the total length of **W** is I references, and each trace w_i makes up a fraction f_i of **W** ($\sum f_i = 1$), then each trace w_i contains $I_i = f_i I$ references, and generates $m_i f_i I = m_i I_i$

misses. Now, the miss ratio for **W** is just the total number of misses from all traces over the total number of references.

$$\begin{aligned} \text{miss ratio of } \mathbf{W} &= \frac{\text{total misses}}{\text{total references}} \\ &= \frac{\sum_{i=1}^{N} m_i I_i}{\sum_{i=1}^{N} I_i} \\ &= \sum_{i=1}^{N} \frac{m_i I_i}{I} \\ &= \sum_{i=1}^{N} f_i m_i \end{aligned} \tag{4.18}$$

which is just the weighted arithmetic mean. To weight the traces equally, we assume that $f_i = 1/N$, then the average miss ratio for **W** is just the unweighted arithmetic mean.

Intuitively, the arithmetic mean is the correct mean because the miss ratio is proportional to a program's execution time [81]. Since this is also true for the effective access time, we also use the arithmetic mean to summarize these results. As discussed in the last section, the traces are broken into four groups. We use the unweighted arithmetic mean to compute an average for each group, and average the group averages to compute an overall average, This has the effect of weighting each group equally, rather than weighting each trace equally. This prevents the SPUR trace group, containing twice as many traces as the next largest group, from unduly biasing the overall average.

4.2 Baseline Analysis

We begin our analysis by examining the performance of in-cache address translation for the cache parameters of the SPUR prototype, summarized in Table 4.3. The memory system of SPUR is representative of modern high-performance workstations, and contains the original implementation of in-cache translation. We exclude the specific memory timing of SPUR, using idealized values for a more abstract and general evaluation.

The first results concern the performance of in-cache translation with this configuration, focusing on the interaction between pagetable entries and instructions and data. Since pagetable entries consume only 2% of the cache, on average, that they do not cause significant interference for a cache of this size. We look at both miss ratios and effective access time, and show that in-cache translation consumes only 2.2% of the time on average. We compare these results to the performance of 11 translation lookaside buffer configurations, and show that only 2 of the largest buffers have superior performance, and that these 2 are within 1% of in-cache translation.

Cache Size	128 kilobytes
Block Size	32 bytes
Associativity	Direct-Mapped
Page Size	4096 bytes
Miss Penalty	12 Cycles
Writeback Penalty	12 Cycles

Table 4.3: Baseline Simulation Parameters

4.2.1 Miss Ratio Analysis

Under in-cache translation, we can view the unified cache as a regular cache being *polluted* by pagetable entries. Bringing pagetable entries into the cache knocks out instructions and data, causing cache misses that would not otherwise occur. But since pages are large, 4 kilobytes in SPUR, each pagetable entry contains the mapping for many cache blocks. Thus we expect that only a small fraction of the cache contains pagetable entries. The first column of Table 4.4 shows this is true: pagetable entries consume only 2% of the cache on average, and less than 4% in the worst case.

This result suggests that pagetable entries collide with instructions and data infrequently, causing only a small increase in the miss ratio (m_{coll}). Table 4.4 shows that the maximum absolute increase is only .065%, with a maximum relative increase of 8% (occurring for the trace with the lowest m_{ideal}). On average, the collisions increase the absolute miss ratio by .029%, a relative increase of only 2%.

An alternative way to view the cache is as a very large translation lookaside buffer polluted by instructions and data. The pagetable entry miss ratio (m_{pte}) is larger than the collision miss ratio; this follows from the observation that at most one block of instructions and data is knocked out of the cache for each block of pagetable entries brought into it[4]. For these traces, m_{pte} averages 2.5 times larger than m_{coll}. As shown in Table 4.4, the pagetable entry miss ratio ranges from as low as .023% to as high as .17%. Combining m_{coll} and m_{pte} produces m_{trans}, the translation miss ratio. Even including both pagetable entry and collision misses, this miss ratio is very low, ranging from .038% to .23%. This amounts to an average of only 10.3 misses per 10000 CPU references, clearly a small impact on total performance.

The translation miss ratio m_{trans} varies considerably from trace to trace. This is due in part to the large variance in the basic cache miss ratio m_{ideal}: a program that misses frequently on instructions and data often knocks out pagetable entries as well. While this

[4]This observation holds only for direct-mapped caches. For set-associative caches, m_{coll} also includes collisions between instructions and data that occur because pagetable entries reduce the effective capacity of the cache.

Trace	Percent PTEs	m_{coll}	m_{pte}	m_{trans}	m_{ideal}	$m_{incache}$	$\frac{m_{incache}}{m_{ideal}}$
mul2	2.9%	0.00042	0.00120	0.00162	0.02427	0.02588	1.06659
mul8	3.8%	0.00041	0.00127	0.00168	0.02237	0.02405	1.07532
savec	1.0%	0.00023	0.00049	0.00071	0.01076	0.01147	1.06632
dec0	2.8%	0.00015	0.00039	0.00055	0.01334	0.01389	1.04109
forf	3.0%	0.00037	0.00085	0.00123	0.01783	0.01906	1.06881
rsim	0.65%	0.00015	0.00030	0.00044	0.00180	0.00224	1.24651
rsim2nd5M	0.61%	0.00029	0.00057	0.00086	0.01288	0.01374	1.06667
slc	1.7%	0.00065	0.00166	0.00230	0.01863	0.02093	1.12368
slc2nd5M	1.5%	0.00054	0.00129	0.00183	0.01329	0.01512	1.13767
weaver	2.3%	0.00013	0.00045	0.00059	0.00781	0.00839	1.07511
weaver2nd5M	1.6%	0.00018	0.00050	0.00069	0.00916	0.00985	1.07482
devel	0.88%	0.00015	0.00023	0.00038	0.00263	0.00300	1.14393
prod5M	1.1%	0.00022	0.00060	0.00083	0.01106	0.01189	1.07476
prod2nd5M	1.1%	0.00033	0.00081	0.00114	0.01211	0.01325	1.09445
ATUM_MP	2.6%	0.00035	0.00099	0.00134	0.01913	0.02047	1.06994
ATUM_UP	2.9%	0.00026	0.00062	0.00089	0.01559	0.01648	1.05695
SPUR	1.4%	0.00032	0.00080	0.00112	0.01059	0.01171	1.10552
Synapse	1.0%	0.00024	0.00055	0.00078	0.00860	0.00938	1.09104
Average	2.0%	0.00029	0.00074	0.00103	0.01348	0.01451	1.07654

Table 4.4: Breakdown of Translation Miss Ratio

This table breaks down the miss ratio into its component parts. The first section of the table gives the results for each of the individual traces, separated into their respective groups. The second section of the table presents the averages for each group, and the final section gives the overall average. The first column lists the percentage of the cache consumed by pagetable entries, the next five columns show the various miss ratios for in-cache translation, and the final column presents the ratio of $m_{incache}$ and m_{ideal}.

is clearly a dominant factor, the increase in miss ratio due to translation, $m_{incache}/m_{ideal}$, indicates that other factors contribute to the variance. The increase averages 8%, but varies from under 5% to as high as 25%. We show in Section 4.3.3, that set conflicts in the direct-mapped cache explain this behavior.

Table 4.5 breaks down the cache misses, showing the miss ratio at each stage of translation. On average, 1.4% of the CPU references miss in the cache (including collision misses), ranging from the well-behaved *Devel* trace at 0.278%, to the *Mul2* trace at 2.47%. When a CPU reference misses, the cache controller finds the pagetable entry over 95% of the time, on average. m_{upte} ranges from a low of 2.43% upto a high of 8.53%; note that the highest values of m_{upte} correspond to the lowest values of m_{cpu}, meaning that the total number of first level pagetable misses is still low since the pagetable is only referenced on cache misses.

The root pagetable entry is the last step in the translation process. On average, these references miss 31% of the time, a much larger percentage than either CPU or first-level pagetable references. However, since root pagetable references occur infrequently, the misses are even more infrequent. Even for the worst trace, a root pagetable miss occurs only once for every 2000 CPU references.

Looking closer, while all four trace groups exhibit a higher m_{rpte} than m_{upte}, the increase is largest for the SPUR traces: these traces average 48% misses, while the other three groups average only 17%. We hypothesize that the high miss ratio for SPUR results from the root pagetables mapping to the low region of the virtual address cache, which tends to have a higher miss ratio than average. We examine this hypothesis further in Section 4.3.3.

At each level in the translation process, the fraction of references that miss increases: 1.4% of CPU references, 4.7% of first-level pagetable references, and 31% of root pagetable references. This is not surprising, considering the relative frequency of the different types of references. Between every first-level pagetable reference there are 75 CPU references, on average, and between every root pagetable reference there are over 1500 CPU references on average. It follows from the principle of temporal locality that the miss ratio increases as the inter-reference time increases. More specifically, between every root pagetable reference there are over 20 cache misses, on average, that tend to knock out the root pagetable entries of the cache. Thus it follows that higher levels of translation should have higher relative miss ratios.

This increasing trend validates the decision to terminate translation recursion at two levels in SPUR, rather than continuing down to the root* pagetable entry. This trend suggests that additional levels of translation would only slightly reduce the total number of misses, and decrease performance by increasing the miss penalty in some cases.

Despite the high marginal miss ratios, in-cache translation only increases the total cache miss ratio by .1%, on average, representing an 8% relative increase. Since the miss ratio is already low for the large cache, we expect that this small absolute increase has a small impact on system performance. The next section uses effective access time to confirm this hypothesis.

Trace	m_{cpu}	m_{upte}	m_{rpte}	$f_{writeback}$
mul2	0.02469	0.04254	0.13989	0.31931
mul8	0.02278	0.05171	0.08202	0.37021
savec	0.01099	0.03561	0.24069	0.33071
dec0	0.01350	0.02454	0.19293	0.32690
forf	0.01821	0.04060	0.15550	0.39510
rsim	0.00194	0.08531	0.79357	0.45016
rsim2nd5M	0.01317	0.02426	0.79332	0.18633
slc	0.01928	0.06164	0.39319	0.14524
slc2nd5M	0.01383	0.06445	0.44752	0.16788
weaver	0.00794	0.04022	0.42171	0.11709
weaver2nd5M	0.00934	0.03561	0.51436	0.12549
devel	0.00278	0.06821	0.18757	0.16628
prod5M	0.01129	0.04403	0.21288	0.19703
prod2nd5M	0.01244	0.05292	0.23456	0.21524
ATUM_MP	0.01948	0.04508	0.12688	0.34007
ATUM_UP	0.01585	0.03427	0.16606	0.36100
SPUR	0.01092	0.04911	0.48397	0.19870
Synapse	0.00884	0.05074	0.21993	0.19285
Average	0.01377	0.04706	0.30803	0.27316

Table 4.5: Miss Ratio at Each Level of Translation

This table shows the miss ratio at each level of translation. On average, one out of every 73 CPU references miss in the cache. Pagetable entry miss ratios are higher: there is one miss for every 21 first-level references, and one miss for every 3 second-level references. However, because we only access the pagetable *after* missing in the cache, first-level pagetable entry misses occur only once for every 1500 CPU references. Second-level pagetable misses occur only once every 5000 CPU references, on average.

4.2.2 Effective Access Time Analysis

To quantify in-cache's effect on system performance, we examine the effective access time of the cache, which weights cache hits and misses by the cycles each requires. Table 4.6 presents a breakdown of the effective access time: t_{ideal} is the effective access time of an ideal cache, t_{coll} is the time due to pagetable entries colliding with useful instructions and data, $t_{penalty}$ is the increase in the miss penalty required to access the first-level pagetable entries (we assume that $t_{upte-hit} = t_{cache}$, $t_{rpte-hit} = 2t_{cache}$, $t_{rpte-miss} = 3t_{cache}$), and finally, $t_{overhead}$ is the time required to handle pagetable entry misses.

As expected from the low value of m_{coll}, the time lost to collisions is very small: the effective access time increases by less than .01 cycles per reference in the worst case (*Slc*), a relative increase of only .4%. The increase in the miss penalty is more substantial: $t_{penalty}$ ranges from a low of .0018 to as high as .024 cycles per reference, with an average of .013. This represents an average increase of 1.1% in the effective access time, with a worst case of 1.7%. The final component, $t_{overhead}$, is also a significant factor. Misses to the pagetable entries add an average of .0098 cycles to the effective access time, ranging from as low as .0028 to as high as .022 cycles per reference. In relative terms, this factor increases the effective access time by 0.8% on average, and as much as 1.7% in the worst case.

Combining these factors yields the effective access time for in-cache translation, $t_{incache}$. For this set of traces, in-cache translation adds an average of .027 cycles to each reference, a relative increase of 2.3%. In the worst case, *Slc*, in-cache performs less than 4% worse than optimal. In other words, for a large set of programs, in-cache translation performs within 3% of an ideal translation scheme.

4.2.3 Comparison to Translation Lookaside Buffers

To place these results in perspective, we compare the performance of in-cache translation with conventional translation lookaside buffers. In two previous studies, Clark, et al, have used measurements of two implementations of the DEC VAX architecture to show that their translation lookaside buffers account for 4% and 6% of all cycles, respectively[20, 19]. Thus in-cache translation performs very favorably compared to these earlier results. Of course, these measurements were for very different workloads than our analysis, so we cannot make a direct comparison. In the rest of this section, we correct this deficiency by using our address traces to simulate a set of translation lookaside buffers.

Table 4.7 presents a sample of popular commercial machines and their respective translation lookaside buffer configurations. Table 4.8 presents the miss ratio results for the four workload groups over the range of these configurations. All the simulations assume 4-kilobyte pages, regardless of architecture; some configurations use a generalization of the IBM 3033 hash function to generate the index[77]:

$$index = a_{24}, a_{23}, a_{22}, a_{21}, a_{20}, a_{19}, a_{18}, a_{17}, a_{22} \oplus a_{16}, a_{21} \oplus a_{15}, a_{20} \oplus a_{14}, a_{19} \oplus a_{13}, a_{18} \oplus a_{12} \quad (4.19)$$

Trace	t_{ideal}	t_{coll}	$t_{penalty}$	$t_{overhead}$	$t_{incache}$	$\frac{t_{incache}}{t_{ideal}}$
mul2	1.38862	0.00671	0.02364	0.01671	1.43567	1.03388
mul8	1.37440	0.00687	0.02160	0.01669	1.41956	1.03285
savec	1.17400	0.00369	0.01060	0.00638	1.19466	1.01760
dec0	1.21468	0.00247	0.01316	0.00475	1.23506	1.01678
forf	1.30242	0.00632	0.01747	0.01196	1.33817	1.02745
rsim	1.03339	0.00271	0.00178	0.00335	1.04123	1.00758
rsim2nd5M	1.18523	0.00411	0.01285	0.00711	1.20931	1.02031
slc	1.25977	0.00905	0.01809	0.02166	1.30857	1.03873
slc2nd5M	1.18980	0.00771	0.01294	0.01658	1.22703	1.03129
weaver	1.10532	0.00179	0.00762	0.00619	1.12092	1.01412
weaver2nd5M	1.12453	0.00247	0.00901	0.00682	1.14282	1.01627
devel	1.03746	0.00218	0.00259	0.00282	1.04505	1.00732
prod5M	1.16064	0.00326	0.01079	0.00797	1.18265	1.01896
prod2nd5M	1.17902	0.00489	0.01178	0.01084	1.20654	1.02334
ATUM_MP	1.31234	0.00576	0.01861	0.01326	1.34996	1.02867
ATUM_UP	1.25855	0.00440	0.01532	0.00835	1.28661	1.02230
SPUR	1.14967	0.00464	0.01038	0.01029	1.17498	1.02201
Synapse	1.12571	0.00344	0.00839	0.00721	1.14475	1.01691
Average	1.21157	0.00456	0.01317	0.00978	1.23908	1.02270

Table 4.6: Effective Access Time Breakdown

This table breaks the effective access time into its main components: t_{ideal} is the effective access time of an ideal cache, t_{coll} is the time due to pagetable entries colliding with useful instructions and data, $t_{penalty}$ is the increase in the miss penalty required to access the first-level pagetable entries (we assume that $t_{upte-hit} = t_{cache}$, $t_{rpte-hit} = 2t_{cache}$, $t_{rpte-miss} = 3t_{cache}$), and finally, $t_{overhead}$ is the time required to handle pagetable entry misses.

Machine	Entries	Associativity	Page Size
DEC VAX 11/780	128	2	512
DEC VAX 8600	512	1	512
DEC VAX 8800	1024	1	512
IBM 370 3033	128	2	4096
IBM 370 3081	128	2	4096
MIPS R2000	64	64	4096
Motorola MC68030	22	22	256-32768
Intel i860	64	4	4096
AMD 29000	64	2	1024-8192
Intergraph Clipper	128	2	4K

Table 4.7: Translation Lookaside Buffer Configurations for Commercial Machines

This table lists some typical translation lookaside buffer configurations. These buffers often have additional features that affect their performance. For example, the VAX buffers are split in half: half for user pages and half for system pages. The VAX must flush the user half on context switches. Several of the machines use an address space identifier to eliminate flushing the buffer on every context switch. The IBM buffers hash the index to spread the pagetable entries more uniformly across the buffer.

TLB Configuration			ATUM_MP	ATUM_UP	SPUR	Synapse	Average
Size	Assoc	Hash					
512	1	no	0.00282	0.00207	0.00326	0.00519	0.00333
1024	1	no	0.00205	0.00160	0.00302	0.00184	0.00213
4096	1	no	0.00190	0.00128	0.00260	0.00007	0.00146
64	2	no	0.00679	0.00592	0.00358	0.00364	0.00498
128	2	no	0.00329	0.00265	0.00198	0.00252	0.00261
256	2	no	0.00182	0.00136	0.00112	0.00149	0.00145
128	2	yes	0.00282	0.00277	0.00188	0.00090	0.00209
256	2	yes	0.00171	0.00116	0.00120	0.00052	0.00115
64	64	no	0.00502	0.00512	0.00143	0.00135	0.00323
128	128	no	0.00270	0.00236	0.00058	0.00063	0.00157
256	256	no	0.00153	0.00111	0.00020	0.00024	0.00077
In-cache Translation			0.00134	0.00089	0.00112	0.00078	0.00103

Table 4.8: Miss Ratios of Translation Lookaside Buffers

This table compares the miss ratios of 11 typical translation lookaside buffer configurations to in-cache address translation. The average for each group of traces is presented separately, along with the overall average. In-cache has a lower miss ratio for 38 of the 44 group averages; the 6 cases where a translation lookaside buffer has the lower miss ratio are printed in boxed type.

where $\oplus$ is the exclusive-or operator. Hashing the index helps spread references more uniformly across the buffer, reducing the frequency of collisions. We have annotated the multiprogramming traces with process ids to avoid flushing the translation lookaside buffer on context switches.

Our results confirm previous studies showing that both size and associativity significantly affect the miss ratio[71, 20, 5]. The miss ratio decreases as the size of the buffer increases, approaching the infinite buffer miss ratio[5] for the largest ones. Increasing the associativity from direct-mapped to two-way set-associative has an enormous impact: a two-way set-associative buffer with 256 entries is better than a direct-mapped buffer with 1024 entries, on average. Hashing the index has a large effect for some traces, but nearly none on others. The Synapse traces improve significantly; probably because of its segmented address space. The Synapse N+1 divides its address space into 16 one-megabyte segments. Without hashing, the first page of each segment maps to the same line in small buffers. The hash function distributes these pagetable entries across the entire buffer. Although SPUR also has a segmented address space, it does not gain much from this particular hashing function, because the function does not include the bits containing the global segment number. We expect that a more appropriate hash function would provide some improvement for the SPUR traces.

Table 4.8 compares the miss ratios of the translation lookaside buffers to in-cache translation, with the cases where the translation buffers perform better printed in boxed type. The results show that in-cache performs at least as well in 38 of the 44 group averages. Looking at individual traces, in-cache performs better in 121 out of 154 simulations. Only the largest of the fully-associative buffers is better on average, by 25%, while in-cache is as much as 480% better than some of the other buffer configurations. These results clearly show that, based on the miss ratio metric, in-cache translation performs superior to translation lookaside buffers.

But the miss ratio is not a direct measure of performance, and hence can be a misleading indicator. If we assume that the cache miss penalty is the same for both in-cache translation and translation lookaside buffers, and that the translation buffer miss penalty is the same as the cache miss penalty, then the conclusions drawn from the miss ratio results remain correct. But these two assumptions are not necessarily valid. For two commercial machines, the translation lookaside buffer miss penalty is 20-25 cycles [20, 19], rather than the 12 cycles we assume for a cache miss. And while a very aggressive implementation of in-cache translation might achieve a cache miss penalty equal to a cache with a translation lookaside buffer (by overlapping the pagetable entry reference with bus arbitration, for example), a more typical implementation would take one or more cycles to access the pagetable entry.

The effective access time takes these differences into account. Using the earlier assumption that $t_{upte-hit} = t_{cache}$, $t_{rpte-hit} = 2t_{cache}$, $t_{rpte-miss} = 3t_{cache}$, and assuming a 20 cycle translation lookaside buffer miss penalty, we can compare a good implementation of in-cache

[5]The miss ratio of a translation lookaside buffer with infinite capacity.

TLB Configuration								
Size	Assoc	Hash	ATUM_MP	ATUM_UP	SPUR	Synapse	Average	$r_{breakeven}$
512	1	no	1.36875	1.29986	1.21486	1.22943	1.27823	0.96937
1024	1	no	1.35331	1.29049	1.21008	1.16246	1.25408	0.98803
4096	1	no	1.35024	1.28410	1.20168	1.12721	1.24081	0.99860
64	2	no	1.44806	1.37685	1.22132	1.19854	1.31119	0.94500
128	2	no	1.37814	1.31148	1.18922	1.17612	1.26374	0.98048
256	2	no	1.34877	1.28584	1.17209	1.15550	1.24055	0.99881
128	2	yes	1.36883	1.31396	1.18730	1.14365	1.25343	0.98854
256	2	yes	1.34651	1.28173	1.17371	1.13607	1.23451	1.00370
64	64	no	1.41274	1.36094	1.17834	1.15274	1.27619	0.97092
128	128	no	1.36642	1.30574	1.16127	1.13833	1.24294	0.99689
256	256	no	1.34297	1.28070	1.15363	1.13046	1.22694	1.00989
In-cache Translation			1.34996	1.28661	1.17498	1.14475	1.23908	1.00000

Table 4.9: Comparison to Effective Access Time

translation to a good translation lookaside buffer implementation. The results, summarized in Table 4.9, show that the extra cycle in-cache needs to look for the pagetable entry has a significant effect. While in-cache was superior in 38 of the 44 miss ratio cases, it is superior in only 28 of the 44 effective access time cases. However, in 12 of the 16 other cases, the performance difference is less than 1%, and never exceeds 2%. Looking at individual traces, in-cache translation performs better in 102 of the 154 simulations, or 66%. At the extremes, in-cache is at most 3% worse and as much as 11% better. For 17 of the 154 simulations, in-cache's total performance is at least 5% better. Finally, only 2 translation lookaside buffer configurations have a lower effective access time, averaged over all traces.

To include the cycle time effects, we calculate $r_{breakeven}$ for each translation lookaside buffer configuration. As shown in the last column of Table 4.9, 9 of the 11 configurations require a *faster* cycle time than in-cache translation to achieve equal performance. Since virtual address caches nearly always have access times faster than or equal to physical address caches, in-cache is a better choice than these configurations for all technologies. For the other configurations, $r_{breakeven}$ has a maximum value of 1.01. If the translation lookaside buffer slows the cycle time by more than 1%, then in-cache translation has superior performance.

For example, the MIPS R2000 translation lookaside buffer contains 64 entries with a fully-associative organization and random replacement[6]. Our simulations show that for the

[6]Eight entries in the R2000 can be wired-down, but we ignore that feature in this discussion.

SPUR cache configuration, the MIPS R2000 cycle time would have to be 3% faster with the translation lookaside buffer than with in-cache. Since the buffer is the critical path of this chip[22], eliminating the buffer would actually decrease the cycle time. This example presents a clear case where a commercial microprocessor could improve performance by using in-cache translation.

4.3 Sensitivity Analysis

The results of the baseline analysis clearly depend upon the particular parameters of the SPUR cache. To generalize the analysis, we must explore a larger region of the design space. We do this by testing the sensitivity of the baseline results to the SPUR parameter values. We vary each of the major memory system parameters, and compare the performance of in-cache translation to three typical translation lookaside buffer configurations. The cache parameters are cache size, block size, associativity, page size, and miss penalty. The three translation lookaside buffer configurations, denoted as *tlb64-64*, *tlb128-2*, and *tlb1024-1*, represent a diverse sample of typical configurations: 64 entry fully-associative, 128 entry two-way set-associative, and 1024 entry direct-mapped. The 64 entry buffer uses a random replacement policy and the set-associative buffer uses a least-recently-used policy.

We show that in-cache translation has superior performance to these translation lookaside buffer organizations over a wide range of cache configurations. The translation lookaside buffers perform better only when the cache size is 32 kilobytes or less, or the miss penalty is much greater than the 12 cycles of our baseline configuration. These results show that in-cache address translation is an attractive alternative to translation lookaside buffers for many applications.

4.3.1 Effects of Cache Size

The performance of in-cache translation depends heavily upon the basic cache miss ratio m_{ideal}. Consider the two extremes: a cache that always misses and a cache that never misses. In the first case, in-cache has terrible performance: every CPU reference requires 3 memory references. In the second case, since CPU references never miss, in-cache translation never occurs and hence does not degrade performance. Obviously, neither extreme is practical, but this example suggests that in-cache translation performs best for caches with low miss ratios.

Since cache size is a major factor in determining a cache's miss ratio, intuition suggests that the performance of in-cache translation should improve as the cache size increases. Figure 4.1 confirms this hypothesis: the translation miss ratio starts high for small cache sizes, then decreases rapidly as the cache size increases. As the cache becomes larger, the miss ratio decreases less rapidly. This decreasing trend mimics the frequently observed behavior of the basic cache miss ratio m_{ideal}. This trend may represent an intrinsic decrease in the marginal improvement; however, Singh, Stone, and Theibaut hypothesize that cold-

start misses actually explain this behavior, and that simulations require much longer traces to accurately model steady-state miss ratios for large caches [74].

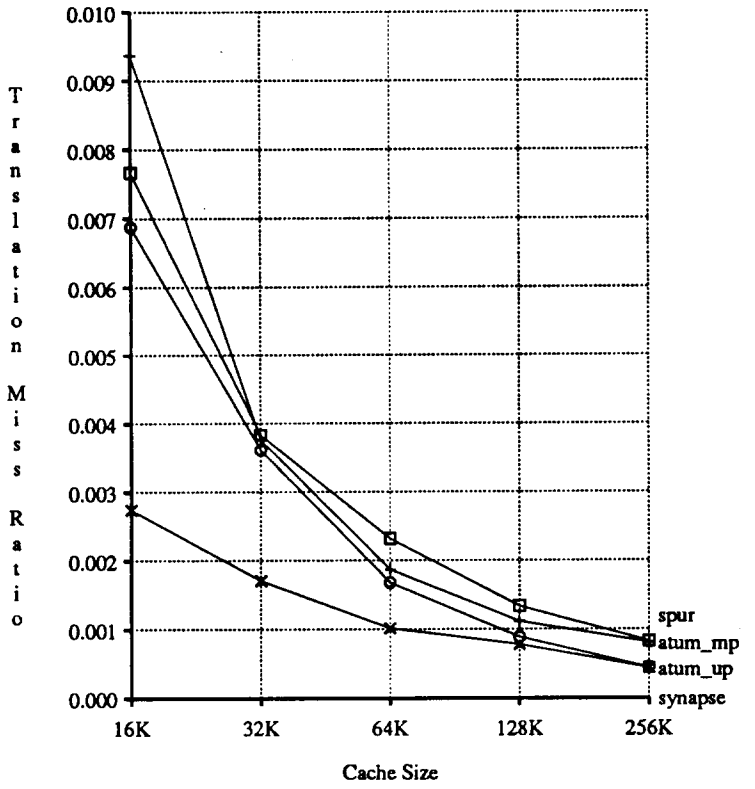


Figure 4.1: Effect of Cache Size on m_{trans}

This figure plots the in-cache translation miss ratio m_{trans} against the cache size. The curves represent the average of each group of traces. The ATUM_MP curve is marked with a □ symbol, the ATUM_UP curve with a ○, the SPUR curve with a +, and the Synapse curve with a ×. All four curves show a declining trend as the cache size increases. The cache is direct-mapped with 32-byte blocks.

Since in-cache translation's performance is so sensitive to cache size, it is clear that below some minimum size (for given associativity and block size) in-cache performs worse than a translation lookaside buffer. Figure 4.2(a) illustrates the thresholds for the selected buffer configurations. Based on miss ratios, the cross-over point for *tlb64-64* lies at a cache size of 32 kilobytes, and for the other two the cross-over lies between 32 and 64 kilobytes. Thus for caches of 64 kilobytes and larger, in-cache translation has a lower miss ratio than these standard translation buffer configurations.

A specific threshold depends not only upon the translation lookaside buffer configuration,

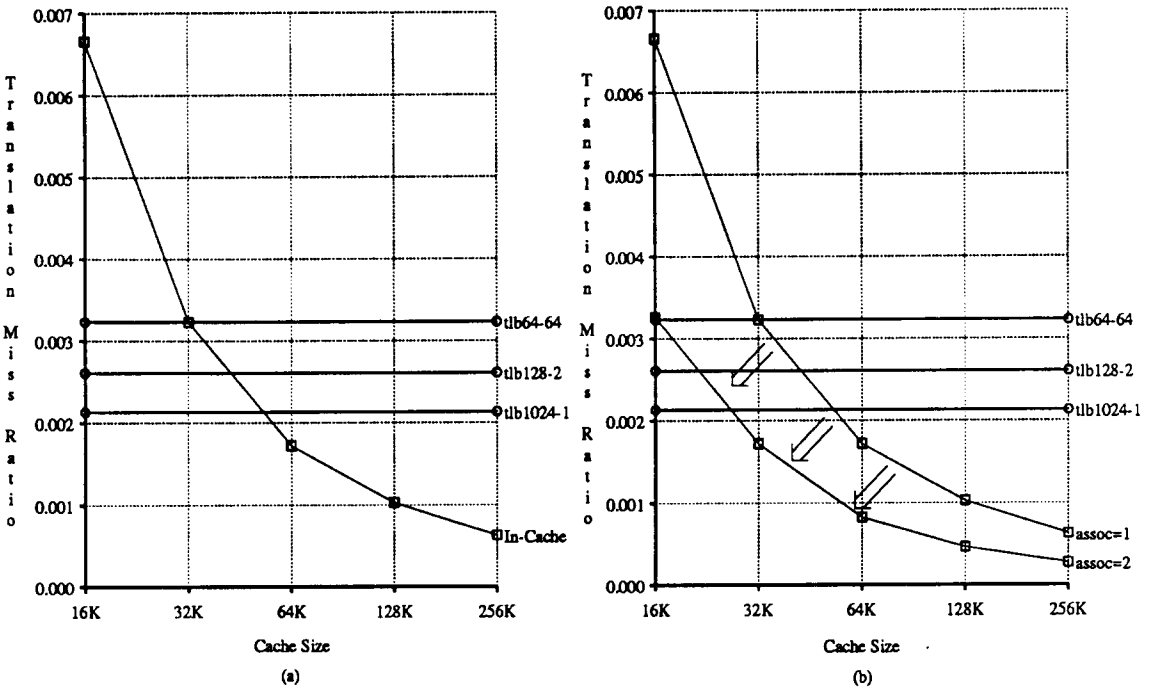


Figure 4.2: Comparison of m_{trans} to m_{tlb} for Varying Cache Sizes

Graph (a) compares m_{trans} for a direct-mapped cache to m_{tlb} for the three translation lookaside buffer configurations. The miss ratio for in-cache translation equals that for *tlb64-64* for a 32-kilobyte cache, and is better than *tlb1024-1* and *tlb128-2* for a 64-kilobyte cache. In-cache translation performs better for larger caches.

Graph (b) shows the reduction in m_{trans} as the associativity increases. The in-cache translation miss ratio decreases by roughly a factor of two, regardless of cache size. This in turn shifts the point where in-cache has the lower miss ratio: a two-way set-associative cache of 32 kilobyes or larger has better performance than all three translation lookaside buffer configurations. The block size is 32 bytes for both graphs.

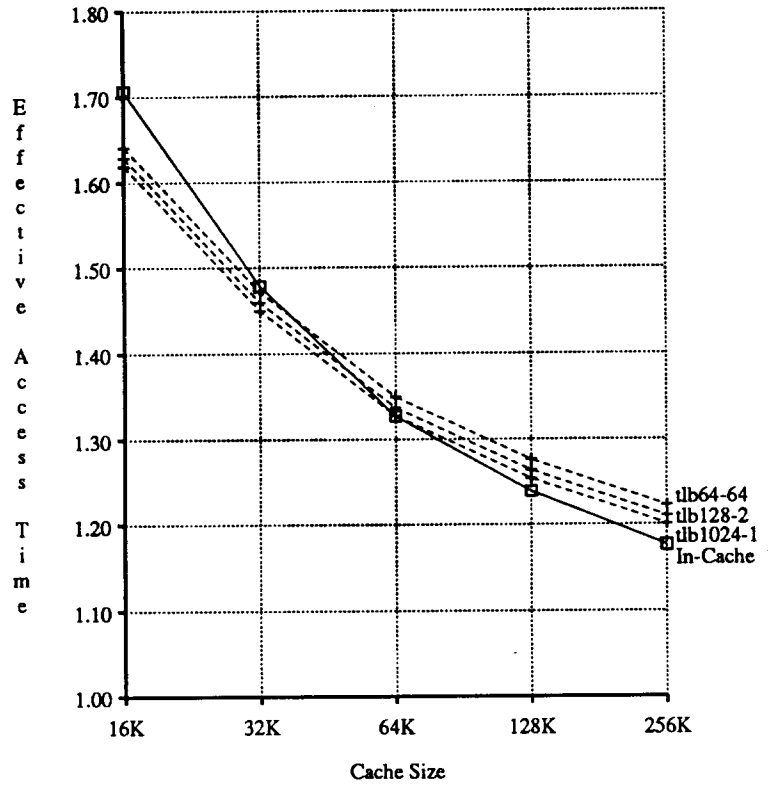


Figure 4.3: Effect of Cache Size on $t_{incache}$ and t_{tlb}

This figure compares the effective access time of in-cache address translation, $t_{incache}$, against three representative translation lookaside buffers. As the cache size increases, the performance of in-cache translation improves relative to the buffers, and performs better than all three when the cache contains at least 64 kilobytes. The cache has 32-byte blocks and is direct-mapped.

but upon the cache parameters as well. In particular, increasing the cache's associativity from direct-mapped to 2-way set-associative significantly reduces both the total miss ratio $m_{incache}$ and the in-cache translation miss ratio m_{trans}. As shown in Figure 4.2(b), increasing the associativity reduces m_{trans} by a factor of two, regardless of cache size, shifting the entire curve towards the origin. This in turn shifts the cross-over points down by a factor of two. Thus with a two-way set-associative cache, the miss ratio for in-cache translation is better than our selected translation lookaside buffers whenever the cache size is 32 kilobytes or larger.

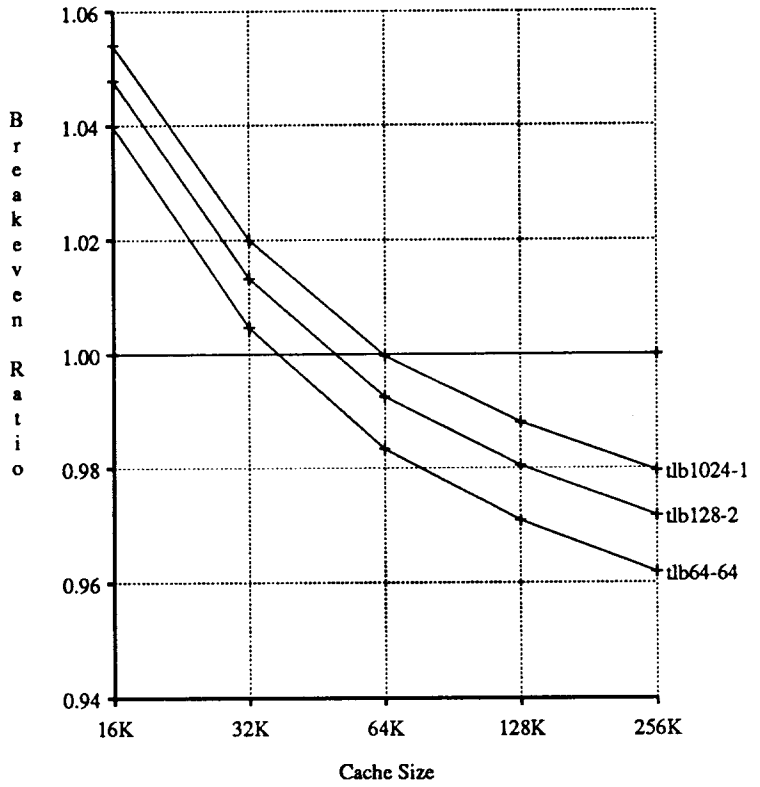


Figure 4.4: Effect of Cache Size on $r_{breakeven}$.

This figure plots $r_{breakeven}$ against the cache size for the three translation lookaside buffer configurations. This figure shows that if the translation lookaside buffer degrades the cache access time by more than 5.4%, then in-cache has the best performance for even the smallest cache size.

Now we consider effective access time. As we saw in the baseline analysis, the increase in the basic miss penalty works against in-cache translation. This implies that the cross-over points will be shifted out towards larger caches. Figure 4.3 illustrates this clearly: the cross-over for the worst performing translation lookaside buffer, *tlb64-64*, occurs between

32 and 64 kilobytes, rather than exactly at 32 kilobytes using miss ratios. The threshold remains essentially the same: in-cache performs better for caches containing at least 64 kilobytes. Considering individual traces, in-cache translation performs better in 33% of the combinations at 64 kilobytes, and 95% at 256 kilobytes.

Figure 4.4 plots $r_{breakeven}$ against cache size for the three translation lookaside buffer configurations. By definition, the intersection of these curves with unity correspond to the cross-over points of Figure 4.3. This graph shows that even at the smallest cache size, in-cache translation performs within 5.4% of the best translation lookaside buffer. In other words, if the translation lookaside buffer degrades the cache access time by more than 5.4%, then in-cache translation has superior performance even for the smallest cache size. Conversely, at the largest cache size, the best translation lookaside buffer requires a 2% faster cycle time than in-cache translation to achieve equal performance.

In summary, these simulation results show that the performance of in-cache translation is very sensitive to the cache size. For small caches, the interference between pagetable entries and instructions and data degrades performance below that of translation lookaside buffer configurations. But the interference decreases for larger cache sizes, improving both the miss ratio and effective access time. For caches larger than 32 kilobytes, the effective access time is better for in-cache translation, and never more than 4% worse than optimal. In addition, cold-start behavior from the ATUM traces, discussed at length in Appendix A, tends to over-estimate the miss ratios for large caches. Eliminating this error from our results should make in-cache translation even more attractive.

4.3.2 Effects of Block Size

The cache block size also has a major impact on miss ratio. In this section we compare the effect of this parameter on in-cache translation to its effect on an ideal cache. We use an ideal cache for comparison, since block size does not affect translation lookaside buffer performance[7]. We show that in-cache achieves its best performance for small block sizes. We present two hypotheses for this behavior, and show that the second hypothesis explains the observation. Finally, we use effective access time to show that the optimal block size is the same in both an ideal cache and in a cache with in-cache translation.

Figure 4.5 shows that for each trace group, the average cache miss ratio m_{ideal} declines as the blocksize increases. At larger block sizes, the miss ratio declines less rapidly; if we continue to increase the block size, the miss ratio eventually increases as memory pollution sets in[77, 80].

To understand this behavior, we must remember why caches work. Caches exploit two types of reference locality: *spatial locality* means that because the processor has referenced a particular word, it is likely to reference a neighboring word, *temporal locality* means that because the processor has referenced a particular word, it is likely to reference it again.

[7]We have made the simplifying assumption that translation lookaside buffer misses bypass the cache.

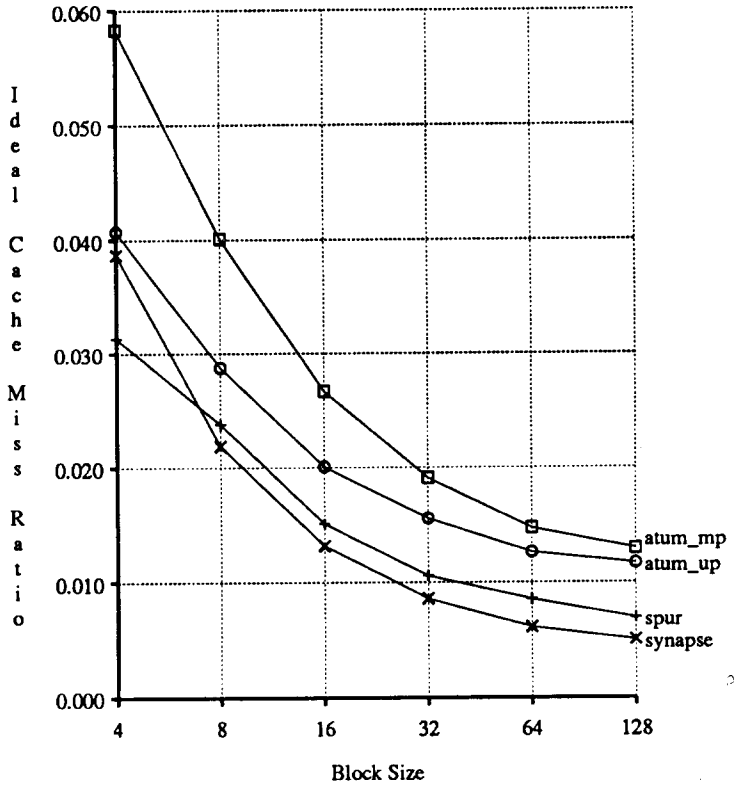


Figure 4.5: Effect of Block Size on m_{ideal} by Trace Group

This graph plots the average miss ratio for the 4 trace groups as the block size increases. The cache is 128 kilobytes and direct-mapped.

Block Size	Fully Associative Cache			Direct-Mapped Cache	
	m_{ideal}	m_{trans}	$\frac{m_{trans}}{m_{ideal}}$	m_{trans} Conflicts	% Conflicts
4	0.02456	0.00036	0.01479	0.00070	66%
8	0.01579	0.00036	0.02308	0.00066	64%
16	0.01005	0.00036	0.03558	0.00063	64%
32	0.00701	0.00035	0.04976	0.00068	66%
64	0.00504	0.00024	0.04799	0.00106	81%
128	0.00384	0.00023	0.05982	0.00130	85%

Table 4.10: Effect of Increasing Block Size on m_{trans}

This table shows the effects of increasing block size in both fully-associative and direct-mapped caches. The conflict component of m_{trans} is defined as $m_{trans}(dm) - m_{trans}(fa)$. The "% Conflicts" column lists the fraction of m_{trans} due to conflict misses in a direct-mapped cache. This percentage increases as the block size increases, accounting for 85% of all translation misses at 128-byte blocks.

By increasing the block size, we exploit spatial locality at the expense of temporal locality: larger blocks bring in more neighboring data, but because there are fewer total blocks, each block stays in cache for a shorter length of time, on average. When the blocks become too large, the cache loses more from temporal locality than in gains from spatial locality: the large blocks "pollute" the cache and the miss ratio begins to increase. However, we do not observe this behavior in Figure 4.5 because the SPUR cache is large enough that the pollution point occurs above 128 byte blocks.

Figure 4.6 illustrates how increasing the block size affects in-cache address translation. The lower curve plots the ideal cache miss ratio, m_{ideal}, the upper curve plots the total miss ratio for in-cache translation, $m_{incache}$. Both curves exhibit the expected declining trend, but $m_{incache}$ does not decline quite as rapidly for larger blocks. While the absolute difference between the miss ratios is small, the relative difference is much larger: $m_{incache}/m_{ideal}$ is 1.025 for 4-byte blocks, but 1.167 for 128-byte blocks. This result clearly shows that in-cache translation works best for small block sizes.

To examine this behavior in more detail, Figure 4.7(a) plots m_{trans} for each of the 4 trace groups. All 4 groups show an increase in the miss ratio for larger block sizes. The SPUR traces show the greatest increase, nearly doubling between 32- and 64-byte blocks. As shown in Figure 4.7(b), the *Rsim2nd5M* trace accounts for this behavior, dominating the average of the SPUR traces. The other SPUR traces show only a slight increase in m_{trans} with increasing block size.

We have two hypotheses to explain this sensitivity to block size. First, pagetable en-

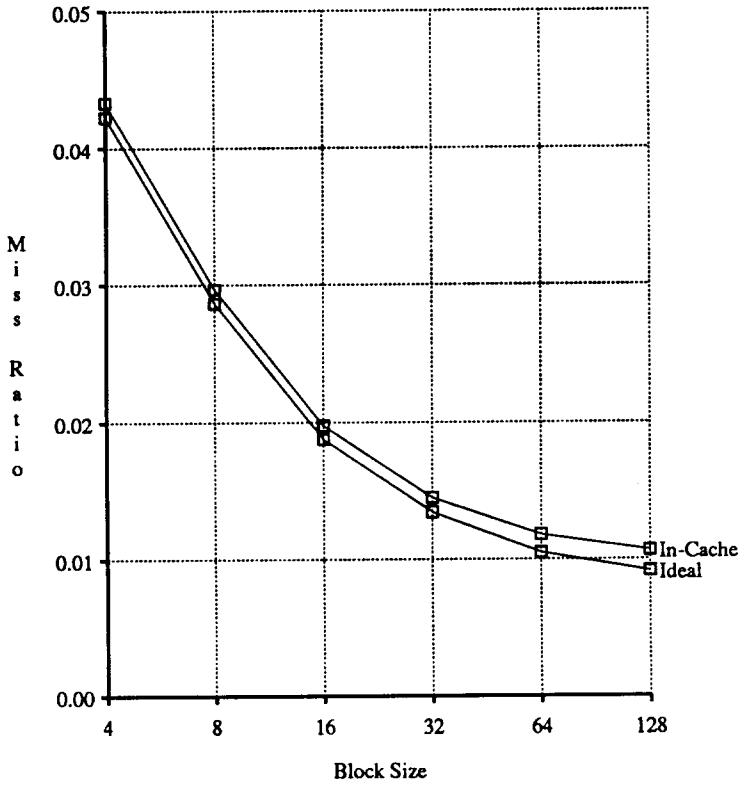


Figure 4.6: Effect of Block Size on m_{ideal} and $m_{incache}$

This figure plots the average of m_{ideal} and $m_{incache}$ for all 14 traces as the block size increases. Both curves show the expected declining trend, but $m_{incache}$ does not decrease as rapidly as m_{ideal}. The cache size is 128 kilobytes and direct-mapped.

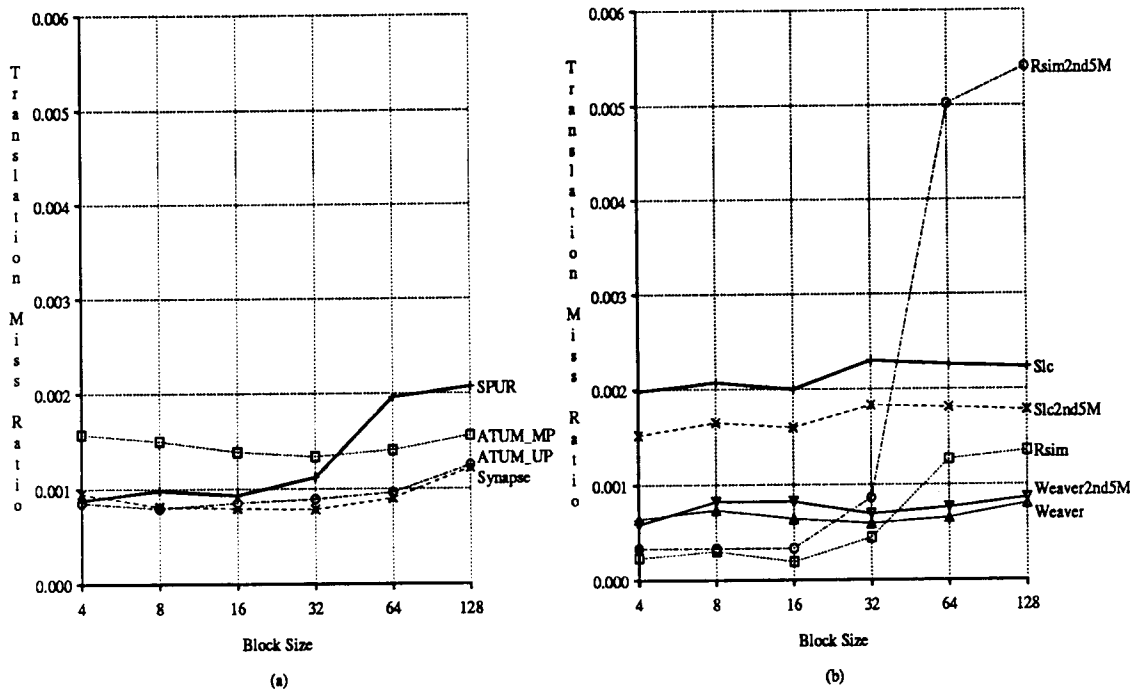


Figure 4.7: Effect of Block Size on m_{trans}

Graph (a) plots the average of m_{trans} against the block size for the 4 trace groups. All 4 groups show an increase in the miss ratio as the block size increases; however, the SPUR traces show a marked jump between 32- and 64-byte blocks.

Graph (b) shows that most of the increase for SPUR is caused by a single trace, *Rsim2nd5M*, which biases the average miss ratio for the SPUR traces. For both graphs, the cache is 128 kilobytes and direct-mapped.

tries benefit more from temporal locality than from spatial locality. This follows from the observation that referencing a page doesn't tell us much about the probability that we will reference a neighboring page in the future, since pages are so large. On the other hand, the fact that we reference a page strongly suggests that we will reference it again. Extending this argument to pagetable entries suggests that the translation miss ratio benefits primarily from temporal locality, and hence works best for smaller blocks.

The second hypothesis attributes the increase in the translation miss ratio to collisions in the direct-mapped cache. As the blocksize increases, more of the misses result from set-conflicts. We expect this to be particularly severe for the SPUR architecture because the pagetable entries for page 0 of each segment map to the same set in the cache.

To test whether the locality hypothesis explains this phenomenon, we examine the effect of increasing the block size in a fully associative cache. Since a fully-associative cache has only one set, it eliminates all set-conflict misses. Thus, if the ratio m_{trans}/m_{ideal} increases as the block size increases, then pagetable entries do not benefit from spatial locality as much as instructions and data. In addition, if m_{trans} increases as the block size increases, then this behavior explains the rise m_{trans} for set-associative (and direct-mapped) caches as well. Table 4.10 summarizes the results from the fully-associative cache simulations. On average, the ratio m_{trans}/m_{ideal} increases with increasing block size, thus the hypothesized behavior does occur and partially explains why $m_{incache}$ and m_{ideal} diverge. However, since m_{trans} decreases, this result does not explain the increase in m_{trans} for the direct-mapped cache.

Our alternative hypothesis claims that set-conflict misses in the direct-mapped cache cause the increase in m_{trans}. Hill defines the set-conflict component of the miss ratio as the miss ratio of a cache minus the miss ratio of a fully-associative cache with the same capacity and block size[42]. Applying this definition to translation misses, we show in Table 4.10 that set conflicts increase with the block size. Since m_{ideal} decreases, the relative importance of the set-conflict misses increases rapidly: set conflicts cause 66% of translation misses for 4 byte blocks, increasing to 85% for 128 byte blocks. This data shows that set-conflict misses account for the increase in m_{trans} for direct-mapped caches.

Increasing the cache from direct-mapped to two-way set-associative significantly decreases the number of set-conflict misses. As shown in Figure 4.8, the increased associativity eliminates the radical jump in m_{trans} observed earlier. This is true not only for the SPUR traces, but for the other trace groups as well.

Increasing the block size also increases the miss penalty for a fixed-width bus, so we must examine the effective access time as well. Figure 4.9 plots both the ideal effective access time, t_{ideal}, and the in-cache translation effective access time, $t_{incache}$. Both curves exhibit the classic bathtub shape: as the block size becomes too large the decline in miss ratio is overshadowed by the increase in miss penalty. Both with and without in-cache translation, the optimal block size is 16 bytes.

While the block size is an important parameter of cache performance, in-cache translation has almost no effect on this design decision. For 13 of the 14 traces, the minimum

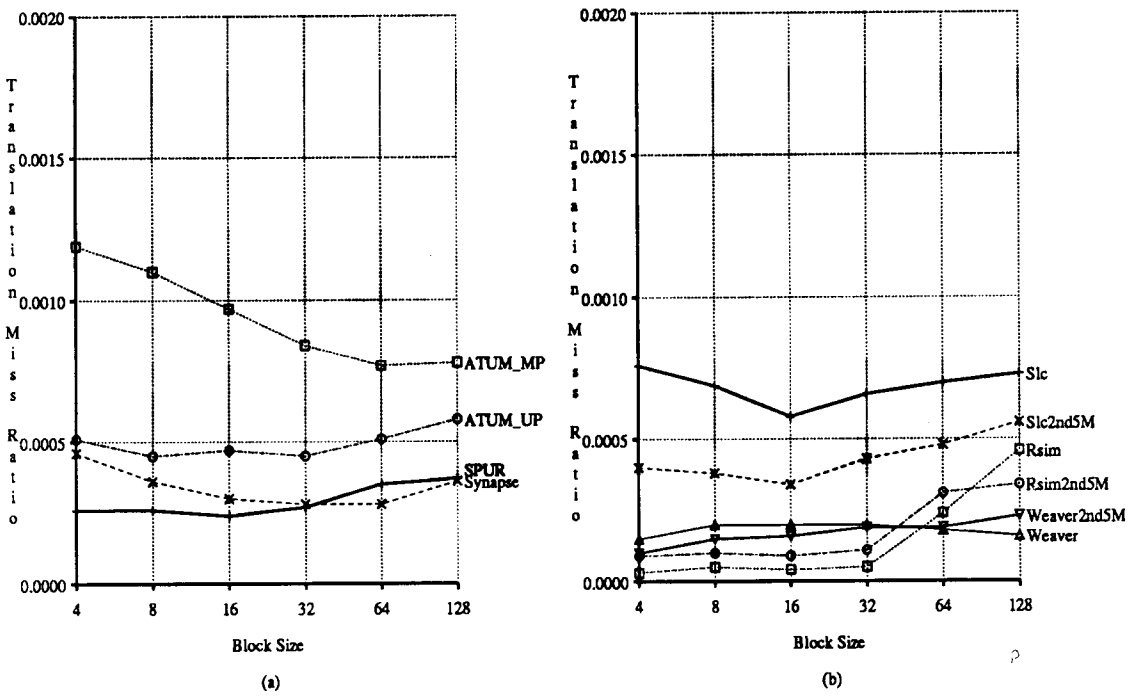


Figure 4.8: Effect of Block Size on m_{trans} for a 2-Way Set Associative Cache

Graph (a) plots m_{trans} for the 4 trace groups, and graph (b) plots the 6 SPUR traces. Comparing these graphs to Figure 4.7 shows that increasing the associativity to 2-way smoothes the erratic behavior of m_{trans} substantially.

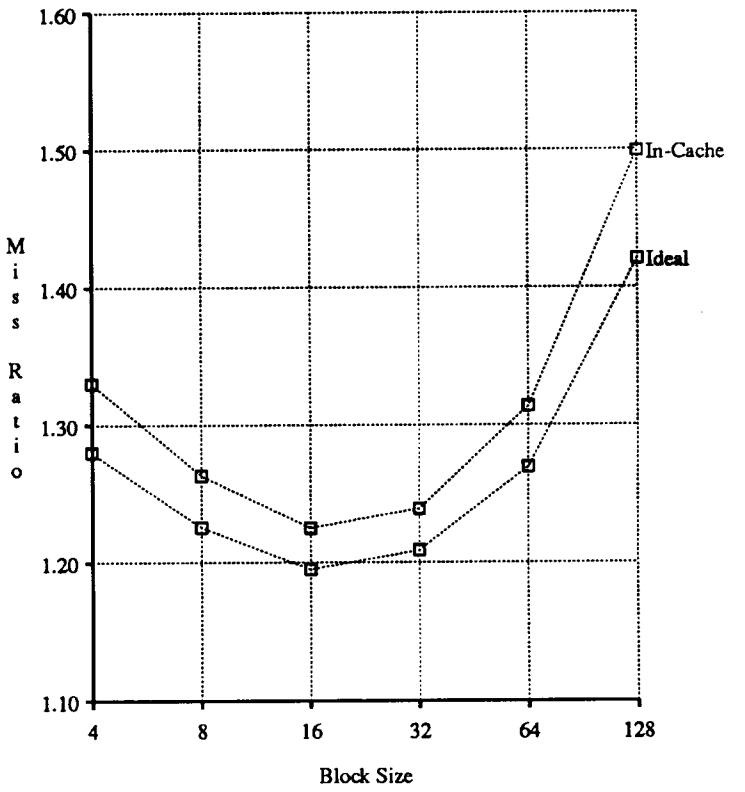


Figure 4.9: Effect of Block Size on Effective Access Time

This graph compares $t_{incache}$ to t_{ideal} as the block size changes. The optimal block size is 16 bytes for both, although 32-byte blocks are within 1%.

values of t_{ideal} and $t_{incache}$ occur for the same block size. For 9 of the 14 traces, the optimal block size for in-cache translation is 16 bytes, and for the other 5 it is 32 bytes. On average, the 32-byte block performance is within 1% of the 16 byte block performance. Thus, for a cache of this size and associativity, either 16 or 32 byte blocks would be a good choice.

4.3.3 Effects of Associativity

In the preceding sections, we have already seen that the degree of associativity is an important cache parameter. In this section, we examine the effect of associativity on in-cache translation. We show that because pagetable entries increase contention for the cache, in-cache translation benefits more from increased associativity than an ideal cache. We also show that set conflicts cause the anomalously high values of m_{rpte} observed for the SPUR traces, and show that shifting the location of the pagetable in the virtual address space eliminates this anomalous behavior.

Increasing the associativity for a given cache size generally reduces the miss ratio, leading to the obvious, but incorrect, conclusion that higher associativity is always better than lower associativity. However, as mentioned earlier, Hill and Przybylski have shown that this conclusion is incorrect [42, 67], because t_{cache} and t_{memory} are not independent of the associativity. In his dissertation, Hill shows that for a typical AS TTL design with CMOS static RAMs (based on the SPUR implementation), the cycle time is 9% slower for a two-way set-associative cache than for a direct-mapped cache. Using our simulations, the average miss ratios, m_{ideal}, for the direct-mapped and 2-way set-associative caches are 1.35% and 0.88% respectively. The effective access times, t_{ideal}, are 1.21 and 1.14 cycles per reference respectively. Since this 6% cycle count improvement is less than the 9% cycle time degradation, increasing the associativity actually degrades total performance by 3%. Hill and Przybylski's results show that higher degrees of associativity are not a panacea, and that detailed implementation information is necessary for a complete analysis. Nonetheless, there are many situations where increasing the associativity is the correct choice. For example, the cycle time of the second-level cache in a hierarchy is less critical, and decreasing the miss ratio becomes more important [42].

Because pagetable entries add another independent reference stream that competes for space in the cache, we might suspect that conflict misses account for a larger fraction of the total misses under in-cache translation than in an ideal cache. If this is the case, then increasing the associativity should result in a larger improvement in $m_{incache}$ than in m_{ideal}. Figure 4.10 plots the ratios of miss ratios of a direct-mapped cache over a set-associative cache, for m_{ideal}, $m_{incache}$, and m_{trans}, with the associativity increasing from 2-way to fully-associative. As predicted, in-cache translation benefits more from associativity: $\frac{m_{trans(dm)}}{m_{trans(sa)}}$ increases more rapidly than $\frac{m_{ideal(dm)}}{m_{ideal(sa)}}$. The drop in the ratio of m_{trans} for fully-associative caches occurs because of an increase in collision misses; the decrease in the cache's effective capacity caused by the pagetable entries is most significant for fully-associative caches. Increasing the associativity from direct-mapped to 2-way decreases m_{ideal} by 35%, while

m_{trans} decreases by over 50%. The greater relative reduction in m_{trans} means that in-cache translation benefits more from higher associativity than an ideal cache; however, the absolute difference is small. The miss ratio $m_{incache}$ drops from 1.45% to 0.93%, and the effective access time drops from 1.24 to 1.16 cycles per reference. But, this is still only a 6% drop, so when we include the 9% cycle time degradation, the direct-mapped cache still has the best performance.

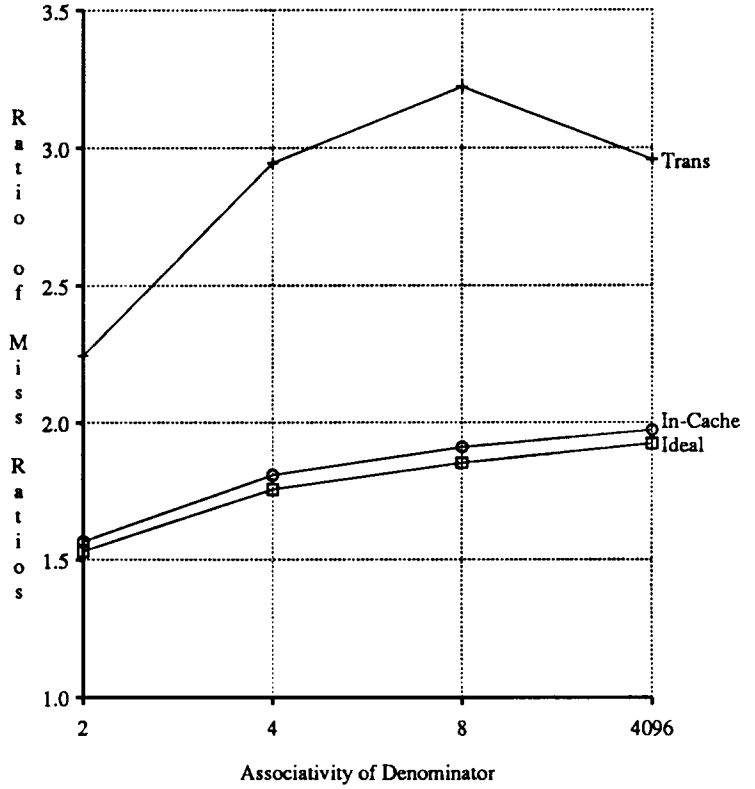


Figure 4.10: Ratio of Miss Ratios: Direct-Mapped Over Set-Associative

This graph plots the ratio of miss ratios of a direct-mapped cache over a cache with associativity ranging from 2-way to fully-associative. The ratios of m_{trans}, $m_{incache}$, and m_{ideal} are each plotted separately. The ratio of m_{trans} is much greater than the ratio of m_{ideal}, indicating that in-cache translation gains more from increased associativity than an ideal cache.

Although direct-mapped caches often have the best *average* performance, they are more likely to exhibit anomalous behavior. The problems arise when two or more frequently accessed blocks map to the same cache set. Since only one of these blocks can reside in the cache at a time, frequent misses result. For example, McFarling has shown that repositioning a program's code in memory significantly affects the miss ratio of a direct-mapped

instruction buffer [58]. His results show that direct-mapped caches are very sensitive to the placement of code and data in the cache.

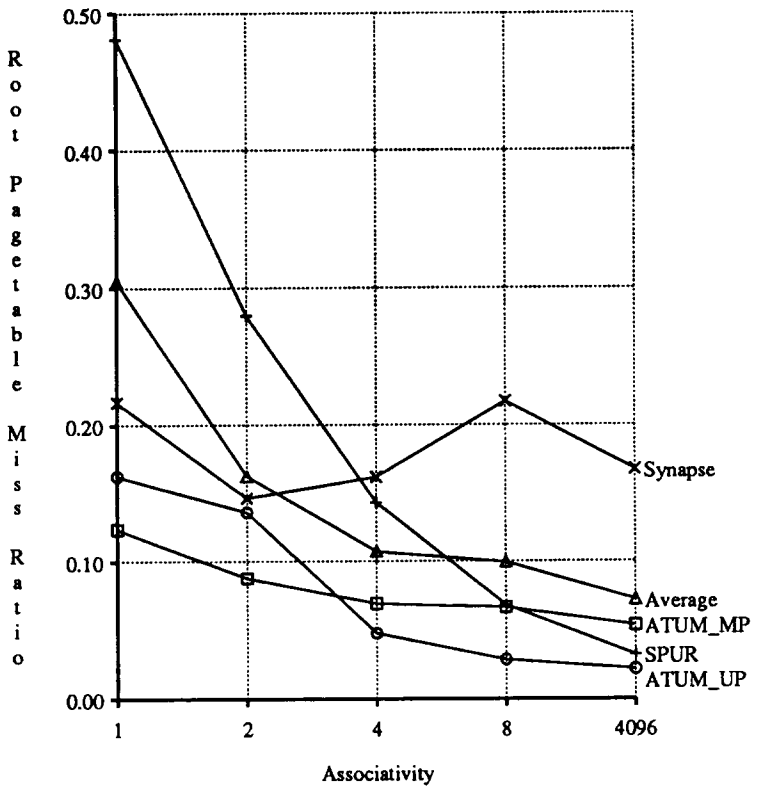


Figure 4.11: Effects of Increasing Associativity on m_{rpte}

This graph plots the effect of increasing associativity on m_{rpte} for the 4 trace groups. The SPUR traces improve dramatically, as hypothesized. The last point on the x-axis is full associativity, not 16-way set-associative.

In the baseline analysis we observed that the miss ratio for root pagetable entries, m_{rpte}, was very high, particularly for the SPUR traces. We hypothesize that conflict misses occur more frequently for SPUR because of a coincidental mapping of the root pagetables to the bottom of the cache. Since the cache is indexed with virtual addresses, the bottom of the cache tends to be more heavily utilized than other regions. If this hypothesis is correct, then increasing the associativity should have a greater effect on m_{rpte} than on either m_{upte} or m_{cpu}. Figure 4.11 shows that m_{rpte} drops significantly for all 4 trace groups when the associativity is increased to two-way, but decreases substantially more for the SPUR traces. For a fully-associative cache, m_{rpte} is worse for the ATUM_MP traces than for the SPUR traces. Thus the high value of m_{rpte} for the SPUR traces is a result of set conflicts in the direct-mapped cache.

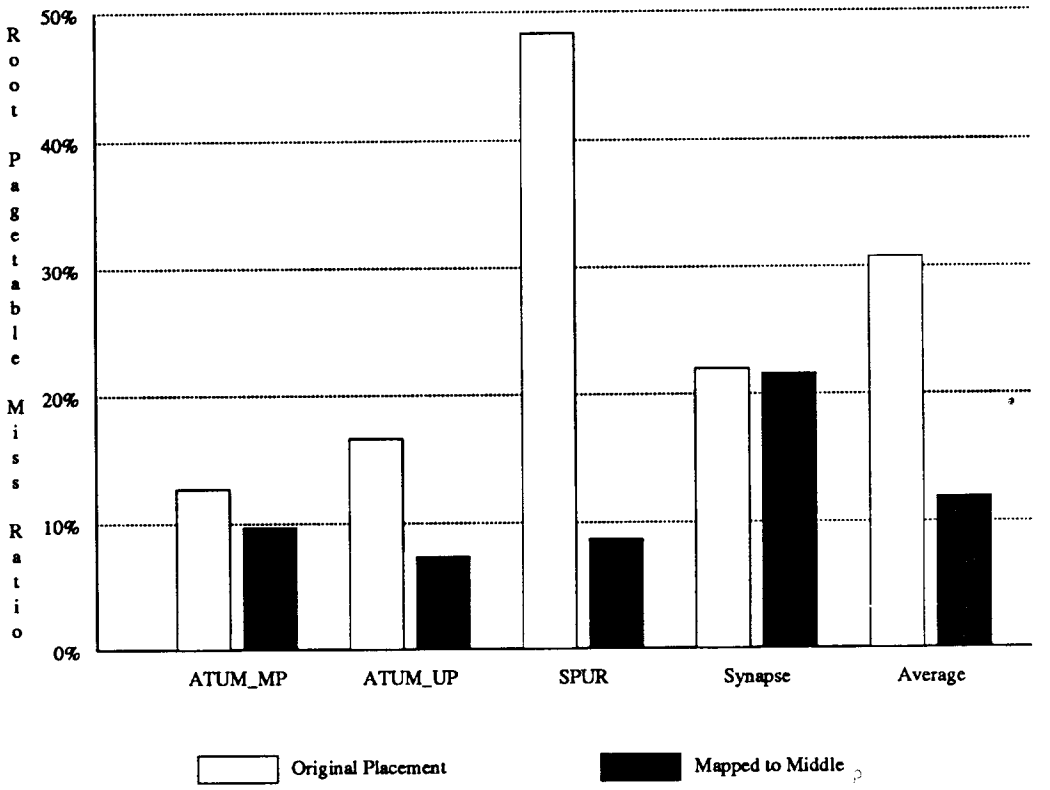


Figure 4.12: Effect of Pagetable Placement on m_{rpte}

This bar chart shows the sensitivity of the SPUR traces to the placement of the root pagetable entries in the cache. Normally, the pagetable is aligned on a 256-megabyte boundary in the virtual address space, causing the root pagetable entries for global segments 0–3 to map to the bottom of the cache. In this experiment, we added a 64-kilobyte offset to the pagetable base, causing the root pagetable entries to map to the middle of the cache. This change reduces m_{rpte} for the SPUR traces from 48% to only 8%.

We can further confirm this hypothesis by changing the position of the root pagetables in the cache. Figure 4.12 shows the effect of shifting the root pagetables so they map to the middle of the the cache, rather than the bottom. All 4 trace groups improve, but the SPUR traces improve much more: dropping from 48% to 8.6% misses.

The location of the active root pagetable entries in the cache depends upon the global segment number of the active segments. In the simulations, we use the identity mapping (process segment N, $N \in \{0,1,2,3\}$ is assigned global segment N), which causes the root pagetable entries to map to the first page of the cache. In a real system, the operating system arbitrarily assigns global segment numbers, which spreads the root pagetable entries more widely across the cache, reducing the frequency of the misses. We re-examine this issue in Chapter 6.

4.3.4 Effects of Page Size

In this section, we examine the effects of page size on in-cache address translation and our three representative translation lookaside buffer configurations. We show that in-cache translation is much less sensitive to this parameter than translation lookaside buffers. We also observe that m_{trans} behaves erratically as the page size increases, and does not decline monotonically as expected. We show that set-conflicts in the direct-mapped cache explain this behavior.

Translation lookaside buffer performance is very sensitive to page size[20, 69], as illustrated in Figure 4.13 for the trace *Mul2*, which is representative of all the traces. The miss ratio for *tlb128-2* is 1.3% for 512-byte pages, but drops to 0.15% for 16-kilobyte pages. This is nearly an order of magnitude difference in miss ratio over the range of page sizes.

Since translation lookaside buffers are so sensitive to the page size, we might assume that in-cache translation would have similar behavior. However, as shown in Figure 4.14(a), the translation miss ratio m_{trans} exhibits a more moderate decline. The *Mul2* trace decreases by only a factor of 5 over the entire range. Figure 4.14(b) shows that on average the decline is even less: only the ATUM_MP traces average more than a factor of 3 difference between the maximum and minimum miss ratios. The other trace groups average less than a factor of 2 difference.

What makes in-cache translation behave differently from a translation lookaside buffer in this respect? Clark and Emer hypothesize that translation lookaside buffers are sensitive to page size because most buffers map only a limited portion of the total virtual address space. For example, a 128-entry buffer with 512-byte pages maps at most 64 kilobytes of address space; a very small amount considering modern workstations often have over 16 megabytes of physical memory. Increasing the page size to a more generous 4 kilobytes increases the maximum mapped space to 512 kilobytes, effectively increasing the size of the buffer. More precisely, doubling the page size is identical to doubling the translation

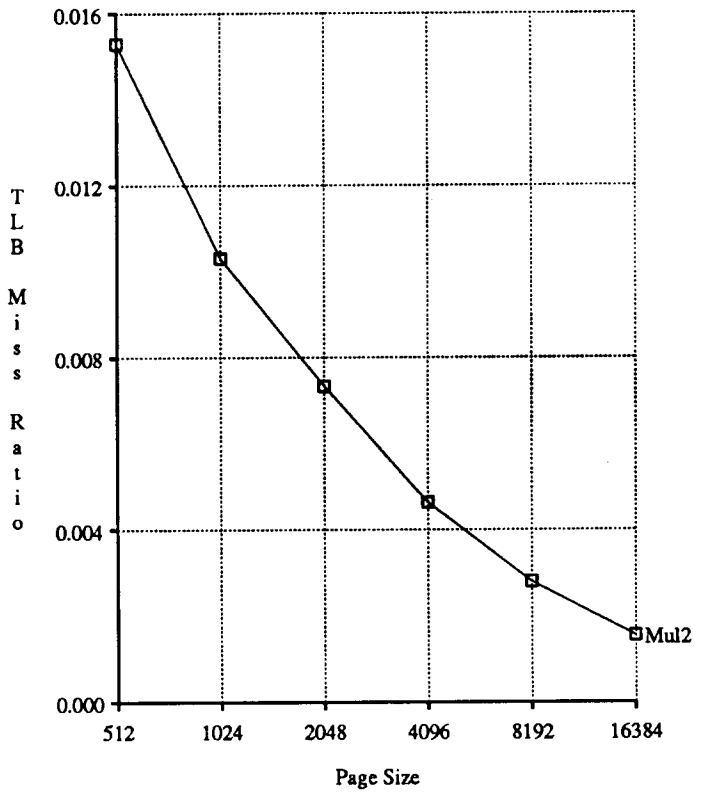


Figure 4.13: Effects of Page Size on m_{tlb}

This figure plots m_{tlb} for the configuration *tlb128-2* (128 entries, 2-way set-associative) for the trace *Mul2*. The miss ratio declines rapidly as the page size increases.

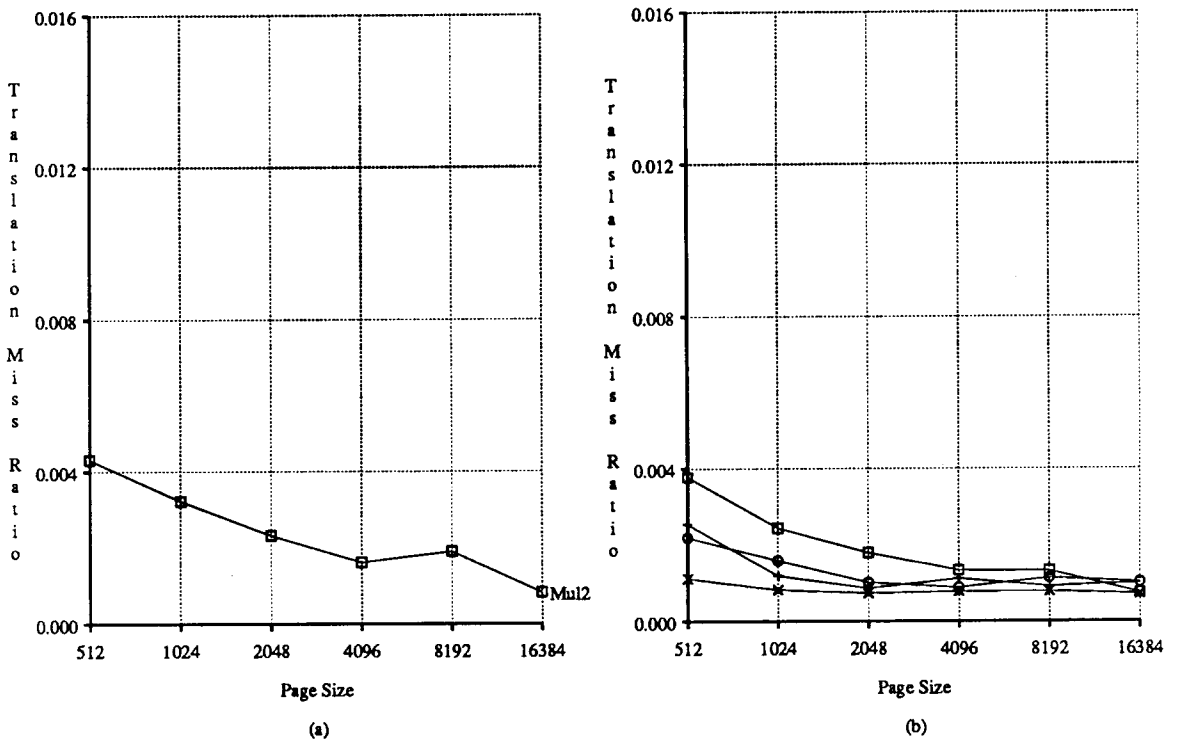


Figure 4.14: Effect of Page Size on m_{trans}

This figure plots the effects of page size on in-cache translation. Graph (a) plots the *Mul2* trace for direct comparison to Figure 4.13, graph (b) plots the averages of the 4 trace groups. The ATUM_MP traces are denoted by a □ symbol, ATUM_UP by a ∘, SPUR by a +, and Synapse by a ×.

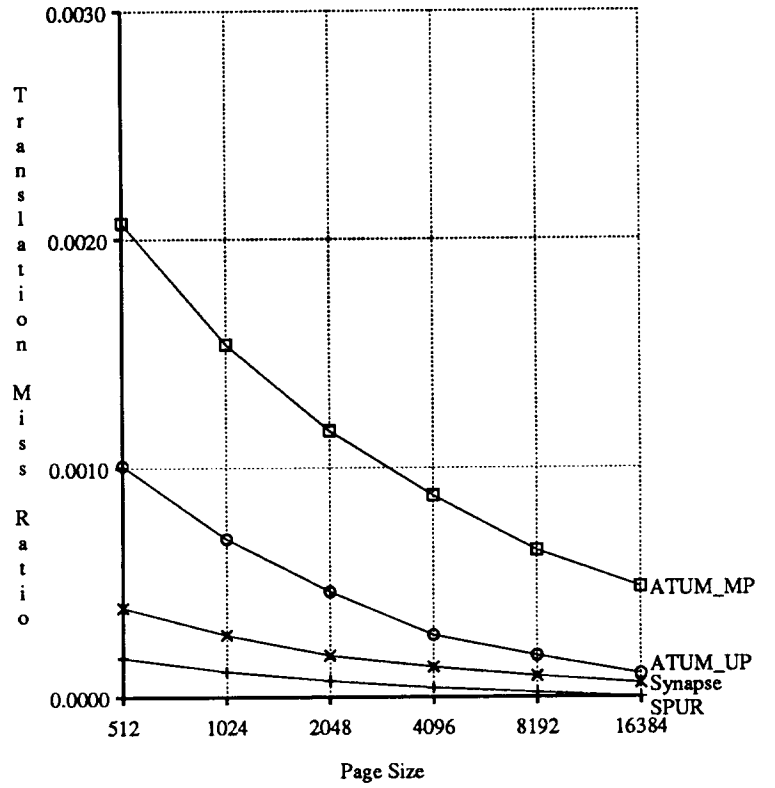


Figure 4.15: Effect of Page Size on m_{trans} for a Fully-Associative Cache

This graph plots the effect of page size on in-cache translation for a fully-associative cache. Since m_{trans} declines monotonically, the erratic behavior observed in a direct-mapped cache is due to set-conflict misses.

Page Size	Pagetable Entries	Blocks of Pagetable Entries	Percent of Cache
512	1440	180	4.4%
1024	1186	148	3.6%
2048	900	113	2.7%
4096	644	81	2.0%
8192	415	52	1.3%
16384	337	42	1.0%

Table 4.11: Fraction of Cache Consumed by Pagetable Entries.

This table shows the fraction of the cache consumed by pagetable entries for different page sizes. The data presented here is the average number of pagetable entries, the average number of blocks containing pagetable entries, and the percent of the cache consumed by pagetable entries. For 512-byte pages, 4.4% of the cache contains pagetable entries. This decreases to 1.0% for 16-kilobyte pages.

lookaside buffer capacity and block size[8].

But increasing the page size only slightly increases the effective capacity of the cache under in-cache translation, since only a small fraction of the cache actually contains pagetable entries. Table 4.11 shows that for 512-byte pages only 4.4% of the cache holds pagetable entries, while for 16-kilobyte pages the fraction decreases to 1.0%. Thus increasing the page size 32 times only increases the effective capacity of the cache by 3.4%, resulting in only a small decrease in the miss ratio.

Although the translation miss ratio declines slowly as the page size increases, it exhibits some erratic behavior. Rather than a monotonic decline, as we see with the translation lookaside buffers, there are upward variations. While the magnitude of these fluctuations is small, with a negligible impact on total system performance, we must understand such anomalous behavior to insure that the simulations are not in error.

We hypothesize that this behavior results from set-conflicts in the direct-mapped cache. As the page size increases, the pagetable size decreases because each pagetable entry maps more of the virtual address space. This means that at different page sizes the pagetable entries for particular data structures map to different cache sets. If by increasing the page size, a frequently accessed pagetable entry maps to a cache set that incurs more set conflicts, then the translation miss ratio will increase.

[8] Although translation lookaside buffers almost always have a block size of 1 in practice, there is no conceptual limitation

To test this hypothesis, we vary the page size for a fully-associative cache. Since a fully-associative cache has only one set, by definition it cannot have set-conflict misses. As shown in Figure 4.15, the translation miss ratio declines monotonically. This is true not only for the group averages, but for the individual traces as well. This result proves that set-conflict misses in the direct-mapped cache cause this erratic behavior.

4.3.5 Effects of Miss Penalty

In the previous sections, we assumed a particular value for the miss penalty when computing the effective access time. In this section, we examine the sensitivity of our results to this assumption, and varying the main memory access time, translation lookaside buffer miss penalty, and in-cache translation overhead.

Memory Access Time

Our baseline system requires 12 cycles to read a block from or write a block to main memory: 4 cycles of latency and 8 cycles transfer time (one word per cycle). Several architectural and implementation factors can affect this timing. Increasing the bus width from 32 bits to 64 bits reduces the transfer time in half, decreasing the memory access time to 8 cycles. Slower memory, or a faster cycle time[9], can increase the best-case latency by a factor of two or more. In addition, multi-stage interconnection networks provide higher bandwidth than busses, but at the cost of increased latency. Thus memory access time may reasonably vary from a factor of 2 lower to a factor of 5 greater than our default value.

The performance of an ideal cache is very sensitive to memory access time. We compute the effective access time as a function of $t_{readblock}$, the time to read a block from memory, by plugging in the miss ratio results for the SPUR cache configuration (see Table 4.4) into the Equation 4.2. Assuming that $t_{writeback} = t_{readblock}$ produces the function:

$$t_{ideal} = 1.0 + 0.0172 t_{readblock} \tag{4.20}$$

The slope of this function is 0.0172, meaning that for each cycle $t_{readblock}$ increases (decreases) about the baseline improves (degrades) the effective access time by 0.0172 cycles per reference, or approximately 1.5%.

Plugging the miss ratio results into Equation 4.15 produces the effective access time for in-cache translation as a function of $t_{readblock}$:

$$t_{incache} = 1.0146 + 0.0186 t_{readblock} \tag{4.21}$$

In-cache is slightly more sensitive to memory access time than an ideal cache, because the translation misses increase the slope of the function. In addition, the overhead of address

[9]Memory latency is a function of the dynamic RAM access time. A system with a faster cycle time spends more cycles waiting for memory.

translation increases the y-intercept above unity, guaranteeing that $t_{incache}$ and t_{ideal} never intersect.

Of course, an ideal cache is not a fair basis for comparison. Instead, we compute t_{tlb} for our three canonical translation lookaside buffer organizations, by substituting miss ratio results from Table 4.8 into Equation 4.5:

$$t_{tlb}(\textit{tlb64-64}) = 1.0646 + 0.0172t_{readblock} \quad (4.22)$$

$$t_{tlb}(\textit{tlb128-2}) = 1.0522 + 0.0172t_{readblock} \quad (4.23)$$

$$t_{tlb}(\textit{tlb1024-1}) = 1.0426 + 0.0172t_{readblock} \quad (4.24)$$

We continue to make the optimistic assumption that translation lookaside buffer misses are handled "ideally", i.e., that they neither pollute the cache nor generate references to memory. This assumption means that all three of these functions have the same slope as our ideal cache. Since the y-intercepts are greater than for in-cache translation, these functions must intersect with $t_{incache}$.

This results show that above a breakeven memory latency, translation lookaside buffers have a lower effective access time. However, the intersection points for our three buffer organizations are 34.5 cycles, 25.9 cycles, and 19.3 cycles, for *tlb64-64*, *tlb128-2*, and *tlb1024-1*, respectively. Thus the memory access time must be quite large for translation lookaside buffers to have superior performance. In addition, as we show next, our computation of t_{tlb} is overly optimistic; a more accurate performance model would push the breakeven points out to much longer memory access times.

Translation Lookaside Buffer Miss Penalty

In our computation of t_{tlb}, we assume that the translation buffer miss penalty, $t_{tlbmiss}$, is a constant 20 cycles. More importantly, we assume that it does not interfere with the cache, nor generate any additional memory references. These assumptions are clearly optimistic; most real systems allow pagetable entries to reside in the cache to reduce $t_{tlbmiss}$. In these systems, the handling of a translation lookaside buffer miss may cause a regular cache miss, potentially colliding with instructions and data.

To examine the effect of this assumption, we examine the cache performance *after* a translation lookaside buffer. We assume that when the translation lookaside buffer misses, we use the in-cache translation algorithm to reload it, making the comparison as fair as possible. We denote these new miss ratios with the prime ($'$) symbol to prevent confusion; for example, m'_{upte} is the probability that a pagetable entry resides in the cache after a translation lookaside buffer miss. The translation lookaside buffer configuration is 128 entries and two-way set-associative (*tlb128-2*).

Table 4.12 shows that the miss ratio for pagetable entries is much higher than regular instructions and data: m'_{upte} averages 30% over all the traces, ranging from a low of 17% up to a high of 70%. This is much higher than m_{upte} for in-cache translation, averaging 4.7%,

Trace	Cache after Translation Lookaside Buffer					In-Cache	
	m'_{upte}	m'_{rpte}	m'_{pte}	m'_{coll}	m'_{trans}	m_{trans}	$\frac{m'_{trans}}{m_{trans}}$
mul2	0.32715	0.14303	0.00092	0.00030	0.00122	0.00162	0.75393
mul8	0.37597	0.08388	0.00102	0.00026	0.00128	0.00168	0.75675
savec	0.39962	0.33014	0.00027	0.00014	0.00041	0.00071	0.57642
dec0	0.17617	0.08671	0.00020	0.00007	0.00027	0.00055	0.50120
forf	0.29591	0.16449	0.00058	0.00024	0.00082	0.00123	0.66931
rsim	0.69880	0.43621	0.00019	0.00009	0.00028	0.00044	0.62466
rsim2nd5M	0.36765	0.64000	0.00027	0.00015	0.00043	0.00086	0.49606
slc	0.45955	0.27703	0.00147	0.00046	0.00193	0.00230	0.83672
slc2nd5M	0.46002	0.33596	0.00124	0.00038	0.00162	0.00183	0.88596
weaver	0.37233	0.46081	0.00045	0.00011	0.00056	0.00059	0.95633
weaver2nd5M	0.49791	0.53574	0.00057	0.00006	0.00063	0.00069	0.91551
devel	0.15735	0.43200	0.00020	0.00007	0.00027	0.00038	0.71530
prod5M	0.19972	0.46519	0.00041	0.00009	0.00050	0.00083	0.60229
prod2nd5M	0.17180	0.47834	0.00055	0.00013	0.00068	0.00114	0.59660
ATUM_MP	0.35731	0.13167	0.00074	0.00023	0.00097	0.00134	0.72356
ATUM_UP	0.25376	0.14548	0.00039	0.00016	0.00055	0.00089	0.61740
SPUR	0.45363	0.37131	0.00070	0.00021	0.00091	0.00112	0.81104
Synapse	0.17773	0.46558	0.00039	0.00010	0.00048	0.00078	0.61771
Average	0.30016	0.34175	0.00055	0.00017	0.00073	0.00103	0.70434

Table 4.12: Translation Miss Ratio *After* a Translation Lookaside Buffer

This table shows the translation miss ratio after a translation lookaside buffer (*tlb128-2*). We resolve translation lookaside buffer misses by invoking in-cache address translation to find the pagetable entry. Although the buffer filters the reference stream, the cache still has over 70% as many translation misses as in-cache translation.

but is not surprising since the translation lookaside buffer filters out many of the references. The root pagetable miss ratios are much closer: m'_{rpte} averages 34%, while m_{rpte} averages 31%.

The most interesting result is m'_{trans}, the number of pagetable entry misses in the cache normalized to the number of CPU references. As shown in Table 4.12, this averages 0.073%, over 70% as large as m_{trans} for in-cache translation. The ratio of $m'_{trans}/mtrans$ ranges from a low of 49.6% to as high as 95.6%. Thus even through this translation lookaside buffer filters out most of the pagetable references, the cache has at least 50% as many pagetable misses as in-cache translation. These additional misses generate significant additional memory traffic and overhead, making our model for t_{tlb} overly optimistic. A more accurate model, which makes t_{tlb} depend upon $t_{readblock}$, shifts the point of equal performance to a much larger memory access time. Thus in-cache translation has superior performance over a larger range of the design space.

In-Cache Translation Overhead

The performance of in-cache translation also depends upon the overhead required when it accesses pagetable entries. We break this down as the overhead to find a first-level pagetable entry in the cache, $t_{upte-hit}$, to miss on the first-level pagetable entry but hit on the second-level entry, $t_{rpte-hit}$, and to miss on both the first- and second-level entries, $t_{rpte-miss}$. We compute $t_{incache}$ in terms of these three parameters:

$$t_{incache} = 1.225 + 0.0131 t_{upte-hit} \quad (4.25)$$

$$t_{incache} = 1.237 + 0.00045 t_{rpte-hit} \quad (4.26)$$

$$t_{incache} = 1.237 + 0.00020 t_{rpte-miss} \quad (4.27)$$

where we again assume the SPUR cache configuration.

These three equations show that $t_{incache}$ is quite sensitive to $t_{upte-hit}$ and quite insensitive to the other parameters. This follows because we incur $t_{upte-hit}$ whenever a CPU reference misses, while the other penalties occur only when we miss on a first- or second-level pagetable entry. Equation 4.25 shows that every cycle required to find the pagetable entry when it's in the cache increases the effective access time by 1.1%. Thus in an aggressive implementation of in-cache translation, a designer should try to minimize $t_{upte-hit}$, so long as it does not increase the cycle time by more than 1%. We discuss the trade-off between cycles and cycle time further in Chapter 5.

Equations 4.26 and 4.27 show that $t_{incache}$ is insensitive to either $t_{rpte-hit}$ or $t_{rpte-miss}$: even if $t_{rpte-hit}$ were 20 cycles and $t_{rpte-miss}$ were 40, the effective access time would increase by only 1.2%. This suggests that we could simplify the implementation by trapping to software to handle these infrequent cases, as done in the VMP system at Stanford[17][10]. However, while VMP traps to software on all cache misses, our results show that this is

[10]To prevent deadlock, VMP provides a separate local memory for the trap handler code and data.

much too expensive for the SPUR cache configuration, because of the sensitivity to $t_{upte-hit}$. Instead, a compromise between VMP and SPUR appears attractive, where only data and instruction misses are handled in hardware, and pagetable entry misses trap to software. Cheriton proposes a similar compromise, where a subset of "simple" misses are handled in hardware[16]. Further work is needed to investigate these alternatives.

4.4 Summary

In this chapter we have used trace-driven simulation to analyze the performance of in-cache address translation and compare it to translation lookaside buffers. When the results have not exhibited the expected behavior, we have formulated alternative hypotheses and tested them with further experiments. Using this experimental approach, we have shown that in-cache translation performs better than many translation lookaside buffers for a wide range of cache configurations.

For the SPUR cache configuration, 128 kilobytes direct-mapped with 32-byte blocks for any trace, in-cache translation consumes at most 3.4% of the cycles, and averages only 2.3%. This compares favorably with our translation lookaside buffer simulations, which as a group average 3.8% of the cycles, and the worst buffer averages over 8% of the cycles. Of the 11 buffer configurations, only the largest fully-associative and set associative buffers perform better on average. These results also compare favorably with previously published results, which show that the translation buffers in two implementations of the VAX consume 4% and 6% of the cycles on average[20, 19].

Taken alone, these performance gains are small. But a virtual address cache can reduce the cache access time by as much as 50%, or eliminate an extra stage from the execution pipeline. If we include these implementation dependent factors, it is clear that in-cache translation has significantly better than some common translation lookaside buffer configurations.

In testing the sensitivity of these results, we have shown that in-cache performs best for large caches. For direct-mapped caches of 32 kilobytes and less, the interference between pagetable entries and instructions and data degrades performance below that of typical translation lookaside buffers. However, the increasing density of integrated circuits make large caches both feasible and desirable, making in-cache translation attractive for many implementations.

The performance of in-cache is less sensitive to the other parameters. In-cache translation performs slightly better for small block sizes, but this does not affect the choice of the optimal block size. It benefits more from increased associativity than a conventional cache, but again the difference is too small to influence this design decision. The page size does not affect in-cache translation as much as it affects translation lookaside buffers. Increasing the page size increases the effective capacity of the buffer substantially, but since pagetable entries consume only 4.4% of the cache even with only 512-byte pages, this change has only a minor impact on the effective capacity of the cache.

Finally, the performance of both conventional caches and in-cache translation is very sensitive to the memory access time. Because in-cache translation generates more cache misses, a translation lookaside buffer performs better with longer memory access times. However, the cross-over points are large, and in-cache translation remains superior for most reasonable memory systems,

In summary, we have shown that in-cache translation is an attractive alternative to conventional translation lookaside buffers. Extensive trace-driven simulations demonstrate that in-cache translation performs better than the translation lookaside buffers implemented in most commercial machines.

Chapter 5

Implementation

Software researchers have traditionally taken an experimental approach. They evaluate new features, algorithms, and designs by constructing prototype implementations and measuring their performance. Prototype implementation followed by rigorous evaluation uncovers incorrect assumptions, design errors, and unexpected behavior. Learning from these mistakes leads to deeper understandings of the interplay between design and implementation, frequently leading to improved methods and improved designs.

Conversely, academics implement new computer architectures much less frequently. Implementing a new architecture entails building hardware: requiring a more extensive effort than writing software because of the lower-level design primitives and longer fabrication cycle. Detailed design and debugging requires expensive design tools and test equipment, in addition to the cost of implementation itself. And while software systems lend themselves to successive refinement, usually hardware must be fully functional the first time. These factors make the design-analyze-prototype-evaluate loop longer for hardware, making it harder to revise designs in a timely fashion.

Despite the cost and commitment of time, the prototyping process is just as important for hardware designs as for software systems. Building a prototype forces designers to consider implementation and technology trade-offs often ignored in design-only projects. Designers often uncover many mistakes during detailed design and functional simulation, detecting subtle functional errors and performance problems.

A prototype implementation also allows performance evaluation using realistic workloads. Although usually limited to a single system configuration, the results are more convincing than analytic models and trace-driven simulations because the measurements come from a real system.

In this chapter we discuss the implementation of in-cache translation in the SPUR prototype. In the next section we describe the SPUR memory system and the constraints that shaped its design. We provide enough implementation details in this section to understand the performance analysis in Section 5.2 and the analyses of Chapter 7 and Chapter 8. In this analysis we break down a cache miss, and examine how SPUR uses each cycle. Then

we propose design optimizations that could improve performance by eliminating cycles in a more aggressive implementation. In Section 5.3 we examine alternative approaches to hardware performance monitoring, and describe the event counters imbedded in the SPUR cache controller chip. We will use these counters in the measurement evaluation of Chapter 6. Finally, we conclude with the implementation status and important lessons.

5.1 Implementation Overview

In this section, we describe the implementation of the SPUR memory system, providing deeper insight into in-cache translation and background for the performance evaluation of Section 5.2. We start by discussing implementation philosophy, and how this affected our design strategy. In Section 5.1.2 we summarize the design of the processor board, explaining the major components and their interactions. In Section 5.1.3 we describe the organization of the cache controller, the custom VLSI chip that implements the in-cache translation mechanism.

5.1.1 Implementation Philosophy

Building a multiprocessor computer requires enormous effort. Several similar projects, conducted at industrial research laboratories and begun about the same time as SPUR, have either been canceled before completion[57], or drastically scaled back[34]. Building a multiprocessor in a university research program is even more difficult, because most universities lack the extensive resources and experienced personnel found in industrial settings.

To prevent SPUR from succumbing to this unfortunate fate, we continually reminded ourselves of *Ousterhout's Observation:*

> The largest speedup a system ever achieves occurs when it goes from not working to working. In this case, the speedup is infinite.

Rather than build the most aggressive, high-performance system possible, we occasionally sacrificed some potential performance in return for reduced design complexity. This improved our chances of developing a working prototype.

In SPUR, we focused our efforts on the novel aspects of the design, choosing to use straight-forward solutions for the standard problems. These expedient solutions often increased the implementation cost, or reduced performance; however, these drawbacks were justified. By simplifying the overall task, we reduced the design time and risk of failure. A commercial implementation, using the greater resources available to industry, could easily correct these shortcomings.

This philosophy affected many design decisions in the SPUR memory system. For example, high-performance systems frequently optimize cache misses by transmitting the requested word as the first word in the block. This allows the processor to begin execution

immediately, rather than waiting for the entire block. However, while this optimization significantly improves performance, it also increases the complexity of the cache and memory controllers. Therefore, in keeping with our philosophy on prototypes, we chose not to implement this optimization. Another important example is the trade-off between number of cycles and cycle time. In several cases, rather than risk increasing the cycle time by combining two infrequent operations into a single cycle, we split the two into separate cycles. This design strategy is conservative when a more aggressive implementation, using trickier circuits or more tightly optimized control, could combine both operations into a single cycle without degrading the cycle time.

5.1.2 Board Design

Each SPUR processor consists of a single 14.5" by 16" processor board containing 3 custom VLSI chips: a central processor unit (CPU), a floating point coprocessor (FPU), and a cache controller (CC). Figure 5.1 shows the block diagram of the SPUR processor board, and Figure 5.2 is a photograph of the actual printed-circuit board. The cache contains 128 kilobytes of data and instructions, organized as direct-mapped with 32-byte blocks.

The cache requires three sets of address tags: processor tags, snoop tags, and physical tags. Both the processor and snoop tags contain virtual address tags, implementing a logically dual-ported memory using standard single-ported static RAMs. The CPU references check the processor tags to detect cache hits. The snoop tags are referenced on each bus transaction, to check whether intervention is necessary to maintain cache coherency. The physical address tags eliminate translation on writebacks, as described in Chapter 3. Both the processor and snoop tags have a two-bit state field for the coherency protocol. The processor tags also have a two-bit protection field, for access validation, and a block dirty bit, to minimize cache writebacks. Finally, the processor tags have a page dirty bit, as explained in Chapter 8.

5.1.3 Cache Controller Design

The cache controller chip provides an illusion of a single shared virtual memory to the CPU and FPU chips. It does this by implementing the in-cache translation mechanism and Berkeley Ownership coherency protocol. The cache controller also supports a complete set of system and diagnostic functions: memory-mapped interrupts, interval timers, EPROM, serial ports, etc. In addition, to allow evaluation of the new memory system algorithms, the chip provides a set of performance counters, described in Section 5.3. These counters enable the performance measurements described in Chapter 6.

The cache controller chip has been fabricated by Hewlett-Packard in a $1.6\mu m$ N-well double-metal CMOS process, through the MOSIS implementation service. The large die, 11.5mm by 11.5mm, contains 68,395 transistors; the 208 pads lining the periphery dictate the size of the die, rather than the transistor count. An additional 20 probe pads provide diagnostic support during chip testing. Figure 5.3 shows a chip microphotograph and

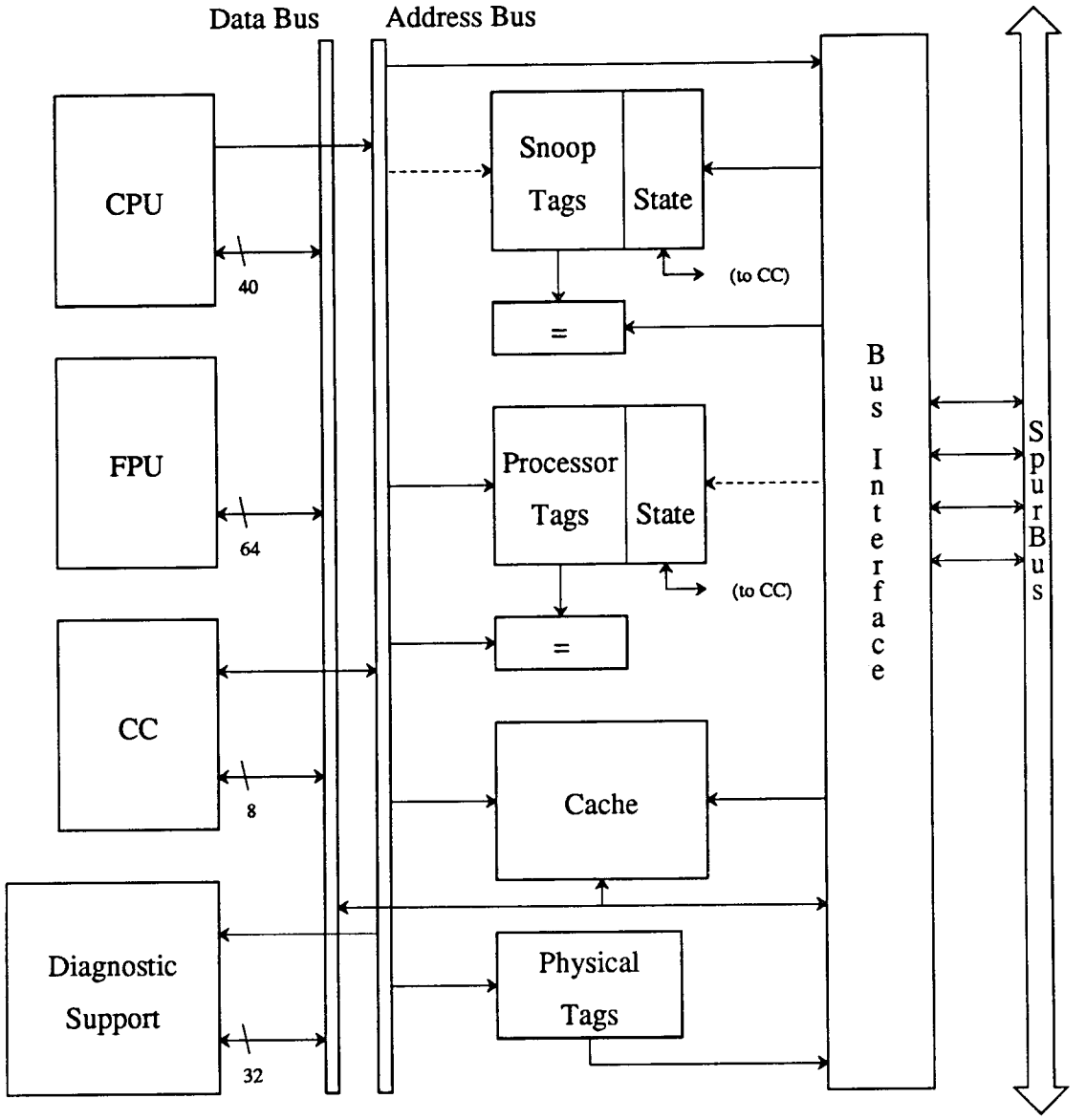
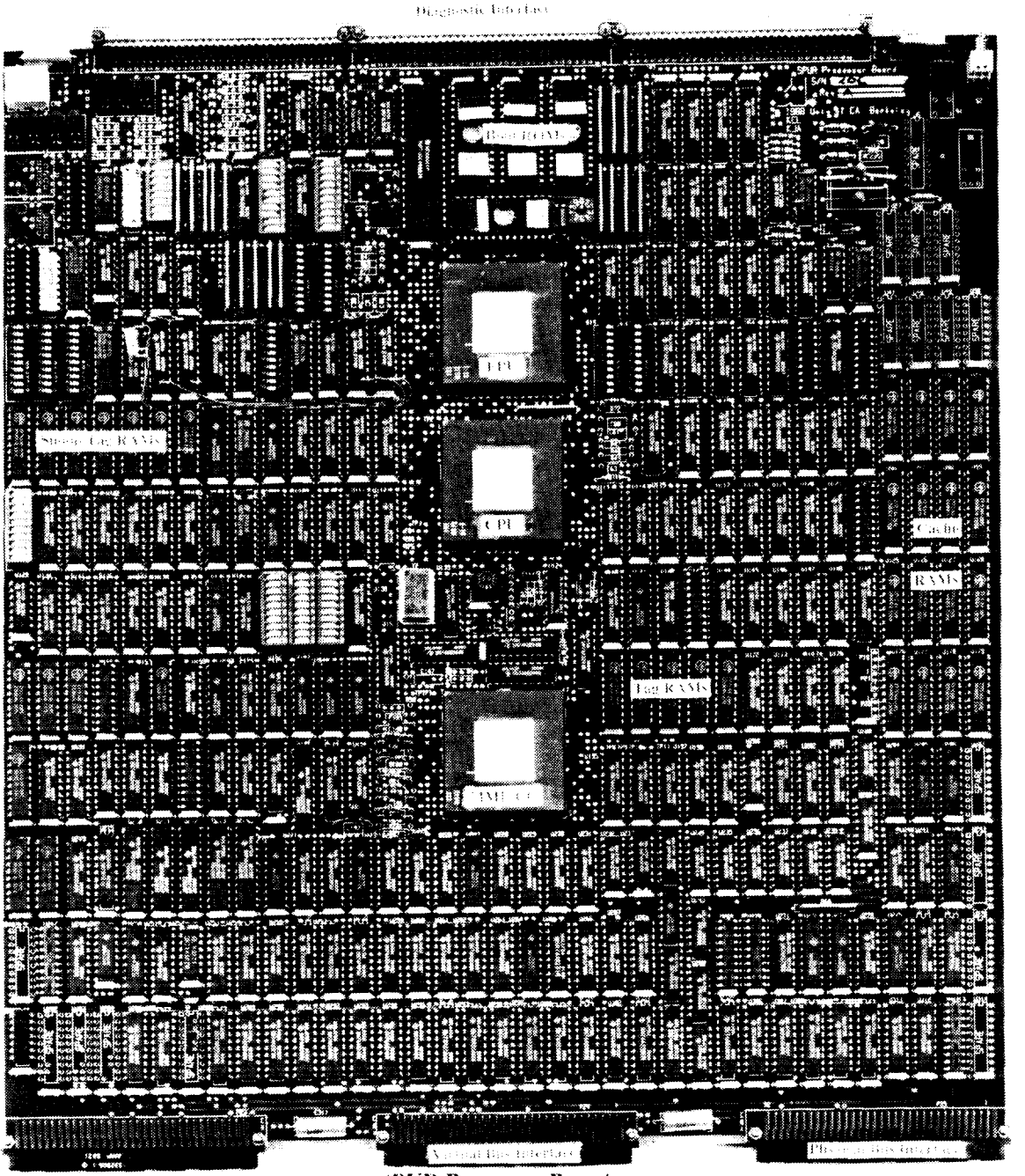


Figure 5.1: Block Diagram of SPUR Processor Board

The SPUR processor board contains three custom chips, the central processing unit (CPU), floating point unit (FPU), and cache controller (CC). These chips communicate with the virtual address cache via separate address and data busses. The bus interface logic connects the processor board to the SpurBus, allowing communication with main memory and other processors.



SPUR Processor Board

Figure 5.2: Photograph of SPUR Processor Board

Number of transistors	68,395
Number of PLAs	19
Total number of PLA product terms	707
Die Size	11.5mm x 11.5mm
Package	208 pin PGA
Power Dissipation	0.7W @ 5V, 10Mhz
Process	Double metal 1.6 μm N-well CMOS

Table 5.1: Cache Controller Chip Statistics

Table 5.1 summarizes the chip implementation statistics.

The block diagram in Figure 5.4 illustrates the separation of the cache controller into two semi-autonomous sections. The *processor cache controller (PCC)* manages the cache on behalf of the CPU, performing address translation and system support functions. The *snooping bus controller (SBC)* communicates with other processors, main memory, and peripherals via the SpurBus[35], an extended version of the NuBus[44]. The SBC monitors the bus and checks the snoop tags to detect transactions that require coherency operations. The two controllers interact whenever a reference misses in the cache or the Berkeley Ownership protocol requires intervention.

The bus and processors are independently synchronous subsystems, running off of separate clocks. The NuBus specification sets the frequency of the bus clock at 10 MHz; there is little flexibility in this parameter because of the electrical and timing properties of the backplane and existing memory boards. Having separate clocks allows the frequency of the processor clock to be determined by the processor implementation, and not be constrained by the bus. However, having separate clocks makes the PCC and SBC asynchronous with respect to each other, requiring a sophisticated synchronization scheme to prevent metastable state problems[45].

The SBC itself consists of several finite-state machines. The *Master* controller handles memory requests on behalf of the PCC. The *Slave* controller monitors the bus, checking each request to see if intervention is necessary to maintain cache consistency. Two additional state machines handle the actual data transfers initiated by Master and Slave, a third monitors the NuBus subset, and a fourth detects bus and system resets. This separation of function simplifies specification and debugging of the controller: each state machine is small and performs only a limited function. On the other hand, since a state machine requires a full cycle to send a signal to a second state machine, this partitioning increases the latency of some operations. However, trading performance for design simplicity is in keeping with our philosophy on prototypes.

The PCC is also split into two controllers. However this partitioning improves performance. The *Hitmiss* controller only detects cache hits and misses, telling the CPU whether

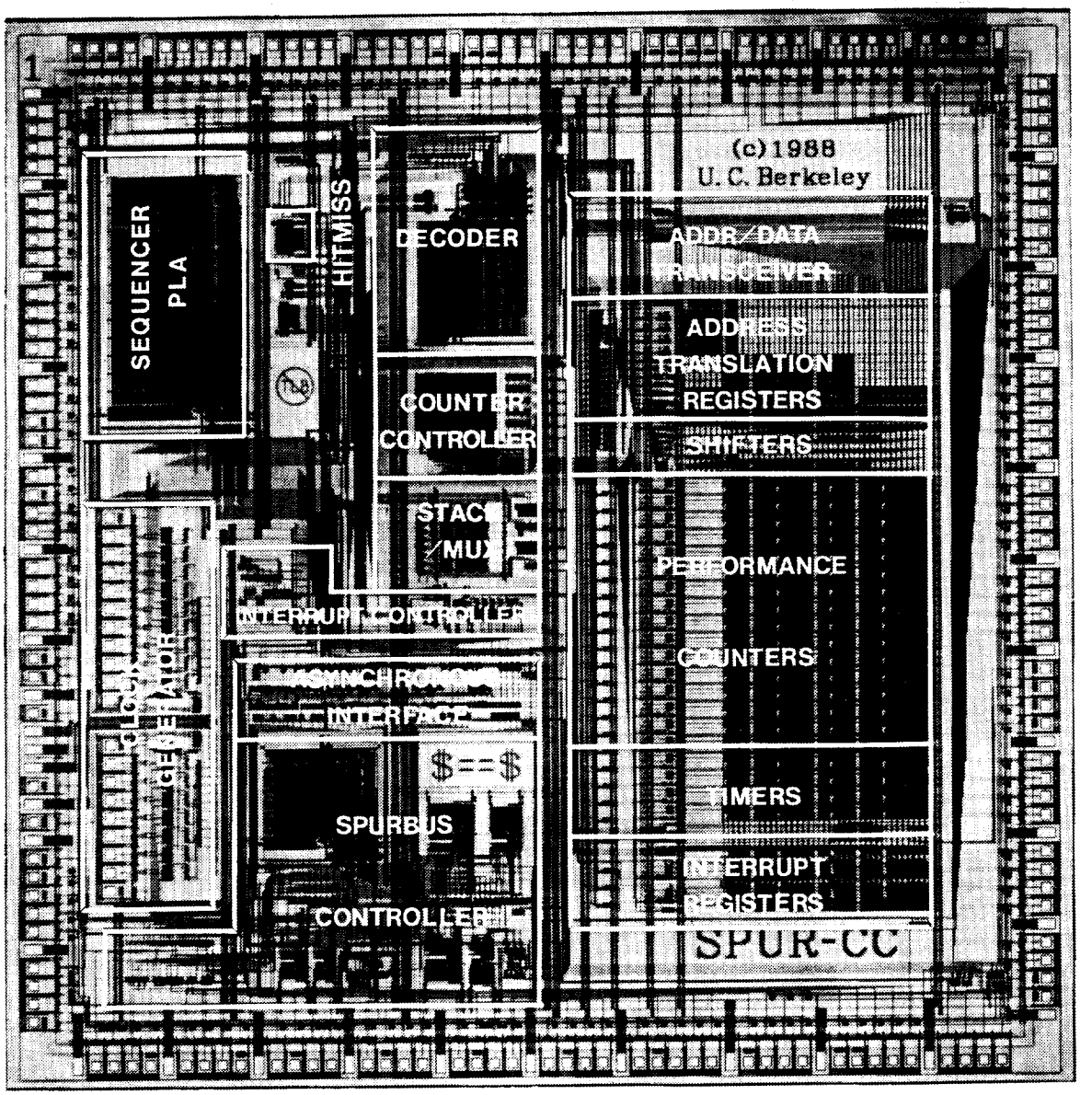


Figure 5.3: Cache Controller Microphotograph

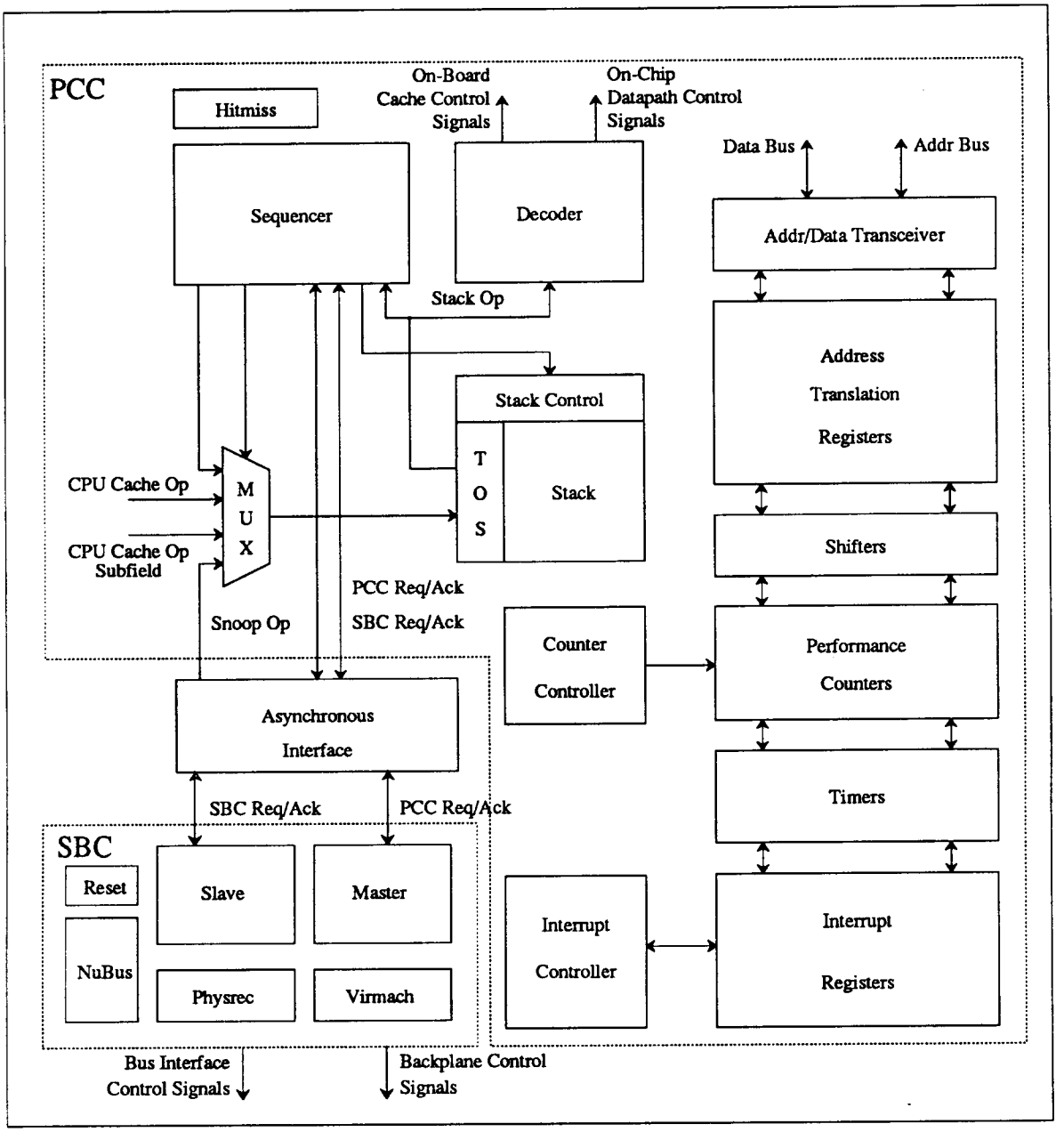


Figure 5.4: Cache Controller Block Diagram

This diagram illustrates the functional blocks of the SPUR cache controller chip. The processor cache controller (PCC) manages the cache on behalf of the CPU, performing in-cache translation on cache misses. The snooping bus controller constantly monitors the SpurBus and notifies the PCC when the Berkeley Ownership protocol requires intervention.

it can continue at full speed, or must wait while the main controller handles a miss. This small controller is implemented as as a simple high-speed finite-state machine. The main controller, called the *Sequencer*, handles all other PCC functions. This controller is the most complex part of the entire system, and is therefore much slower than the Hitmiss controller. By separating the most common cases, cache hits, into a separate fast controller, we reduce the cache access critical path and improve overall performance.

Because of the recursion of in-cache translation, coupled with the asynchronous behavior of the cache coherency protocol, we found it difficult to express the main cache controller functions as a finite-state machine. Instead, inspired by the theoretical pushdown automata, we implemented the Sequencer using a pushdown stack. On each cycle, the controller can push, pop, or replace the top-of-stack entry. Because the stack provides a subroutine capability, it simplifies the expression of the in-cache translation algorithm.

The cache controller datapath runs under PCC control, and consists of three major sections: in-cache address translation support, performance counters, and system support. The privileged CPU instructions *ld_external* and *st_external* read and write the datapath registers. Because of pin limitations on the cache controller, these instructions only operate on one byte at a time; the datapath provides a byte inserter/extractor to access the correct bytes.

The address translation section implements the design described in Chapter 3, with one minor exception. To simplify the datapath control and implementation, two registers point to the pagetable in virtual space: one contains the base address of the pagetable, and another points directly to the root pagetable. Since the operating system initializes the pagetable address at system bootstrap time and never changes it, the two copies do not pose a significant consistency problem.

The 16 performance counters provide a means to evaluate the memory system performance. A counter controller, implemented as a PLA, detects events and increments the appropriate counters. Section 5.3 describes the operation and application of these counters in depth.

The system support functions consist of timers, interrupts, and exceptions. Three timers, controlled by separate bits in a mode register, are incremented each cycle they are enabled. The 64-bit timer provides a fine-grain timestamp mechanism. The two 32-bit interval timers generate interrupts when they expire, providing a periodic interrupt capability for time-slicing and profiling. In addition to local interrupts, such as the interval timers, the system provides 16 memory-mapped interrupts. Writes to *slot space*, a region of the physical address space, generate interprocessor interrupts, which can be individually masked out. Exceptions, such as page faults and parity errors, are reported in a 32-bit status register.

5.2 Prototype Performance

Our conservative design strategy helped us develop a working prototype, but it also led to a lower performance implementation. While the most common case – cache hits – executes in a single cycle, the less frequent cases, like cache misses and address translation, require more cycles than absolutely necessary. For example, in Chapter 4 we assumed a 12 cycle cache miss penalty, with one additional cycle for in-cache address translation. In the SPUR implementation, cache misses actually require 25 cycles[1] including translation, as described in this section. We show in this section that most of the difference is independent of in-cache translation; the one cycle that would not be needed in a system with a translation lookaside buffer could be eliminated with a design optimization.

In the first section, we breakdown a cache miss and attribute the cycles to the various functions and controllers. In the second section, we analyze the cycles and suggest changes to the cache controller design that could improve performance, generally at the cost of increased complexity.

5.2.1 Cache Miss Breakdown

We examine the simplest case of a cache miss: where a valid pagetable entry resides in the cache, and the block selected for replacement does not require a writeback. Table 5.2 breaks the cache miss into its component cycles, explaining the operations taking place in each.

The CPU reference occurs in cycle 0; the Hitmiss controller detects the miss and stalls the CPU. According to our definition of effective access time (see Equation 4.15), this cycle is attributed to the reference, not the cache miss. The Sequencer samples the control signals at the end of the this cycle, and consumes the entire next cycle generating control signals for the pagetable entry access in cycle 2. Because of the partitioning of the CPU and the cache controller, this cycle would be necessary even if the Sequencer did not require it. When the cache controller detects a miss, it must turn off the CPU's address drivers and turn on its own, driving the pagetable entry virtual address to the cache. Because of the capacitive loading of this control signal, and the slow enable/disable speed of the external TTL buffers, changing the address source and reading the cache in a single cycle would increase the cycle time by nearly 10%. Since cache misses only occur about 2% of the time, adding the extra cycle affects performance less than increasing the cycle time.

The Sequencer samples its input signals at the end of cycle 2 and spends the third cycle checking that the pagetable entry is valid and that the CPU's request should not generate a fault. The Sequencer then requests that the SBC begin a bus transaction. After a synchronizer delay incurred while passing across the asynchronous interface, the SBC Master finite-state machine initiates the read request. Arbitration takes two cycles if the processor is not *parked*: i.e., if no other board has become bus master since this processor's

[1] Assuming that the bus and processor clocks are equal.

Cycle	Description
Cycle 0	CPU Reads Cache (Assume Miss)
Cycle 1	Sequencer Control Delay: Detect Miss and Generate Control Signals;
	Change Source of Address Bus
Cycle 2	Read Pagetable Entry from Cache (Assume Hit)
Cycle 3	Sequencer Control Delay: Check Pagetable Entry
Cycle 4	Synchronizer Delay
Cycle 5	Handshake Delay: SBC Master Initiates Operation
Cycle 6–7	Arbitration (Assume not Parked)
Cycle 8	Send Address to Main Memory
Cycle 9–11	Memory Latency
Cycle 12-19	Data Transfer, ACK on Last Cycle
Cycle 20	Handshake Delay: SBC Master Relays ACK to PCC
Cycle 21	Synchronizer Delay
Cycle 22	Sequencer Control Delay: Generate Update Signals
Cycle 23	Update Processor State and Tags
Cycle 24	Change Source of Address Bus
Cycle 25	CPU Rereads Cache (Hit)

Table 5.2: Summary of Miss Penalty Cycles

This table itemizes the cycles incurred during a cache miss when the first-level pagetable entry resides in the cache. With our definition of effective access time (see Equation 4.15), cycle 0 is not considered part of the miss penalty, although we include it here for completeness.

last request. The bus transaction itself requires one cycle to send the address, three cycles of memory latency, and 8 cycles of data transfer (one word per cycle).

The SBC consumes one cycle to relay the acknowledgement to the PCC, the asynchronous interface consumes a second cycle, and the Sequencer requires a third cycle to generate control signals in response to the acknowledgement. The Sequencer updates the cache state in cycle 23. Cycle 24 is the reverse of cycle 1: the address bus is returned to the CPU. Note that this cycle would be required even if we used a translation lookaside buffer rather than in-cache translation, since the cache controller must drive the low order address lines (the word index) while the data is transferred into the cache. Finally, the CPU retries its reference, which is guaranteed to hit[2].

5.2.2 Performance Optimizations

While performance was not our first concern in the prototype, we are still interested in determining how we might improve the design for a commercial system. Table 5.3 analyzes the miss penalty in more detail, attributing the different cycles to their respective sources of overhead. The table's last column suggests techniques to reduce or eliminate the cycles. The first cycle is the reference that misses, and is not considered part of the miss penalty. The next 12 cycles correspond to the miss penalty ($t_{readblock}$) assumed in Chapter 4. Except for arbitration, the remaining cycles are overhead incurred by the cache controller. A more aggressive cache controller design could eliminate many of these extrinsic cycles, although doing this may increase the cycle time. We discuss each of the optimizations in turn.

State Update

The SPUR cache controller does not implement a common cache miss optimization, which improves performance by overlapping necessary operations. SPUR defers updating the cache state until the entire block is written into the cache. We could overlap this state update with the block transfer or memory latency, saving a cycle at the end of the cache miss. However, block transfers from memory can be interrupted by parity errors and other recoverable exceptions. If we employ this optimization, recovering from errors becomes more difficult, requiring additional recovery logic. While commercial systems can justify this extra complexity, it is not needed in a prototype system[3].

[2]To prevent a potential *live-lock* condition, the cache controller must guarantee that the CPU's read completes without an intervening coherency operation. Otherwise, two processors could "ping-pong" the block back and forth, never making forward progress.

[3]An alternative position for a prototype is to not recover from exceptions such as parity errors, since they occur infrequently.

Number of Cycles	Cycles Affected	Source of Overhead	Reduction Techniques
1	0	CPU Reads Cache (Miss)	None.
1	8	Send Address to Memory	None.
3	9-11	Memory Latency	Use faster memory chips.
8	12-19	Data Transfer	• Use wider bus. • Transfer data to CPU "on the fly".
1	2	Pagetable Entry Lookup	None.
1	23	State Update	Overlap state update with data transfer or memory latency. Makes error recovery more difficult.
1	25	CPU Rereads Cache	Transfer data to CPU "on the fly", rather than waiting until transfer completes. Makes error recovery more difficult.
2	6-7	Arbitration	• Use faster arbitration scheme (requires a different bus). • Overlap arbitration with pagetable entry lookup.
2	4,21	Synchronizer Delays	• Make system fully synchronous. • Repartition control to overlap synchronizer delays with address source delays.
2	5,20	Handshake Delays	Integrate SBC Master and PCC Sequencer to eliminate handshake.
2 (3)	(1),4,22	Control Delays	• Fit Sequencer into cache access time. Increases the cycle time. • Use local control logic to optimize common cases.
1 (2)	(1),24	Change Address Source	• Integrate CPU and CC onto a single chip. • Overlap with synchronization delay by repartitioning control.

Table 5.3: Breakdown of Miss Penalty Cycles

This table categorizes the cycles during the miss penalty by type. By our definition of effective access time, the cycle required to detect the miss is not considered part of the miss penalty. The next 12 cycles correspond to the ideal miss penalty and the remaining cycles are cache controller overhead.

CPU Rereads Data

A similar optimization eliminates the final cycle of a cache miss. When SPUR transfers a block of data from memory, it stores the entire block into the cache then requires an extra cycle for the CPU to reread the requested word. The optimization bypasses the cache for the requested word: in addition to writing the desired word into the cache, it also sends it directly to the CPU. This change requires an asynchronous *load* signal on the CPU's memory data latch, since the block transfer occurs synchronous to the bus clock, rather than the processor clock.

Data Transfer

Transferring the data to the CPU "on the fly" allows a further optimization. Once the CPU has the requested word it can resume execution in parallel with the remainder of the data transfer. Some systems re-order the cache block, transferring the requested word from memory first. These optimizations significantly reduce the average miss penalty, but increase the complexity of the cache controller. Since the processor state may change before the transfer completes, error recovery becomes more difficult.

Arbitration

Our analysis of cache misses assumes that arbitration consumes 2 cycles. To initiate a bus transaction, a processor must first negotiate with other processors to become *bus master*. Once elected master, it remains *parked* until another board requests the bus. If the processor is still parked when initiating a request, it bypasses arbitration and begins the transaction immediately. The probability of finding the bus parked depends upon how heavily the bus is loaded; in a large multiprocessor this probability tends to be very low.

In the SPUR implementation, the PCC performs in-cache translation before signaling the SBC to begin the bus operation. An aggressive implementation could initiate arbitration as soon as it detects a cache miss, rather than waiting for address translation. This allows the pagetable entry lookup to execute in parallel with arbitration, reducing the total latency.

Since the bus ultimately limits the performance of a multiprocessor, we don't want to waste bus bandwidth. This proposed optimization clearly wastes cycles in some situations. First, when the processor is parked, it wastes cycles until in-cache translation completes. Second, if the cache does not contain the needed pagetable entry, we must make additional cache accesses, holding the bus idle until we are ready to fetch a block. Finally, in the case that the reference generates a page fault, we have unnecessarily obtained the bus, which must then be released with a null bus transaction.

However, none of these cases result in a significant performance loss. First, to be parked, the bus must have been idle since the last request by this processor; wasting two cycles from an idle bus is of no consequence. Both the second and third cases occur infrequently: less than one reference in a thousand requires a reference to the root pagetable, and page faults

occur even less often. In addition, bus throughput becomes most important for heavily loaded systems. But since average waiting time increases as bus utilization increases, the average number of cycles lost to these cases declines as the system load increases.

This optimization is particularly important because the extra cycle to reference the pagetable entry accounts for a significant fraction of the in-cache translation overhead, as shown in Chapter 4. By overlapping the pagetable entry lookup with arbitration, which is necessary in any bus-based multiprocessor, this optimization makes the miss penalty the same for both in-cache translation and a translation lookaside buffer design.

Synchronizer Delays

The cache controller implementation suffers from three types of controller delays: synchronization delays across the asynchronous interface, handshake delays between communicating controllers, and control delays because Sequencer execution requires a full cycle. We can eliminate the extra cycles resulting from these control delays with a more aggressive cache controller design. However, eliminating these cycles may require an increase in the cycle time.

We can only eliminate the synchronization delays by implementing a fully synchronous system, with the processors and bus sharing a single clock. While this approach recovers the lost cycles, it makes it difficult to take advantage of faster processor implementations. The speed of the system is set by the bus clock, which does not scale easily with technology improvements. Thus eliminating synchronization delays may ultimately degrade performance.

Handshake Delays

The main source of handshake delays is the separation of the SBC's Master finite-state machine from the PCC's Sequencer. On every cache miss, the Sequencer requests that Master initiate a bus transaction; when it completes, the Master relays the acknowledgement back to Sequencer. Each handshake consumes a single cycle. Eliminating these handshake cycles requires merging the two controllers; however, the asynchronous interface forces this functional partitioning. To eliminate this partitioning requires that we move the asynchronous interface.

One possibility is to use an asynchronous bus, such as the proposed IEEE P896 FutureBus[13]. Since the bus has no common clock, we can combine Sequencer and Master into a single controller that generates bus signals directly. The main drawback of this approach is that the Sequencer is already large and slow: integrating the functions of Master will probably not improve its performance. In some technologies, the cycle time differences between a large combined controller and a smaller Master controller may justify the extra handshake cycles.

Alternatively, we can move the asynchronous interface between the Hitmiss controller

and the Sequencer. Since the Sequencer only executes on cache misses, it does not need to be synchronous with the processor, and can be driven off the bus clock. On cache misses, Hitmiss would signal Sequencer to begin miss handling. This alternative has the same trade-offs as the asynchronous bus, except that since the SpurBus is synchronous the combined controller must achieve a predefined cycle time. The other advantage of this scheme is that synchronizer delays overlap with the address source delays discussed below.

Sequencer Control Delays

Sequencer control delays arise because we allow a full cycle for Sequencer evaluation. Three cycles, cycles 1, 3, and 22, are spent *thinking* (although cycle 1 is also required by an address source delay). The simplest way to eliminate these cycles is to lengthen the cycle time so that a cache reference and the Sequencer can both execute in the same cycle. But, as we argued earlier, this is poor trade-off because these additional cycles are relatively infrequent, and increasing the cycle time (by at least 30%) affects all cycles.

There are alternative ways to eliminate some of the additional cycles. Sequencer is a slow global controller; a standard optimization technique is to use fast local controllers to accelerate certain cases. For example, while Sequencer requires a full cycle to detect a cache miss and generate the appropriate control cycles, Hitmiss detects the miss at the end of the cache reference. By slightly extending Hitmiss, this fast controller could generate the control signals needed to read the pagetable entry in cycle 1, rather than waiting until cycle 2 (assuming the address source delay is also eliminated).

Similarly, the control delay in cycle 22 occurs when Sequencer sees the acknowledgment from Master, and generates control signals to update the cache state. If we overlap the cache state update, as discussed above, then local control logic could forward the acknowledgement directly to the CPU. While Sequencer would still think for a full cycle, the CPU would not wait for it to complete.

Address Source

During cycles 1 and 24, Sequencer changes the source of the address bus: in cycle 1, it disables the CPU and enables the cache controller, and in cycle 24, it switches back. Because of our conservative design rules, each custom chip's address bus is buffered by an AS TTL driver, with a worst-case turnaround time of 9 nanoseconds. We allocate an extra cycle to change the address source because otherwise the off-chip drivers would increase the cycle time by nearly 10%. From Equation 4.25 we know that adding an extra two cycles to the miss penalty only decreases performance by 2%. *Therefore, increasing the cycle count is better than increasing the cycle time.* We could eliminate these extra cycles by integrating the address translation portion of the cache controller datapath onto the CPU chip: the extra cycles become unnecessary because the address would always be driven from the same source. Even with the current sparse layout, this portion of the datapath only measures 2mm by 4mm, or 6% of the total chip area. The CPU chip could accommodate

this additional logic without much effort.

5.3 Performance Monitoring Support

One of the reasons to build prototype systems is so we can evaluate their designs under realistic workload conditions. By measuring real programs running on real hardware, we obtain more convincing results. However, to evaluate the effectiveness of specific features requires additional monitoring hardware.

There are several approaches to hardware performance monitoring, each with its advantages and disadvantages. One approach is to trace the machine's execution, using a technique like ATUM, and then analyze the trace[4]. However, as we observed in Chapter 4 and Appendix A, resource limitations often restrict the length of the traces, leading to distortions in the subsequent analysis.

Another approach, popularized by Clark, et al., in their studies of the VAX 11/780 and VAX 8800 machines[20, 19], uses a microcode histogram mechanism. This device counts the number of times each micro-instruction executes successfully. A second histogram device counts the number of cycles spent *stalled* in each micro-instruction. A third device counts the cycles and events on the bus.

The histogram approach is very powerful because it captures the frequency of most operations in the system. However, the apparatus is very expensive. For example, the VAX 8800 requires 16384 buckets for each of the two main histogram devices. There are more bits of high-speed RAM in this performance monitor than in the VAX 8800 cache. Because the monitor is so expensive, only a few systems will be equipped with them.

Histogram mechanisms require access to control state information. Unfortunately, in a highly integrated design like SPUR, most of the state is concealed within the custom chips. Since package pins are a critical resource, it is hard to justify dedicating a substantial number merely to permit access for an external histogram apparatus that will be attached to only a few machines.

An attractive alternative is to imbed some performance monitoring hardware directly on the chip. This gives us the necessary access to control signals without dedicated pins. Of course, since chip area is also a critical resource we cannot afford to build a general-purpose histogram device. Instead, we can only count a limited set of important events. However, if we select the events carefully, then we make the most important measurements available on all machines without requiring expensive add-on hardware.

We took this approach in the SPUR cache controller, including a set of counters to measure the memory system performance. Area and timing considerations limited the chip to only 16 counters, less than the number of interesting events. To extend the number of countable events, the counter control supports 5 different measurement modes: *Off*, *Snoop*, *User*, *Kernel*, and *Combined*. The first mode, Off, disables the counters, allowing the operating system to read a consistent set of counts. In Snoop mode, the counters measure

the performance of the cache consistency protocol. In the last three modes, referred to collectively as Cache mode, the counters measure basic cache performance and in-cache translation performance. The three different counter modes permit differentiation between user and kernel mode references.

Table 5.4 briefly describes each of the events, Table 5.5 summarizes which events are counted in each of the different modes. Each counter records either the number of times a specific event occurs or the number of cycles consumed by an event.

In many cases, the counters measure implementation-specific events, rather than architectural events. For example, one counter measures the total number of instruction fetches; another counter measures the total number of instruction fetches *plus* the number of instruction misses. Simple arithmetic suffices to compute the desired metrics: subtracting the two counters yields the number of instruction misses. Table 5.6 summarizes the equations needed to compute the most interesting metrics.

The counters measure the number of hits and misses of each type of reference: instruction prefetches[4], instruction fetches, data reads, data writes, and first-level and second-level pagetable references. These measurements allow us to compute miss ratios for each type of reference. We also count the number of blocks written back to memory and the number of cycles spent waiting for the bus. Additional information on the counters is published in a technical report [92].

Because the counter control is hardwired (modulo the different modes), we must determine which events to count when we design the chip. This requires careful planning to insure that we do not inadvertently exclude an important event. Since even the best laid plans often go astray, we dedicated two counters and two cache controller pins to count external events. These two external event counters provide a limited patch capability, allowing us to count any events detectable using only external state. These counters also provide a way to measure events that don't directly relate to the cache controller, such as the number of floating point instructions executed and the instruction buffer miss ratio.

The counters are each 32 bits long. If we had implemented them with a simple ripple-carry, then the carry-propagation delay could have become the chip's critical path. Carry-lookahead techniques, while appropriate for arithmetic units, require too much area for a bank of counters. The VAX 8800 histogram mechanism solved this problem by implementing their incrementer with a linear feedback shift register[19]. While this solution is both simple and elegant, it has the major drawback that the conversion from these "pseudo-counts" to 2's complement integers (or floating point) is difficult and slow. We would like to make reading the counters as fast as possible.

In SPUR we chose to implement the counters in an 8-bit pipelined fashion. On each increment, the carry ripples through 8 bits and the carry-out is buffered in a dynamic latch. Since a single event takes several cycles to propagate through the counter, they must be read carefully. The operating system turns off the counters while reading them to prevent

[4]Since the instruction buffer was not operational in this study, this metric is always 0.

Event Name	Description
PreFetches	Number of instruction prefetches.
PreFetchHits	Number of instruction prefetch hits.
Fetches	Number of instruction fetches.
FetchesSuperset	Number of instruction fetches plus instruction fetch misses.
Reads	Number of data reads.
ReadsSuperset	Number of data reads plus data read misses.
Writes	Number of data writes.
WritesSuperset	Number of data writes plus data write misses plus DirtyMisses plus MasterWFIs.
Writebacks	Number of blocks written back to memory (includes explicit flushes and copybacks that are canceled by an external invalidation).
UPTEMisses	Number of misses to first-level pagetable entries.
RPTEsSuperset	Number of references plus misses to second-level pagetable entries.
DirtyMisses	Number of extra misses to check the pagetable entry dirty bit on a write.
External1	Number of events on external event pin 0.
External2	Number of events on external event pin 1.
MasterWFI	Number of WriteInv bus operations generated, including those changed to Reads by an external invalidation.
BusWait	Number of cycles the PCC waited for a bus transfer to complete, including arbitration.
MasterRFO	Number of ReadOwn bus operations generated.
MasterRS	Number of Read bus operations generated.
OwnRS	Number of times the cache owned a block on an external Read.
OwnRFO	Number of times the cache owned a block on an external ReadOwn.
OwnPrivateRS	Number of times the cache held an OwnPrivate block on an external Read.
OwnDirty	Number of times the cache transferred an owned dirty block.
PCCBusyCache	Number of cycles the PCC stalled a snoop request for the cache.
PCCBusyState	Number of cycles the PCC stalled a snoop request to update its state.
RFOInv	Number of invalidations due to an external ReadOwn.
WFIInv	Number of invalidations due to an external WriteInv.
WaitWFI	Number of cycles the PCC waited for WriteInv to complete (including arbitration).
WaitUpdate	Number of cycles the PCC was idle while the snoop updated its state on a cache flush (of a clean block only).
WBInterference	Number of Writebacks canceled by an external ReadOwn or WriteInv.
WFIInterference	Number of WriteInvs canceled (and changed to ReadOwn) by an external ReadOwn.

Table 5.4: Events Counted by the SPUR Cache Controller

Counter Number	User Kernel Combined	Snoop
C0	Prefetches	BusWait
C1	FetchesSuperset	MasterWFI
C2	ReadsSuperset	MasterRS
C3	WritesSuperset	MasterRFO
C4	PrefetchHits	WaitWFI
C5	Fetches	WaitUpdate
C6	Reads	WFIInv
C7	Writes	RFOInv
C8	RPTEsSuperset	OwnRS
C9	UPTEMisses	OwnDirty
C10	Writebacks	OwnPrivateRS
C11	DirtyMisses	OwnRFO
C12	BusWait	PCCBusyCache
C13	MasterWFI	PCCBusyState
C14	External1	WFIInterference
C15	External2	WBInterference

Table 5.5: SPUR Event Counters

This table lists the specific events measured by each counter in snoop and cache modes.

Metric	Formula	Description
FetchMisses	C1 - C5	Number of instruction fetch misses.
ReadMisses	C2 - C6	Number of data read misses.
WriteMisses	C3 - C7 - C11 - C13	Number of data write misses.
UPTERefs	FetchMisses + ReadMisses + WriteMisses	Number of first-level pagetable references.
RPTERefs	C9	Number of second-level pagetable references (same as number of first-level misses).
RPTEMisses	C8 - C9	Number of second-level pagetable misses.

Table 5.6: Equations for Typical Cache Metrics

This table lists equations needed to compute architectural events from the implementation events measured by the counters.

seeing spurious values (disabling interrupts to minimize the measurement distortion).

Since the counters are only 32 bits, and the SPUR cycle time is nominally 100ns, the minimum overflow time is approximately 7 minutes. To permit measuring longer programs, the Sprite operating system maintains a set of 64-bit "software" counters. Once every 5 minutes, the kernel reads the hardware counters, and after checking for overflow updates the 64-bit software versions. Since this routine is quite efficient and only executes once every 5 minutes, the measurement distortion is imperceptible.

We read and write the counters using the standard *ld_external* and *st_external* instructions, the same way we access the address translation registers. Because these operations are privileged, only the operating system can execute them. User programs access the counters through a special software device named "/dev/pcc", using the standard *read()* system call interface. Reading this device interrupts all processors, causing them to update their software counters. The mode register is set using an *ioctl()* system call. Several programs make use of the counters, the most important being the Sprite benchmark driver. This program gathers cache and operating system statistics for the specified program or script. Thus we can measure the cache behavior of any program simply by invoking it under this benchmark driver.

Traditionally, measuring cache performance has required special add-on hardware, making it impractical for everyday use. But because *every* SPUR processor includes a set of performance counters, programmers can easily measure the cache performance of their applications. Thus, when a program runs unexpectedly slow, the programmer can determine whether it is due to poor cache performance or some other problem.

5.4 Implementation Status

The initial discussions of the SPUR project began in the academic year 1983-1984. The goals of the project solidified over the next several years, and detailed design began during the second half of 1985. The design was completed by September of 1986, when we began functional simulation using a random test case generator[93]. We submitted the cache controller for fabrication in November 1987, and received silicon in February 1988. We completed unit testing in April 1988, using test vectors extracted from the functional simulation. In May 1988, we completed uniprocessor hardware testing, with the CPU and cache controller running diagnostics together in a processor board. Multiprocessor testing, delayed by competing demands for resources and personnel, was completed in December 1988 after less than one month of effort. The operating system group began to port Sprite to SPUR hardware in July 1988, and had a reliable uniprocessor system by September. As of November 1989, a five processor SPUR system is operational.

5.5 Summary

In this chapter we described the implementation of in-cache translation in the SPUR prototype. We described the organization of the cache controller, the custom VLSI chip that manages the cache and interfaces to main memory. This chip implements the in-cache translation algorithm, as well as the Berkeley Ownership coherency protocol.

We examined the performance of the cache controller, breaking down a cache miss into its component cycles. We showed that cache misses in SPUR require 25 cycles for an eight-word block, rather than the 12 assumed in the idealized analysis. We presented a list of performance optimizations that could be implemented in a more aggressive cache controller implementation. With these optimizations, the miss penalty could be as low as 6 cycles. More important, in many cases the pagetable entry look-up can overlap with arbitration, making the miss penalty for in-cache translation equal to that for a cache with a translation lookaside buffer. Since the results of Chapter 4 showed that this extra cycle is significant, eliminating it makes in-cache translation even more attractive.

We also described the cache controller chips performance monitoring hardware. The 16 event counters measure the memory system behavior in 5 different modes. With this mechanism we can compute the miss ratio and effective access time metrics. In the next chapter, we use these counters to evaluate the performance of in-cache translation for real programs.

Chapter 6

Evaluation

In Chapter 4, we used trace-driven simulation to analyze the performance of in-cache address translation. The flexibility of simulation makes it easy to analyze many different cache designs. However, the simulation results are only as good as the address traces. The traces we used in Chapter 4 contain a total of 53 million references, equivalent to less than 10 seconds of execution time at SPUR's design speed. The longest individual traces contain 5 million references, representing less than 1 second of execution time. Since each trace captures such a short sample of a workload's execution, we cannot be sure that it accurately represents the typical cache behavior of the entire workload. Simulations using these traces may not accurately predict the true steady-state miss ratio.

In addition to the general limitations of trace-driven simulation, discussed in Chapter 4, we encounter other problems when we try to use the simulation results to predict the performance of the SPUR implementation of in-cache translation. None of the traces accurately represent the expected behavior of the SPUR processor. Even though the SPUR traces accurately model the memory reference stream, they only contain user-mode references and lack any multiprogramming behavior. Prior studies[20, 2] have shown that traces with these limitations often yield optimistic results. Both the Synapse and ATUM traces include system references and multiple processes, but are taken from systems with very different architectures than SPUR. The Synapse's MC68000 processor is only a 16-bit machine, while the VAX is the canonical complex-instruction-set computer. Because both instruction sets have compact encodings and use small register files, we expect the reference patterns to be quite different than those of a reduced-instruction-set computer like SPUR, which has 8 register windows. Also, the operating systems of the VAX and Synapse machines are different from Sprite. Agarwal has shown that 20% of all references occur in the kernel for his VMS traces, while 50% of all references are in the kernel for his Ultrix traces. The combined effects of these differences suggest that a sizable error may exist between our simulation results and the actual performance of the SPUR implementation of in-cache translation.

To understand how well the simulations predict the performance of in-cache translation, we must compare those results to measurements of the SPUR prototype. With a working

system, it is possible to measure many different applications, running for many millions, or even billions of references. Measurement evaluation lacks the flexibility of simulation: we can only evaluate the exact configuration implemented in SPUR. However, we eliminate all the weaknesses of the simulation input, increasing our confidence in the results.

In this chapter, we use SPUR's integrated event counters to evaluate the prototype's implementation of in-cache translation. We measure the performance of 23 workloads, and compare the results to the trace-driven simulations of Chapter 4. The measured results do not always agree with the simulation predictions. In those cases, we propose new hypotheses to explain the behavior, and run additional experiments to test them. With this experimental approach we are able to explain many of the differences.

We show that the total miss ratios from simulations and measurements differ by less than 10%, on average. These are very close considering the dissimilarity of the two workloads. However, the simulations do not predict in-cache translation performance with the same accuracy. We show that the pagetable entry miss ratios depend heavily upon the pagetable placement, as hypothesized in Chapter 4. We then examine effective access time, comparing the empirical, or measured, value to an analytic model based on event counts. The original model predicts within 1% in user mode, but much worse in kernel mode. This model should always under-predict the empirical effective access time, but in fact over-predicts it.

We show that non-cacheable pages cause the over-prediction, because the model does not accurately include their effects and the operating system uses them more heavily than expected. A second model incorporates this factor, and eliminates the over-prediction. However, the relative error of this model is still large, averaging over 4%, which is still greater than the predicted overhead of in-cache translation.

A second missing factor, cache controller register accesses, accounts for the bulk of the error. A third, and final, model predicts the effective access time within 1% on average, and never worse than 1.6%. With this accurate model, we show that in-cache translation accounts for less than 4% of the cycles. This compares favorably to 4.6% to 6% of the cycles for two implementations of the VAX architecture. This confirms the main simulation result: in-cache translation performs better than most translation lookaside buffer designs.

In Section 6.1 we describe the prototype environment, metrics, workloads, and experiment design. In the next section, we examine the preliminary experimental results and contrast them with the simulation results of Chapter 4. We identify the problems with these results and propose a second experiment. In Section 6.3 we describe the changes necessary to the event counters, and the effect of the additional factors on the miss ratio and effective access time results. In the final section, we summarize the results of the chapter.

6.1 Experimental Methodology

The measurements reported in this chapter were taken on a prototype SPUR system. A consequence of being a prototype is that this machine does not run reliably at its design

speed. Noise problems on the processor board reduced the mean time to failure to under an hour when operating at the 100ms design speed. Slowing down the clock greatly improves its reliability. For these measurements, the processor clock period is 150 nanoseconds (a 50% degradation) and the backplane bus clock is 125 nanoseconds (a 25% degradation).

In addition, the processor's on-chip instruction buffer (a small special-purpose cache for instructions) was not operational due to a design error. This caused the CPU to issue an instruction no faster than once every 3 cycles, resulting in an average of at least 3 cycles per instruction. If the instruction buffer were operational, we would expect each instruction to require roughly 1.5 cycles, on average (including data references and instruction buffer misses but not cache misses)[42]. While this problem slows the actual performance of SPUR substantially, it does not directly affect cache performance because we compute miss ratios from the number of events, not the number of cycles, and we define the effective access time as the number of cycles per reference, not per instruction. In Chapter 4 we ignored the effects of the instruction buffer entirely, and continue to do so in this chapter. We simply factor out the extra cycles from our measurements when computing the effective access time[1].

6.1.1 Metrics

As in Chapter 4, we use both the miss ratio and effective access time metrics. However, because we gather the measurements from a running system, there are some limitations in the metrics we can compute. In the analysis, we frequently used the translation miss ratio m_{trans} to determine the increase in the total miss ratio due to in-cache translation. This metric includes both misses to pagetable entries and misses due to collisions between pagetable entries and instructions and data. Unfortunately, the counters are only able to measure the pagetable entry misses, not the collision misses.

However, we know that every block of pagetable entries brought into the cache knocks out at most one block of instructions and data. Therefore, the translation miss ratio is always less than or equal to twice the pagetable miss ratio[2]:

$$\begin{aligned} m_{coll} &\leq m_{pte} \\ m_{trans} &= m_{pte} + m_{coll} \\ m_{trans} &\leq 2m_{pte} \end{aligned}$$

From the trace-driven simulations in Chapter 4, we know that m_{coll} is always less than m_{pte}, ranging from 29% to 68%, with mean of 40%. We use m_{pte} rather than m_{trans} throughout this chapter.

[1] This simply requires subtracting two times the number of instructions from the number of cycles.

[2] m_{coll} is only bounded by m_{pte} for direct-mapped caches. For set-associative caches, m_{coll} includes collisions between instructions and data blocks that occur because the pagetable entries reduce the effective capacity of the cache. Thus the bound on m_{trans} is higher for set-associative caches.

The effective access time also differs between this chapter and Chapter 4. The measurement results differ substantially from the simulation results because the analysis assumes idealized values for the operation times, rather than the actual values from the SPUR implementation. We discuss this in more detail in Section 6.2.4.

6.1.2 Workload

In any measurement study, the results depend heavily upon workload choice. No actual or synthetic workload accurately represents all workloads. However, by including important and commonly executed applications we maximize the relevance of the results.

Our workload, summarized in Table 6.1, consists of four groups of programs: *utilities*, *development*, *CAD*, and *multiprogramming*. The *utilities* group consists of 7 UNIX utilities commonly used for a wide range of tasks. The *development* group contains 5 software development applications: the SPUR C compiler, the SPUR Lisp compiler, the SPUR linker, a parallel version of the *make* program *(pmake)*, and a symbolic debugger. The *CAD* group consists of a timing analyzer, a cache simulator, a switch-level simulator, and a logic optimization program. We run the later program on 3 different input files of varying sizes. Finally, two of the *multiprogramming* workloads consist of 3 different UNIX utilities running concurrently. The other 3 are the *pmake* program run with 2, 3, and 4 concurrent processes, respectively.

Although one of SPUR's design goals was fast execution of Lisp, the SPUR Lisp system was never thoroughly debugged, so many Lisp applications do not run reliably on the prototype. As a result, we include only one Lisp application, the compiler, in our set of workloads. All the other workloads are written in the C programming language.

6.1.3 Experiment Design

While we can exactly reproduce simulations, measurements always have some random error that we cannot control. In addition, if we do not design the experiments carefully there can be systematic errors that bias our results.

One potential source of significant random error is the operating system's filesystem cache. This cache is similar in concept to the processor cache, but instead of holding blocks of instructions and data, it holds entire filesystem blocks (4 kilobytes in Sprite). An application's performance varies substantially depending upon how much of its code and data already reside in the cache. To control for this type of error, we flush the filesystem cache before beginning execution of each application.

A potential source of systematic error is the *idle loop*. When the processor is idle, it executes in a tight code loop waiting for an interrupt, thus the cache miss ratio during this time is not representative of real programs. To eliminate this type of error, Sprite disables

Group	Workload	Description
Utilities	awk	Runs an 86 line *awk* script over a 75,000 byte file.
	diff	Compares two 130,000 byte files using *diff*.
	grep	Searches for several patterns in a 130,000 byte text file.
	ls	Lists the contents of 3 large directories with *ls*.
	qsort	Sorts 250,000 random integers using the *qsort* library routine.
	sed	Runs a 9 line *sed* script over a 130,000 byte file.
	sort	Sorts 100,000 random numbers (in ASCII) using *sort*.
Development	cc	Compile and link a 400 line C program.
	gdb	Symbolic debugger executing the *esp1* application in batch mode.
	pmake1	Recompile 12 C files using *pmake* (executing sequentially).
	slc	Compile 4 of the Gabriel benchmarks three times with the SPUR Lisp compiler.
	sld	Link 107 object files to form an executable.
CAD	bdsim	Switch-level simulation of the SPUR cache controller using *n2verify2/bdsim* with a 100 vector input file.
	crystal	Determine the critical paths of the SPUR cache controller chip using *crystal*.
	esp1	Use *espresso* to optimize a PLA with 32 inputs, 16 outputs and 68 product terms.
	esp2	Use *espresso* to optimize a PLA with 14 inputs, 56 outputs and 104 product terms.
	esp3	Use *espresso* to optimize a PLA with 41 inputs, 36 outputs and 207 product terms.
	idinero	Simulate the SPUR cache on a 100,000 address trace using *idinero*.
Multiprogramming	mp1	Run *awk*, *sed*, and *diff* concurrently.
	mp2	Run *ls*, *grep*, and *sld* concurrently.
	pmake2	Run *pmake* with two concurrent processes.
	pmake3	Run *pmake* with three concurrent processes.
	pmake4	Run *pmake* with four concurrent processes.

Table 6.1: Summary of Application Programs in Workload

the counters while it executes in the idle loop[3].

To minimize bias from uncontrolled random error, we use a completely randomized experiment design[64]. We run 5 iterations of each application in each of the 3 cache measurement modes: User, Kernel, and Combined. Thus the 23 applications are each run 15 times, for a total of 345 data points. By randomly choosing the order of the 345 measurements, we significantly reduce the probability of biased results.

6.2 Preliminary Results

In this section we present preliminary results from measurements taken using the imbedded performance counters. We assume in this section that the counters measure all significant events, and compare these results to our simulation results from Chapter 4. The miss ratio results, with a few exceptions, are generally in line with our expectations. But the effective access time results clearly show that the counters exclude at least one important event. This leads to a supplemental study, described in Section 6.3.

6.2.1 Workload Characteristics

The 23 programs that make up our synthetic workload are all significantly longer than any single trace used in the simulation analysis. In fact, all but 2 applications generate more references than all traces combined. The shortest program issues 30 million references, while the longest generates over 14 billion. The mean execution length is 2.7 billion references, but the distribution is highly skewed, as indicated by a median of only 630 million references. Combined, the programs execute a total of 63 billion references, over three orders of magnitude more references than the traces contain.

As shown in Table 6.2, the programs vary substantially in their behavior. *esp3* is very compute intensive, and issues only 17% of its references in kernel mode. Conversely, *ls* and *grep* are very I/O intensive and generate over 65% of their references in kernel mode. On average, roughly a third of all references occur in the operating system kernel.

All the workloads also include the effects of context switches. Even those workloads that consist of a single program must compete for the processor with operating system *daemons*. The context switch interval defined as the number of references between full context switches, averages 139,000 references. The *qsort* application has the largest interval, with over 174,000 references between switches. This application consists of a very compute

[3]Because of a programming error, the counters were only re-enabled when a process was rescheduled. Thus interrupts that occur in the idle loop were not measured. To test the sensitivity of the results to this problem, we re-ran 4 of the most I/O intensive workloads (grep, ls, sed and awk) under a corrected kernel. The total number of kernel-mode references increased by as much as 11%. However, the miss ratio results were much closer, varying up or down by at most 1.6%, which is not statistically significant. Since this problem affects I/O intensive workloads much more than compute intensive workloads, we conclude that this error has a negligible impact on our results.

Workload	References (millions)	Percent Kernel Mode	Percent Instructions	Context Switch Interval (1000s)
awk	279	24	82	164
bdsim	12852	26	87	77
cc	247	35	81	136
crystal	14573	20	82	170
diff	266	29	72	169
esp1	559	20	86	166
esp2	2719	18	86	167
esp3	12044	17	86	166
gdb	759	25	84	156
grep	30	67	80	128
idinero	1707	18	80	169
ls	34	68	82	114
mp1	627	28	78	120
mp2	157	55	80	112
pmake1	3173	28	81	135
pmake2	3243	29	81	105
pmake3	3209	29	81	106
pmake4	3213	30	81	104
qsort	786	18	79	174
sed	69	33	84	152
slc	597	44	83	134
sld	103	48	80	139
sort	472	22	81	135
CAD	7409	20	84	153
Development	976	36	82	140
Multiprogramming	2090	34	80	109
Utilities	277	37	80	148
Average	2683	32	82	139

Table 6.2: Summary of Workload Characteristics

The workload consists of 23 application programs and scripts. This table summarizes the length of the application, in millions of references, the percent of references in kernel mode, the percent of references that are for instructions, and the number of references between context switches. We compute the unweighted arithmetic mean for each group, and also for all applications.

intensive single process, requiring very few I/O operations. The *bdsim* application has the shortest context switch interval, under 77,000 references. This is probably due to the close communication between the simulation process *(bdsim)* and the driver process *(n2verify2)*, which communicate using the Sprite *pipe* mechanism. For all workloads, the context switch interval is much larger than in our address traces: the ATUM traces have an average interval of 15,000 references and the Synapse traces average 34,000 references. The differences between operating systems, applications, processor speeds, and degrees of multiprogramming contribute to the wide spread of context switch intervals. Quantifying the effect of context switch intervals on our results is left for future research.

6.2.2 Basic Cache Performance

We look first at the basic cache performance, as summarized in Table 6.3. The miss ratio m_{cpu}, which includes only demand misses (excluding pagetable misses), varies widely between different programs. Looking first at user mode references, we see that *qsort* has the lowest miss ratio at 0.19%, while *Slc* has the highest at 1.69%. The mean user mode miss ratio is 0.69% and the median is 0.55%. The average is much lower than observed from simulation: even the user-mode-only SPUR traces have an average miss ratio of 1.06%.

The composition of the two workloads accounts for most of the difference. Comparing the one program common to both, *Slc*, the measurement and simulation workloads yield much closer results. Our simulations of the *Slc* traces, snapshots of the execution of *Slc*, predict a miss ratio of 1.54%. This falls within 10% of the measured user mode miss ratio of 1.69%. Several factors contribute to this small difference. First and most importantly, the two *Slc* address traces capture less than 1% of the user mode references generated by the running program. This small sample may not accurately model the real workload. Second, although the traces contain 5 million references and we exclude the initial 500,000 from the simulation metrics, the cache is large enough that the simulation miss ratios probably include some effects from cold-start transients. Cold-start behavior tends to increase the simulation miss ratio above the true miss ratio. Finally, although we do not count kernel mode references when we measure the user mode miss ratio, we cannot exclude them from the cache. Some of the user mode misses result because kernel references knock user instructions and data out of the cache. Thus our measured user mode miss ratio includes a component of user/kernel interference, which does not appear in the simulations of the SPUR traces. Despite these differences, the *Slc* simulation results are within 10% of the measured *Slc* miss ratio.

Since the *Slc* address traces contain only user mode references, the comparison between simulation and measurement results do not directly predict the performance of this program on the prototype. We must include kernel mode references as well, which exhibit very different behavior. While the average miss ratio is 0.69% in user mode, it is 2.78% in kernel mode, a factor of 4 difference. Over all applications, the kernel mode miss ratio ranges from as low as 1.95% to as high as 4.13%. Since roughly one third of all references occur in kernel mode, the kernel miss ratio significantly affects the combined miss ratio. The program with

Workload	User Mode	Kernel Mode	Combined
awk	0.00550	0.02518	0.01018
bdsim	0.00635	0.02305	0.01047
cc	0.00989	0.03221	0.01794
crystal	0.00531	0.02500	0.00931
diff	0.00539	0.02284	0.01042
esp1	0.00365	0.02181	0.00733
esp2	0.00451	0.02009	0.00727
esp3	0.00462	0.02051	0.00744
gdb	0.00384	0.02324	0.00870
grep	0.00739	0.03263	0.02458
idinero	0.00398	0.02396	0.00779
ls	0.00863	0.04133	0.03052
mp1	0.00674	0.02727	0.01243
mp2	0.00587	0.03404	0.02099
pmake1	0.01077	0.03132	0.01664
pmake2	0.01176	0.03212	0.01746
pmake3	0.01216	0.03232	0.01816
pmake4	0.01244	0.03239	0.01841
qsort	0.00190	0.01955	0.00509
sed	0.00269	0.03203	0.01219
slc	0.01688	0.03373	0.02435
sld	0.00470	0.03096	0.01713
sort	0.00312	0.02279	0.00741
CAD	0.00474	0.02240	0.00827
Development	0.00922	0.03029	0.01695
Multiprogramming	0.00979	0.03163	0.01749
Utilities	0.00494	0.02805	0.01434
Average	0.00687	0.02784	0.01401

Table 6.3: Demand Miss Ratio m_{cpu}

This table presents the results for m_{cpu} in all three cache measurement modes. They show that 1.4% of demand (CPU) references miss, on average.

the best performance, *qsort*, has only 18% of its references in kernel mode, for a combined miss ratio of 0.51%. At the other extreme, *ls* executes 68% of its references in kernel mode, for a combined miss ratio of 3.05%. On average, the combined miss ratio is nearly a factor of 2 more than the user mode miss ratio. Agarwal found similar results in his trace-driven simulations using the ATUM traces [2].

The kernel-mode miss ratio in SPUR is inflated by references to non-cacheable pages, since by design they always miss. In addition, while references to non-cacheable pages are counted as cache misses, they are not counted as cache references. This anomaly occurs because the reference begins as a normal cache miss, but is converted to a single word physical memory reference. We do not count direct physical memory references, so the reference to the non-cacheable page is also excluded. When we designed the counters, we expected the error from this anomaly would be small, because we *assumed* that non-cacheable pages would only be used to access I/O devices. As we show later, this assumption is incorrect and in fact non-cacheable pages are used to map user pages into the kernel's address space.

Taken as whole, the simulations accurately predict the actual demand miss ratio m_{cpu}. On average, the simulations predict a 1.32% demand miss ratio while we observe an average miss ratio of 1.40% on the prototype. This is a relative error of only 6%. The similarity of the results, despite the substantial differences in the two workloads, attest to the robustness of the simulation results.

6.2.3 Performance of In-Cache Translation

Trace-driven simulation analysis predicts a pagetable entry miss ratio m_{pte}, defined as pagetable entry misses divided by demand references, ranging from 0.023% to 0.166%, with a mean of 0.074%. Measurements of the SPUR hardware, presented in Table 6.4, are similar. The combined miss ratio ranges from 0.035% to 0.154%, with a mean of 0.087%. The difference between means is less than 15%, despite significant differences in the workloads. The simulation prediction falls well within one standard deviation of the measurement mean.

Focusing only on the SPUR traces, the simulation results range from 0.030% up to 0.166%, with a mean of 0.080%. The user mode measurements, comparable because the SPUR traces lack system references, range from 0.021% to 0.157%, with a mean of 0.073%. The difference between simulation and measurement means is less than 10%, again well within the standard deviation.

These results, summarized in Table 6.5, show that simulations accurately predict the performance of in-cache translation. However, the accuracy of the prediction of m_{pte} masks some interesting differences between measurement and simulations. In Table 6.5, we observe that in kernel mode, the pagetable miss ratio is only 1.7 times larger than in user mode. This seems small compared to the factor of 4 difference we see for m_{cpu} (in Table 6.3). This implies that the first-level pagetable entry miss ratio m_{upte}, defined as first-level pagetable entry misses divided by first-level pagetable entry references, is larger in user mode than in

Workload	User Mode	Kernel Mode	Combined
awk	0.00070	0.00095	0.00076
bdsim	0.00092	0.00158	0.00106
cc	0.00110	0.00136	0.00128
crystal	0.00028	0.00137	0.00049
diff	0.00060	0.00105	0.00074
esp1	0.00033	0.00095	0.00047
esp2	0.00021	0.00095	0.00035
esp3	0.00024	0.00103	0.00039
gdb	0.00032	0.00087	0.00046
grep	0.00074	0.00085	0.00078
idinero	0.00064	0.00189	0.00090
ls	0.00100	0.00090	0.00100
mp1	0.00070	0.00111	0.00082
mp2	0.00066	0.00099	0.00082
pmake1	0.00114	0.00160	0.00129
pmake2	0.00127	0.00164	0.00125
pmake3	0.00115	0.00168	0.00141
pmake4	0.00132	0.00173	0.00154
qsort	0.00023	0.00092	0.00036
sed	0.00039	0.00096	0.00055
slc	0.00157	0.00144	0.00154
sld	0.00049	0.00101	0.00079
sort	0.00084	0.00119	0.00096
CAD	0.00044	0.00129	0.00061
Development	0.00092	0.00126	0.00107
Multiprogramming	0.00102	0.00143	0.00117
Utilities	0.00064	0.00097	0.00074
Average	0.00073	0.00122	0.00087

Table 6.4: Pagetable Entry Miss Ratio m_{pte} Results

This table presents the pagetable entry miss ratio m_{pte} for the 23 workloads in each of the 3 measurement modes.

Source	Mean	Minimum	Maximum
User Mode	0.00073	0.00021	0.00157
Kernel Mode	0.00122	0.00085	0.00189
Combined	0.00087	0.00035	0.00154
All Traces	0.00074	0.00023	0.00166
SPUR Traces	0.00080	0.00030	0.00166

Table 6.5: Comparison between Measurement and Simulation Values of m_{pte}

This table compares the measured values of m_{pte} versus the simulation results. The top 3 values are average values for m_{pte} in each of the three measurement modes. The bottom two values are simulation results: the first is the average over all address traces, and the second is the average over only the user-mode-only SPUR traces.

kernel mode.

We show that this is true in Table 6.6. In user mode, m_{upte} ranges from 4.34% to as high as 26.3%, with a mean of 10.2%. In kernel mode, m_{upte} is much lower, ranging from 1.54% up to 5.11%, with a mean of 3.06%. Thus the average first-level pagetable miss ratio in user mode is over 3 times worse than in kernel mode.

The combined miss ratio lies between the two, of course, and ranges from 2.44% to 11.0%, with mean of 5.43%. This is 15% higher than the simulations predict: in the simulations, m_{upte} ranges from 2.43% to 8.53%, with a mean of 4.71%. However, the user-mode-only SPUR traces predict an average m_{upte} of 4.91%, while in user mode the measured mean m_{upte} is over 10%. The error in the simulation prediction is over 50%, much larger than for m_{cpu} or m_{pte}.

Why is user mode m_{upte} so much higher in actual execution than the simulation, and why is kernel mode m_{upte} so much lower? To understand the first question, we examine the differences in the two workloads. The most striking difference is that all SPUR address traces were taken from Lisp applications, while only one of the 23 measurement workloads uses Lisp. The one program in common, *Slc*, does not have such a striking difference between measurement and simulation. The user mode measurement is 8.3%. The simulation results for m_{upte} are 6.2% and 6.4% for the two *Slc* traces, a relative error of only 24%; this is much less than the 52% average error. Since the address traces only capture 1% of *Slc*'s user mode references, we must be cautious in our conclusions. Nonetheless, the similarity between the simulation and measurements of *Slc* suggest that the high average user mode m_{upte} is at least partially due to the differences between C and Lisp.

C programs use their address space very differently than Lisp programs. C programs

Workload	User Mode	Kernel Mode	Combined
awk	0.12324	0.02436	0.06541
bdsim	0.13434	0.04691	0.08541
cc	0.09816	0.02934	0.05565
crystal	0.04968	0.03596	0.04058
diff	0.10510	0.03331	0.06023
esp1	0.08388	0.02923	0.04925
esp2	0.04341	0.03037	0.03779
esp3	0.04691	0.03348	0.04065
gdb	0.07764	0.02504	0.04278
grep	0.09288	0.01825	0.02444
idinero	0.15434	0.05110	0.09552
ls	0.10719	0.01536	0.02589
mp1	0.09793	0.02839	0.05573
mp2	0.10320	0.02066	0.03072
pmake1	0.09340	0.03521	0.06253
pmake2	0.09520	0.03525	0.06024
pmake3	0.08832	0.03610	0.06332
pmake4	0.09369	0.03698	0.06627
qsort	0.11036	0.02984	0.05507
sed	0.11710	0.02064	0.03394
slc	0.08076	0.03091	0.05136
sld	0.09667	0.02326	0.03687
sort	0.26268	0.03424	0.10962
CAD	0.08542	0.03784	0.05820
Development	0.08933	0.02875	0.04984
Multiprogramming	0.09567	0.03148	0.05526
Utilities	0.13122	0.02514	0.05351
Average	0.10244	0.03062	0.05431

Table 6.6: Measurements of First-Level Pagetable Miss Ratio m_{upte}

This table presents the measurement results for m_{pte}, the first-level pagetable entry miss ratio. In user mode, m_{pte} averages 10.2%, varying from 4.34% to 26.3%. In kernel mode, it averages 3.06%, ranging from 1.54% to 5.11%. The combined miss ratio ranges from 2.44% to 11.0%, with mean of 5.43%.

locate their code, stack, and heap at the bottom of separate segments. This causes the first-level pagetable entries for all segments to map to the low region of the cache. As we saw in Chapter 4, contention for this region of the cache causes a significant number of set-conflicts. In contrast, the SPUR Lisp system allocates both code and data objects from the heap segment[95], spreading pagetable entries more widely across the cache. By spreading the pagetable entries across the cache, the Lisp system reduces the frequency of set-conflicts. Because the C programs map all their pagetable entries to the most congested region of the cache, their m_{upte} miss ratios are higher.

A similar factor partially explains why m_{upte} is lower in kernel mode than in user mode. In kernel mode, the process's code, stack[4], and heap regions all share the single system segment. As with the Lisp system, this tends to spread the pagetable entries more widely across the cache, reducing the frequency of set-conflicts.

A second factor is non-cacheable pages. References to regular pages only cause us to touch pagetable entries when they miss while every reference to a non-cacheable page generates a pagetable reference. Thus it seems likely that non-cacheable pages tend to decrease m_{upte} in kernel mode. However, we have *assumed* that the operating system uses non-cacheable pages infrequently, so we expect their impact to be low. As we show in Section 6.3, this assumption is incorrect.

The root pagetable entry miss ratio m_{rpte} also exhibits interesting behavior. We observed in Chapter 4 that m_{rpte} depends significantly upon where the root pagetable entries map into the cache. If they map to the first page of the cache, the miss ratio is very high, averaging over 50%. If they map halfway up the cache, the miss ratio drops by an order of magnitude. Since the global segment number, the high order address bits, determines the placement of the root pagetable entries in the cache, and the operating system assigns global segments in an arbitrary fashion, we expect to see a large variance in the measured values of m_{rpte}.

Table 6.7 summarizes the measured values of m_{rpte} for the 23 workloads. In user mode, the miss ratio ranges from 2.4% to 25.2%, with a mean of 9.0%. The miss ratio varies widely, even within the individual runs of a single workload. For example, of the 5 user mode runs of the program *cc*, m_{rpte} ranges from as low as 3.7% to as high as 34.9%, nearly an order of magnitude difference. This is similar to the behavior observed in the simulations of Section 4.3.3.

In kernel mode, m_{rpte} is more consistent. The system segment is always global segment 0, so the operating system's root pagetable entries always map to the first 1024 bytes of the cache. The miss ratio ranges from 37.8% to 57.8%, with a mean of 46.8%. The variance between individual runs of the same workload is much less: the greatest relative difference between minimum and maximum is only 22%, compared with 960% in user mode.

The kernel-mode measurements of m_{rpte} are quite close to the simulation results from the SPUR traces. The simulation miss ratio ranges from 24.3% up to 85.3%, with a mean

[4]Processes switch to a separate kernel stack when they enter the Sprite kernel.

Workload	User Mode	Kernel Mode	Combined
awk	0.02773	0.54728	0.13819
bdsim	0.07572	0.46319	0.18194
cc	0.11340	0.43943	0.27335
crystal	0.04506	0.52562	0.28538
diff	0.06538	0.38486	0.18678
esp1	0.07168	0.49874	0.29557
esp2	0.08721	0.55925	0.25998
esp3	0.12917	0.49509	0.28512
gdb	0.06607	0.49690	0.23553
grep	0.07871	0.42508	0.30426
idinero	0.04435	0.54235	0.20772
ls	0.07686	0.42286	0.26741
mp1	0.06748	0.43740	0.18331
mp2	0.08302	0.40443	0.27426
pmake1	0.11908	0.45472	0.23636
pmake2	0.12248	0.44995	0.19061
pmake3	0.07284	0.44409	0.22022
pmake4	0.11434	0.44219	0.24952
qsort	0.11225	0.57846	0.27890
sed	0.25161	0.45074	0.33141
slc	0.15071	0.37834	0.23242
sld	0.07455	0.40035	0.25631
sort	0.02423	0.52196	0.17743
CAD	0.07553	0.51404	0.25262
Development	0.10476	0.43395	0.24679
Multiprogramming	0.09203	0.43562	0.22358
Utilities	0.09097	0.47589	0.24063
Average	0.09017	0.46797	0.24139

Table 6.7: Measurements of Second-Level Pagetable Miss Ratio m_{rpte}

This table presents measurement results from m_{rpte}, the second-level pagetable entry miss ratio. In user mode, m_{rpte} ranges from 2.42% to 25.1%, with a mean of 9.01%. In kernel mode, it varies much less, ranging from 37.8% to 57.8% and averaging 46.8%. The combined miss ratio averages 24.1%, ranging from 13.8% to 33.1%.

of 49.1%. The difference between simulation and measurement means is less than 5%. Although the SPUR simulations include only user mode references, we compare them to kernel mode measurements, because in both cases the root pagetables map to the low region of the cache. As we saw in Chapter 4, the root pagetable mapping has the greatest effect on the miss ratio.

6.2.4 Effective Access Time Evaluation

The effective access time observed on the prototype is slower than simulation predictions because of the different durations required to perform certain operations. The simulations reflected an aggressive implementation, while the SPUR prototype was more conservative.

In Table 6.8, we compare SPUR's actual operation times to the simulation's idealized values. We determined the actual values by measuring a uniprocessor SPUR prototype using a logic analyzer. The measured memory access time was less than we described in Chapter 5 for two reasons. First, since there was only a single processor, it never needed to arbitrate for the bus[5], reducing the memory accesses by as much as 2 bus cycles. Second, we normalize the operation times to processor cycles, to account for the bus being faster than the processor. This reduces the cycle count but the operation times remain constant.

In Chapter 4 we assumed that all cache hits were treated equally. This is not the case in SPUR's implementation: while instruction fetches and data reads take one cycle, data writes take two. Writes may require additional overhead to maintain cache consistency; when writing to a block that is valid, but read-only (either the UnOwned or OwnShared states), an invalidation operation is needed to obtain ownership. This penalty, t_{wfi}, takes 9 cycles on the prototype.

Incorporating these differences, effective access time is modeled as:

$$\begin{aligned} t_{incache} = {} & f_{fetch}t_{fetch} + f_{read}t_{read} + f_{write}(f_{wfi}t_{wfi} + t_{write}) \\ & + m_{cpu}(1 - m_{upte})(t_{memory} + t_{upte-hit}) \\ & + m_{cpu}m_{upte}(1 - m_{rpte})(2t_{memory} + t_{rpte-hit}) \\ & + m_{cpu}m_{upte}m_{rpte}(3t_{memory} + t_{rpte-miss}) \end{aligned} \quad (6.1)$$

where f_{fetch}, f_{read}, and f_{write} are the fraction of references that are instruction fetches, data reads, and data writes, respectively, and f_{wfi} is the fraction of writes that require invalidation operations. The memory access time t_{memory} remains the same as in Chapter 4:

$$t_{memory} = t_{readblock} + f_{writeback}t_{writeback} \quad (6.2)$$

where $t_{readblock}$ is the time to get a block from memory and update the cache, $f_{writeback}$ is

[5]This is almost correct. The system contains one I/O board, an Ethernet controller. This controller does not support DMA, but does become bus master for one transaction whenever it generates an interrupt. However, this is clearly infrequent relative to cache misses.

Metric	Idealized Value	Actual Value	Description
t_{fetch}	1	1	Time to fetch an instruction (cache hit). Corrected to eliminate effects of instruction buffer.
t_{read}	1	1	Time to read a word (cache hit).
t_{write}	1	2	Time to write a word (cache hit).
t_{wfi}	n/a	9	Time to obtain ownership for a read-only block.
$t_{upte-hit}$	1	2	Overhead to lookup first-level pagetable entry.
$t_{rpte-hit}$	2	5	Overhead to lookup second-level pagetable entry.
$t_{rpte-miss}$	3	5	Overhead on second-level pagetable entry miss.
$t_{readblock}$	12	18	Time to read a block from memory.
$t_{writeback}$	12	19	Time to write a block back to memory.

Table 6.8: Actual Operation Times for SPUR Implementation

This table compares the actual operation times for the SPUR prototype used for these measurements to the idealized values used in the simulations. The actual values reflect the difference in bus and processor speeds. The effective access time model of Chapter 4 did not include the coherency protocol, and therefore does not depend upon t_{wfi}.

the fraction of blocks that must be written back to memory, and $t_{writeback}$ is the time to write a block back to memory.

Because we also measure the number of cycles required to execute a particular workload, we can also compute an empirical effective access time:

$$\hat{t}_{incache} = \frac{\text{Total Number of Cycles}}{\text{Total Number of CPU References}} \tag{6.3}$$

where the values are determined directly from the performance counters (we wired one of the external event pins to logic 1 to count cycles). We correct the numerator to compensate for the effects of the instruction buffer. Because the modeled effective access time $t_{incache}$ does not account for infrequent events, such as page faults and interrupts[6], we anticipate a small difference between model predictions and empirical results. Because the model *excludes* certain events, it should always *under-predict* the empirical effective access time.

In Table 6.9, we compare the $t_{incache}$ model to the empirical $\hat{t}_{incache}$. In user mode the model is quite accurate, predicting an effective access time of 0.6% below $\hat{t}_{incache}$, on average. Only 6 of the 23 workloads have an error greater than 1%. In the worst case, $t_{incache}$ is only 1.5% worse than the empirical $\hat{t}_{incache}$. We conclude that for user mode accesses the model is very accurate.

Unfortunately, in kernel mode the model is not as good: nearly one third of the workloads have an error exceeding 5%. Since simulation results predict that in-cache translation accounts for only 2.3% of the cycles, on average, this model is clearly too inaccurate to be useful. At one extreme, $t_{incache}$ for workload *esp3* is 7% lower than the empirical effective access time. Although this program has the error with the greatest magnitude, it is not the worst case. Seven of the 23 workloads *over-predict* the empirical effective access time. For example, the workload *ls* over-predicts $\hat{t}_{incache}$ by 6%. This is very odd, because the model should never predict more than the true value.

Since the model is accurate in user mode, we hypothesize that there must be some operation that the operating system performs that causes the inaccuracy in $t_{incache}$. But while there are several events the model does not include, these tend to make the model under-predict $\hat{t}_{incache}$. To explain the over-prediction, we must find either an event that is counted incorrectly (i.e., a counter counts too many events), or an operation that takes less time on average than we measured on the prototype.

As we have hinted earlier, non-cacheable page references are the culprit. We count non-cacheable page accesses as cache misses but not as cache references. Also, non-cacheable page references take less time to handle than regular cache misses because they only access a single word rather than an eight-word block. If there are many non-cacheable page references, then we have incorrectly assumed that we can ignore these inaccuracies. For applications that cause the kernel to use non-cacheable pages frequently, the model will

[6]The event counters include references generated by the page fault and interrupt handlers, but the model does not account for additional cycles that arise when the exception occurs.

Workload	$error = 100 * \frac{t_{incache} - \hat{t}_{incache}}{\hat{t}_{incache}}$		
	User Mode	Kernel Mode	Combined
awk	-1.07346	-3.25912	-1.76904
bdsim	-0.77626	-5.02983	-2.19005
cc	-0.27429	0.59648	0.17633
crystal	-0.36028	-5.35345	-1.70860
diff	-0.01763	-1.81133	-0.65762
esp1	-1.18959	-4.62575	-2.11157
esp2	-1.03818	-6.54733	-2.38289
esp3	-1.00981	-7.00297	-2.43739
gdb	-1.00263	-3.40918	-1.78667
grep	-0.76205	4.06006	2.88160
idinero	-0.31295	-5.63634	-1.62231
ls	-0.73017	6.01808	4.50990
mp1	-0.42355	-1.55059	-0.83164
mp2	-0.85748	3.54961	1.91426
pmake1	-0.22293	-1.12312	-0.53482
pmake2	-0.16487	-0.98708	-0.57563
pmake3	-0.18889	-1.02308	-0.46423
pmake4	-0.12995	-1.09262	-0.43500
qsort	-1.53666	-4.94400	-2.37832
sed	-0.39699	1.41733	0.41005
slc	-0.59122	1.36257	0.33788
sld	-0.96374	2.11494	0.81254
sort	-0.60789	-3.86173	-1.51564
Minimum	-1.53666	-7.00297	-2.43739
Maximum	-0.01763	6.01808	4.50990

Table 6.9: Difference between $t_{incache}$ and $\hat{t}_{incache}$

This table compares the modeled effective access time $t_{incache}$ with the empirical effective access time $\hat{t}_{incache}$. In user mode the error is small, less than 1.5%. However, it is much worse in kernel mode, ranging from as low as -7.0% to 6.0%. The latter value is odd, since the model should never over-predict the effective access time.

tend to over-predict the empirical effective access time.

We provided non-cacheable pages specifically to support I/O devices, allowing the operating system to map control registers into the virtual address space. Since these control registers are accessed infrequently, we assumed that non-cacheable page references would also be infrequent. However, the Sprite operating system also makes a page non-cacheable when it maps a user process's page into the kernel's address space: a simple but inefficient way to prevent virtual address synonyms (we discuss alternatives in Chapter 3). Sprite maps a page whenever it must copy or initialize it, as when a process *forks* or when a page fault occurs. Since references to mapped pages occur much more frequently than references to I/O control registers, ignoring non-cacheable pages introduces significant error into the effective access time model. In the next section, we extend the model to include this factor and revise our results.

6.3 Revised Results

In this section, we revise our analysis by including the effects of non-cacheable pages. We show that non-cacheable page references account for as much as 48% of demand misses in kernel mode. Similarly, these misses represent a large fraction of the references to pagetable entries. We characterize their effect on miss ratios, and estimate the improvement that would result from making mapped pages cacheable.

Then we include non-cacheable page references into the effective access time model. This factor improves the model by completely eliminating the over-prediction problem. However, the revised model still under-predicts the true effective access time by as much as 8%. We show that cache controller register reads and writes account for the bulk of the discrepancy; these operations occur much more frequently than expected because of software that compensates for a hardware design error in the CPU. When we include this additional factor into the model the average error drops below 1%. Finally, we use the revised model to analyze the performance of in-cache translation, and show that address translation accounts for less than 4% of all cycles.

Since the pre-programmed counters do not measure non-cacheable page references or cache controller register operations, we need to use the external event counters. We measure non-cacheable page references by counting the number of single word read and write operations on the backplane. During normal execution, these operations only arise from non-cacheable pages[7]. Cache controller register operations are detected by monitoring the cache interface signals generated by the CPU. In both cases, the necessary control signals are fed into a programmable logic device (PAL) which generates a logic "1" when the event occurs. The outputs of the PAL are connected to the external event pins on the cache controller chip.

[7]Direct physical address access to memory or the Ethernet board only occurs at boot-time or when the system crashes.

We reran all 23 workloads an additional 5 times, to measure these two new events. As before, we randomized the execution order to minimize bias from uncontrolled random error. In following sections we combine the results from these additional runs with the original data to compute revised metrics.

6.3.1 Basic Cache Performance

Including the effects of non-cacheable pages changes our miss ratio results in kernel mode. Since non-cacheable pages are not exposed to the user, they have no effect on the user-mode-miss ratio. Even in kernel mode the effect on the demand miss ratio m_{cpu} is small, because we already counted the non-cacheable page accesses as misses, just not as references. Including these references reduces m_{cpu} by a relative difference at most 2%; the change exceeds 1% for less than a third of the workloads.

However, non-cacheable page references account for a large fraction of all kernel mode misses. In one case, *ls*, 71% of data misses and 48% of all demand misses occur to these pages. The average is more moderate, but still high: 41% of data misses and 17% of all misses occur to non-cacheable pages. Obviously, these misses substantially increase the kernel mode miss ratio.

In Table 6.10, we consider the miss ratio excluding references to non-cacheable pages, and compare this to the original and corrected values of m_{cpu}. Excluding these references reduces the miss ratio by as much as 47%. For 18 of the 23 workloads, the miss ratio is reduced by over 20%. On average, the miss ratio for cacheable pages is 27% less than m_{cpu}. If Sprite could cache mapped pages, we expect the miss ratio to drop significantly.

Let us estimate the benefit of caching mapped pages (with the requisite flushing operations to maintain consistency). When Sprite maps a page, it always reads or writes the entire page. It follows that each cache block (eight words) would generate one miss, rather than a miss for each word in the block. We call this estimate of the miss ratio $\hat{m}_{cpu}$:

$$\hat{m}_{cpu} = \frac{\text{Demand misses} - (7/8 * \text{Non-cacheable page references})}{\text{Demand references}} \tag{6.4}$$

Since non-cacheable pages account for a significant fraction of the misses, reducing them by a factor of 8 should lead to a substantial reduction in the miss ratio.

We compare this estimate of the miss ratio with the measured miss ratio in Table 6.10. In the best case, caching mapped pages reduces the miss ratio by 42%, in the worst case, the reduction is only 7.6%. The average reduction over all workloads is 24%. Since kernel references account for roughly one-third of all references, we can expect a significant improvement in the total miss ratio.

This estimate is somewhat optimistic, because it ignores the purging effect that caching mapped pages would have on the cache. Mapping pages as non-cacheable has the one advantage that the cache is not disturbed during full page copies. Caching these pages has the disadvantage of purging useful blocks from the cache. In the worst case, although

Workload	Fraction Non-Cacheable Page Misses	Orignal m_{cpu}	Corrected m_{cpu}	m_{cpu} without Non-Cacheable Pages	Estimated $\hat{m}_{cpu}$
awk	0.22657	0.02518	0.02504	0.01947	0.02007
bdsim	0.11022	0.02305	0.02299	0.02051	0.02077
cc	0.32342	0.03221	0.03188	0.02180	0.02286
crystal	0.11206	0.02500	0.02493	0.02220	0.02249
diff	0.24326	0.02284	0.02271	0.01728	0.01788
esp1	0.21649	0.02181	0.02171	0.01709	0.01760
esp2	0.11375	0.02009	0.02004	0.01780	0.01804
esp3	0.08782	0.02051	0.02047	0.01871	0.01890
gdb	0.23414	0.02324	0.02312	0.01780	0.01838
grep	0.43809	0.03263	0.03217	0.01831	0.01981
idinero	0.12190	0.02396	0.02389	0.02104	0.02135
ls	0.48013	0.04133	0.04052	0.02144	0.02346
mp1	0.25818	0.02727	0.02708	0.02023	0.02096
mp2	0.40712	0.03404	0.03357	0.02018	0.02161
pmake1	0.25772	0.03132	0.03107	0.02325	0.02406
pmake2	0.25266	0.03212	0.03186	0.02401	0.02482
pmake3	0.25143	0.03232	0.03206	0.02419	0.02501
pmake4	0.26441	0.03239	0.03212	0.02383	0.02469
qsort	0.23062	0.01955	0.01946	0.01504	0.01553
sed	0.38687	0.03203	0.03164	0.01964	0.02093
slc	0.31327	0.03373	0.03338	0.02315	0.02422
sld	0.37275	0.03096	0.03061	0.01942	0.02063
sort	0.23968	0.02279	0.02267	0.01732	0.01791
CAD	0.10811	0.02240	0.02234	0.01956	0.01986
Development	0.27416	0.03029	0.03001	0.02108	0.02203
Multiprogramming	0.26080	0.03163	0.03134	0.02249	0.02342
Utilities	0.27855	0.02805	0.02774	0.01836	0.01937
Average	0.17489	0.02784	0.02761	0.02016	0.02096

Table 6.10: Comparison of Corrected Kernel Mode m_{cpu} to Original and Estimate

This table incorporates the non-cacheable page results into the kernel mode demand miss ratio computation. The second column presents the fraction of misses due to non-cacheable pages. The third and fourth columns compare the original values of m_{cpu} with the corrected values. Column 5 presents the miss ratio excluding non-cacheable page references (and misses). The last column presents $\hat{m}_{cpu}$, our estimate for the miss ratio if mapped pages were made cacheable.

caching these pages reduces the miss ratio it can actually *increase* the effective access time because regular cache misses take longer than non-cacheable page references. Determining the performance impact of this effect is an area for future research.

6.3.2 In-Cache Translation Performance

We recall from the preliminary measurements that m_{upte}, the miss ratio for first-level pagetable references, was surprisingly low in kernel mode. We hypothesized that this was attributable to two factors: non-cacheable page references and the spreading of pagetable entries throughout the cache. With the measurements of non-cacheable pages, we can estimate their effect on m_{upte}.

Under Sprite's usage of mapped pages, the major source of non-cacheable page references, the pages are always read or written completely. If mapped pages are made cacheable, then their first-level pagetable entries would be accessed once for each block rather than once for each word (ignoring the effect of set conflicts). This has the effect of reducing the denominator of m_{upte} by 7/8 times the number of non-cacheable page references. We estimate the miss ratio as:

$$\hat{m}_{upte} = \frac{\text{First-level pagetable entry misses}}{\text{First-level pagetable entry references} - (7/8 * \text{Non-cacheable page references})} \tag{6.5}$$

Since the numerators of m_{upte} and $\hat{m}_{upte}$ are the same, $\hat{m}_{upte}$ is always greater than or equal to m_{upte}.

In Table 6.11, we compare the estimated miss ratio $\hat{m}_{upte}$ to the measured miss ratio m_{upte}. m_{upte} ranges from a low of 1.54% to a high of 5.11%, with an average of 3.06%. $\hat{m}_{upte}$ is somewhat higher, ranging from 2.65% up to 5.79%, with an average of 3.9%. As expected, caching mapped pages increases the first-level pagetable miss ratio. The increase averages 22%, but ranges from as low as 8.3% for the large compute intensive program *esp3*, to as high as 73% for the utility program *ls*.

The estimated kernel mode miss ratio $\hat{m}_{upte}$ is still much lower than the measured miss ratio in user mode. Factoring out non-cacheable page references increases the kernel mode miss ratio from 3.06% to 3.9%, but it still does not approach the average user mode miss ratio of 10.2%. Thus non-cacheable page references are not a major contributor to the difference. This result supports the second hypothesis: that m_{upte} is sensitive to how the pagetable entries map into the cache. Spreading the allocation of memory throughout each segment, rather than simply allocating from the bottom, reduces the pagetable entry miss ratio.

Workload	m_{upte}	Estimated $\hat{m}_{upte}$	Percent Increase
awk	0.02436	0.03038	24.7%
bdsim	0.04691	0.05191	10.6%
cc	0.02934	0.04090	39.4%
crystal	0.03596	0.03988	10.9%
diff	0.03331	0.04236	27.2%
esp1	0.02923	0.03604	23.3%
esp2	0.03037	0.03374	11.1%
esp3	0.03348	0.03626	8.32%
gdb	0.02504	0.03150	25.8%
grep	0.01825	0.02968	62.6%
idinero	0.05110	0.05719	11.9%
ls	0.01536	0.02651	72.6%
mp1	0.02839	0.03672	29.3%
mp2	0.02066	0.03211	55.4%
pmake1	0.03521	0.04547	29.1%
pmake2	0.03525	0.04527	28.4%
pmake3	0.03610	0.04629	28.2%
pmake4	0.03698	0.04811	30.1%
qsort	0.02984	0.03739	25.3%
sed	0.02064	0.03120	51.2%
slc	0.03091	0.04261	37.9%
sld	0.02326	0.03451	48.4%
sort	0.03424	0.04340	26.7%
CAD	0.03784	0.04250	11.0%
Development	0.02875	0.03900	26.3%
Multiprogramming	0.03148	0.04170	24.5%
Utilities	0.02514	0.03442	26.9%
Average	0.03062	0.03911	21.7%

Table 6.11: Estimated Effect of Non-Cacheable Pages on Kernel Mode m_{upte}

This table compares the measured m_{upte} with an estimated miss ratio $\hat{m}_{upte}$ if mapped pages were made cacheable. We estimate that the miss ratio would increase by an average of 22%.

6.3.3 Effective Access Time

Model Including Non-Cacheable Pages

The original effective access time model, $t_{incache}$, over-predicts the empirical effective access time $\hat{t}_{incache}$ in many cases. We hypothesize that this is due to non-cacheable pages. We suspect non-cacheable pages because references to these pages are excluded from the total reference count, and because each reference is counted as a full cache miss. Accesses to non-cacheable pages average about 12 cycles (writes require 11 cycles, reads 13 cycles) while a regular cache miss takes 18 cycles. Both factors tend to increase the modeled effective access time[8].

We propose a new model t_{ncp} that incorporates the true cost of non-cacheable page references. We compare this new model to the original model and $\hat{t}_{incache}$ in Table 6.12. We see that non-cacheable page references explain the over-prediction of the original model: t_{ncp} is less than $\hat{t}_{incache}$ for all 23 workloads. The new model never over-predicts the empirical effective access time.

This new model still has a problem, however. The average of the absolute value of the error is actually *larger* than the original model. In the worst case, t_{ncp} under-predicts the empirical value by almost 8%. On average, the model under-predicts by over 4%. The average error of this model is larger than the overhead of in-cache translation, making it too inaccurate to be useful.

Since the model always under-predicts $\hat{t}_{incache}$, the error must occur because t_{ncp} does not include some frequently executed operation. The model is accurate in user mode, so the missing operation is probably a privileged instruction. Based on this hypothesis, we might expect that programs that spend most of their time in the operating system would have the largest error. However, as we show in Figure 6.1, the opposite is actually true. The more time a workload spends in kernel mode, the lower the model's error.

A possible explanation for this unusual behavior is that the error is a linear function of the application's total execution time. The error then appears larger for those workloads that spend a smaller percentage of their time in the kernel.

This is indeed the case. Like most operating systems, Sprite uses an interval timer to generate periodic interrupts. In addition to the usual scheduling functions, Sprite must perform certain operations that refresh dynamic nodes in the CPU chip. These nodes exist because of a design error, but do not cause a problem as long as they are refreshed periodically.

Because these nodes must be refreshed frequently, an interval timer generates an interrupt every 6 ms. The interrupt handling code reads and clears the fault and interrupt status registers on the cache controller. Both effective access time models exclude these cache controller read and write operations. Since these interrupts occur regardless of whether the

[8]The non-cacheable page references were excluded from the denominators of both $\hat{t}_{incache}$ and $t_{incache}$. We correct both in this section.

Workload	$\hat{t}_{incache}$	Percent Error $t_{incache}$	Percent Error t_{ncp}
awk	1.71	-3.26	-5.25
bdsim	1.72	-5.03	-5.91
cc	1.80	0.60	-2.85
crystal	1.73	-5.35	-6.32
diff	1.71	-1.81	-3.75
esp1	1.66	-4.63	-6.32
esp2	1.65	-6.55	-7.38
esp3	1.67	-7.00	-7.65
gdb	1.67	-3.41	-5.35
grep	1.73	4.06	-0.84
idinero	1.72	-5.64	-6.65
ls	1.86	6.02	-0.26
mp1	1.78	-1.55	-3.91
mp2	1.78	3.55	-1.06
pmake1	1.81	-1.12	-3.78
pmake2	1.83	-0.99	-3.63
pmake3	1.83	-1.02	-3.66
pmake4	1.84	-1.09	-3.86
qsort	1.62	-4.94	-6.60
sed	1.77	1.42	-2.73
slc	1.82	1.36	-2.09
sld	1.75	2.11	-1.80
sort	1.69	-3.86	-5.80
Minimum	1.62	-7.00	-7.65
Maximum	1.86	6.02	-0.26
Average Absolute Value	1.75	3.32	4.24

Table 6.12: Revised Effective Access Time Model t_{ncp}

This table compares the original effective access time model $t_{incache}$ to t_{ncp}, which includes the effect of non-cacheable pages. While the original model over-predicted $\hat{t}_{incache}$ in many cases, the new model never does. However, the average absolute value of the error is over 4%, making it too large to be useful for analyzing in-cache translation.

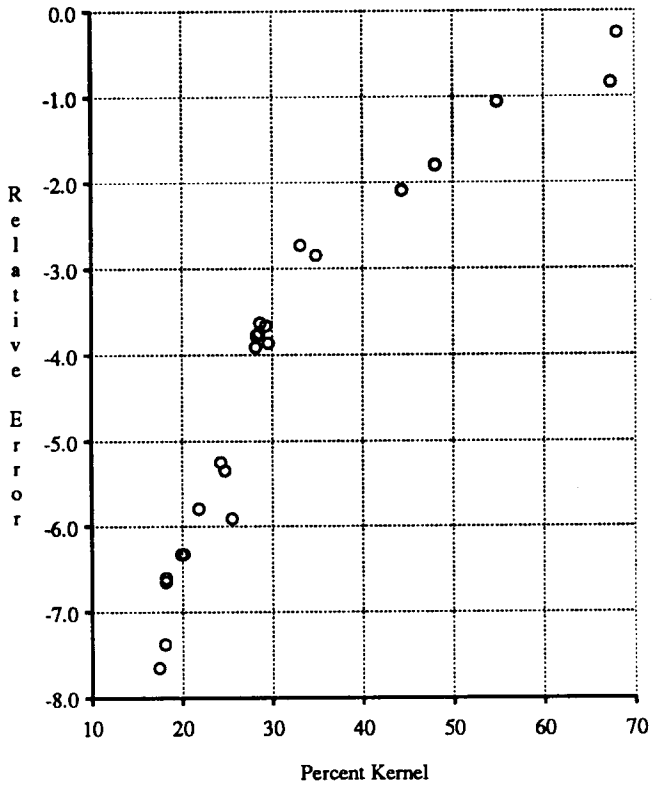


Figure 6.1: Relative Error of t_{ncp} versus Percentage of References in Kernel Mode

This scatter plot compares the relative error of the revised model t_{ncp} against the percentage of references in kernel mode. Since the error decreases as the percentage increases, we hypothesize that the error is a function of the application's total execution time.

system is executing in user or kernel mode, the error is a linear function of the execution time. In compute bound workloads, the interrupt handler, which always runs in kernel mode, accounts for a large fraction of all kernel time; hence, excluding cache controller register operations leads to a large error in these cases.

Model Including Register Operations

We incorporate this new factor into a third model for effective access time, t_{reg}. We assume that each cache controller register operation requires an average of 4.5 cycles to execute, since reads require 4 cycles and writes require 5. As shown in Table 6.13, this new model is much more accurate than the previous two. The average error is less than 1% and is never more than 1.6%. The model over-predicts the empirical effective access time in a few cases, but the magnitude of the error is sufficiently small that we ignore it.

This model for effective access time is accurate enough that we can use it to determine the overhead of in-cache address translation for the SPUR implementation. In Table 6.14, we break the effective access time into its main components. The basic access time, t_{cpu}, includes the effects of cache hits, cache misses, invalidation operations, non-cacheable page references, and cache controller register operations. This is similar to t_{ideal} used in Chapter 4, but includes the additional factors discussed in this chapter. t_{cpu} also includes the effects of collision misses, which are excluded from t_{ideal}, but as we showed in Chapter 4, this component is very small, averaging less than .005 cycles per reference. Even with the slower miss penalty of the SPUR implementation, this factor should remain less than .01 cycles per reference. In-cache translation has two major contributions to the effective access time: $t_{penalty}$, the increase in the basic cache miss penalty from looking for the pagetable entry in the cache, and $t_{overhead}$, the overhead due to pagetable entry misses. The total effective access time, t_{reg}, combines the in-cache translation overhead with the basic effective access time.

From Table 6.14, we see that t_{cpu} ranges from as low as 1.17 for the program *qsort*, to as high as 1.60 for the *ls* workload. The average is 1.34 cycles per reference. This is much greater than t_{cpu} for the simulations, where $t_{cpu} = t_{ideal} + t_{coll}$. From the earlier results, in Table 4.6, we see that the average for the simulations is 1.22 cycles per reference, a difference of 10%. But the simulations assumed an implementation more aggressive than we built for SPUR. Revising the simulations to use the actual operation times, the simulation average for t_{cpu} increases to 1.32% cycles per reference, predicting less than 2% below the observed value.

The overhead of in-cache translation consists of two components: $t_{penalty}$ and $t_{overhead}$. $t_{penalty}$ ranges from 0.010 to 0.060 cycles per reference, averaging 0.028. $t_{overhead}$ is generally lower, ranging from 0.0078 to 0.035 cycles per reference, averaging 0.020. $t_{overhead}$ is larger than $t_{penalty}$ for only 3 of the 23 workloads. On average, $t_{penalty}$ accounts for 63% of the total overhead of in-cache translation. These results are slightly higher than the simulation results using the actual operation times: $t_{penalty}$ averages 0.026 and $t_{overhead}$ averages 0.016.

Workload	$\hat{t}_{incache}$	Percent Error $t_{incache}$	Percent Error t_{ncp}	Percent Error t_{reg}
awk	1.71	-3.26	-5.25	-0.98
bdsim	1.72	-5.03	-5.91	-1.45
cc	1.80	0.60	-2.85	0.31
crystal	1.73	-5.35	-6.32	-1.30
diff	1.71	-1.81	-3.75	-0.09
esp1	1.66	-4.63	-6.32	-1.07
esp2	1.65	-6.55	-7.38	-1.60
esp3	1.67	-7.00	-7.65	-1.62
gdb	1.67	-3.41	-5.35	-0.96
grep	1.73	4.06	-0.84	1.08
idinero	1.72	-5.64	-6.65	-1.31
ls	1.86	6.02	-0.26	1.64
mp1	1.78	-1.55	-3.91	-0.22
mp2	1.78	3.55	-1.06	1.21
pmake1	1.81	-1.12	-3.78	-0.11
pmake2	1.83	-0.99	-3.63	-0.05
pmake3	1.83	-1.02	-3.66	-0.12
pmake4	1.84	-1.09	-3.86	-0.24
qsort	1.62	-4.94	-6.60	-1.05
sed	1.77	1.42	-2.73	0.49
slc	1.82	1.36	-2.09	0.51
sld	1.75	2.11	-1.80	0.70
sort	1.69	-3.86	-5.80	-0.93
Minimum	1.62	-7.00	-7.65	-1.62
Maximum	1.86	6.02	-0.26	1.64
Average Absolute Value	1.75	3.32	4.24	0.83

Table 6.13: Revised Effective Access Time Model t_{reg}

This table compares the relative errors for the three effective access time models: $t_{incache}$, t_{ncp}, and t_{reg}. The first model, $t_{incache}$, both over-predicts and under-predicts $\hat{t}_{incache}$ by a large margin. The second, t_{ncp}, corrects the over-prediction, but has a larger average error. The last model, t_{reg}, reduces the average error to under 1%. Although this model does over-predict $t_{incache}$ in some cases, the magnitude of the error is small enough to ignore.

Workload	t_{cpu}	$t_{penalty}$	$t_{overhead}$	t_{reg}	$\frac{t_{reg}}{t_{cpu}}$
awk	1.27188	0.02033	0.01746	1.30967	1.02971
bdsim	1.26925	0.02092	0.02507	1.31524	1.03624
cc	1.40905	0.03575	0.02925	1.47405	1.04613
crystal	1.25666	0.01861	0.01072	1.28599	1.02334
diff	1.36166	0.02081	0.01932	1.40179	1.02947
esp1	1.18851	0.01465	0.01083	1.21399	1.02144
esp2	1.18798	0.01453	0.00777	1.21027	1.01877
esp3	1.19099	0.01487	0.00887	1.21473	1.01993
gdb	1.22175	0.01738	0.01080	1.24993	1.02307
grep	1.51344	0.04868	0.01858	1.58070	1.04444
idinero	1.23005	0.01558	0.02034	1.26597	1.02920
ls	1.60423	0.06023	0.02412	1.68858	1.05258
mp1	1.35709	0.02482	0.02040	1.40231	1.03332
mp2	1.46236	0.04167	0.01979	1.52382	1.04203
pmake1	1.38699	0.03320	0.02919	1.44937	1.04498
pmake2	1.40494	0.03483	0.02846	1.46823	1.04505
pmake3	1.41966	0.03623	0.03202	1.48791	1.04807
pmake4	1.42456	0.03674	0.03479	1.49608	1.05021
qsort	1.17248	0.01018	0.00880	1.19146	1.01618
sed	1.27505	0.02429	0.01303	1.31236	1.02926
slc	1.53270	0.04847	0.03548	1.61664	1.05477
sld	1.40164	0.03406	0.01923	1.45494	1.03802
sort	1.22180	0.01480	0.02278	1.25938	1.03075
CAD	1.22057	0.01653	0.01394	1.25103	1.02496
Development	1.39043	0.03377	0.02479	1.44899	1.04212
Multiprogramming	1.41372	0.03486	0.02709	1.47567	1.04382
Utilities	1.34579	0.02847	0.01773	1.39199	1.03433
Average	1.33760	0.02790	0.02031	1.38580	1.03604

Table 6.14: Effective Access Time Breakdown for Combined User and Kernel Modes
This table breaks down the effective access time into its main components: t_{cpu}, $t_{penalty}$, and $t_{overhead}$. t_{cpu} is the basic effective access time, excluding translation misses (but including collision misses). $t_{penalty}$ is due to the extra time in-cache translation spends looking for pagetable entries. $t_{overhead}$ is the overhead incurred when pagetable entries are not in the cache.

In the last column of Table 6.14, we compare the total effective access time t_{reg} to the basic effective access time t_{cpu}. We see that the overhead of in-cache translation increases the effective access time by 3.6% on average, ranging from as low as 1.6% to as high as 5.5%. This is higher than the 1.9% increase (over $t_{ideal} + t_{coll}$) seen in the simulation results of Chapter 4. However, when we factor in the higher operation times of the SPUR implementation, the simulations predict an increase in the effective access time of 3.2%, which is close to the observed increase.

In other words, for a large and diverse workload, the SPUR implementation of in-cache translation accounts for under 4% of the total cycles, on average. To put this in perspective, Clark, et al., found in their study of the VAX 8800 that the translation lookaside buffer accounted for 4.6% of the total cycles[18]. In an earlier study, Clark and Emer found that the translation lookaside buffer accounted for roughly 6% of the cycles on the VAX 11/780[20]. Compared to these two commercial implementations, even this prototype implementation of in-cache translation performs well.

6.4 Summary

In this chapter we addressed three major issues in the evaluation of in-cache address translation. First, we described a methodology for evaluating the cache performance of a running system using imbedded performance counters. We showed that despite careful planning, our preprogrammed counters did not measure all the events needed to accurately characterize performance. Fortunately, the two external event counters enabled us to measure these other factors with only minimal additional hardware. Second, we compared the results of the measurements using these counters to the simulation results. We showed that when we include the actual SPUR operation times, the simulations predict an effective access time within 2% of the measured results, despite the differences in the two workloads. Finally, we showed that the performance of the SPUR implementation of in-cache translation compares favorably with two successful commercial translation lookaside buffer designs.

Imbedding a set of event counters into a custom VLSI chip has two advantages over traditional external performance monitors. First, an external monitor may not have access to the necessary state information, requiring an integrated design. Second, with an integrated approach, *every* system has a set of counters, extending the performance monitoring capability to all users, at no additional cost.

The main limitation of on-chip counters is that we must choose the events to count when we design the chip. This requires careful planning, and some assumptions about the behavior of the completed system. As we showed in this chapter, these assumptions are not always correct. This experience suggests that an imbedded performance monitor should provide at least a limited expansion capability, to help cope with the unexpected. Fortunately, we anticipated the possibility of such a problem and included two external event counters in the SPUR design.

Using the imbedded counters, we measured the cache performance of 23 workloads.

These workloads generate a total of 63 billion references, over 3 orders of magnitude more references than the address traces used in Chapter 4. The basic cache performance of the prototype is quite close to the simulation results: the simulations predict a 1.32% demand miss ratio on average, while we observed a 1.40% miss ratio. A relative error of 6% is small considering the differences between the two workloads.

The simulation results for in-cache translation do not predict the measurements as well. The two results for the pagetable entry miss ratio m_{pte} are within 15%; however, this obscures the fact that the first-level pagetable entry miss ratio m_{upte} is much higher than expected in user mode and much lower in kernel mode. Since the simulation prediction for the one program in common, *Slc*, is much more accurate, we suspect that the workload compositions cause the difference. We hypothesized that m_{upte} is very sensitive to the placement of pagetable entries into the cache. C programs tend to map their pagetable entries to the low region of the cache, where high contention increases the set conflicts. Conversely, Lisp programs and the operating system spread their pagetable entries more widely across the cache, reducing set-conflicts.

We also observed a significant variation in the root pagetable entry miss ratio m_{rpte}. Again, this is due to the sensitivity of in-cache translation's performance to the placement of pagetable entries in the cache. This observation confirms the simulation results of Chapter 4. This is a weakness of in-cache translation, which we address in the next chapter.

In this chapter we computed both the empirical effective access time, the total number of cycles divided by the number of references, and modeled effective access times, where we estimated the number of cycles by weighting each event. Our original model, $t_{incache}$, not only had a large error in kernel mode, but often over-predicted the empirical effective access time. We showed that the over-prediction arose because the operating system uses non-cacheable pages when it maps pages into the kernel address space. A second model, t_{ncp}, includes the effects of non-cacheable pages. However, while this model eliminates the over-prediction, the magnitude of its error is actually larger. We showed that cache controller register operations account for most of the error, and included this factor into a third model, t_{reg}. The error of this last model averages less than 1%. Finally, we used this model to analyze the effective access time of in-cache translation. The effective access time results differ substantially between this chapter and Chapter 4. However, the difference in the operation times between the simulations and measurements accounts for most of the discrepancy. Revising the simulation results to use the actual SPUR operation times brings the prediction within 2% of the measured values.

The main result of this chapter is that in-cache translation works well compared to commercial translation lookaside buffer designs. For a large set of important programs, the prototype implementation of in-cache translation accounts for less than 4% of the total cycles. This compares to the 4.6% and 6% overhead observed for the VAX 8800 and the VAX 11/780 translation lookaside buffer implementations. A more aggressive implementation, using the optimizations described in Chapter 5, could further reduce this overhead, by a factor of 2 or more. We conclude from these results that in-cache translation is at least

competitive with, and probably superior to, traditional translation schemes.

Chapter 7

Alternatives

In the previous chapters, we have shown that in-cache translation is an attractive alternative to traditional translation lookaside buffers. However, it has two minor problems we would like to eliminate: the high translation miss ratio for some applications, and the potentially poor memory utilization of the pagetable itself. In this chapter we discuss these problems and show that both depend upon the pagetable structure, which is organized as an array. We propose a variation of in-cache translation which maintains the virtual-physical mapping in a hashtable, or inverted pagetable. We use trace-driven simulation to show that this new variation reduces the translation miss ratio (m_{trans}) by over 50% for the SPUR traces and SPUR cache configuration. However, this new scheme does not work as well for the ATUM traces, but the effective access times are no more than 1% worse than the original. The largest potential performance improvement comes from reducing the physical memory requirements of the pagetable, which improves performance by reducing the paging rate.

In the first section we discuss the two problems with in-cache translation, and describe the general approach to solving them. In the next section we describe the proposed variation of in-cache translation in detail. In Section 7.3 we analyze the implementation complexity of the two schemes. Finally, we compare the performance of the two using trace-driven simulation.

7.1 Problems with In-Cache Translation

We have shown in previous chapters that in-cache translation performs well compared to translation lookaside buffer designs. However, we have also shown that the performance is sensitive to how a process utilizes its address space. The allocation of memory affects where the active pagetable entries map into the cache, which in turn affects the miss ratio. For example, C programs generally have high translation miss ratios because they allocate memory from the bottom of each of their segments. This causes their active pagetable entries to map to the lower region of the cache, which tends to have higher contention. Conversely, Lisp applications and the operating system kernel have lower translation ratios

because they spread the allocation of memory across their segments, and hence spread their pagetable entries more uniformly across the cache.

We showed in Chapter 4 that increasing the associativity of the cache reduces pagetable entry misses more than it reduces demand misses. Thus caches with higher degrees of associativity are less sensitive to the pagetable entry mapping. However, as we discussed in the earlier chapter, increasing the associativity may decrease total system performance by increasing the cache access time, and therefore is not generally attractive. An alternative solution is to modify in-cache translation so that the translation miss ratio is less sensitive to the memory access pattern.

A second problem with in-cache translation is that the self-referential pagetables do not use physical memory efficiently. Under worst-case conditions, the pagetable consumes as much as 50% of the physical memory. Such inefficient use of main memory is clearly undesirable, and can seriously degrade performance by increasing paging activity.

The inefficiency arises because of internal fragmentation. The pagetable is implemented as an array in the virtual address space, which is broken into pages. If any pagetable entry within a page is valid, then the entire page must be resident in physical memory. As long as the address space is densely populated, then most of the pagetable entries in each page (of pagetable entries) are valid, wasting little space. However, if the address space is sparsely populated, as can occur under Lisp and object-oriented systems, then serious internal fragmentation can occur. In the worst case, each page of pagetable entries contains only a single valid entry, thus the pagetable consumes half of the physical memory[1].

A similar situation arises when the workload consists of many small processes, as in some real-time systems[65]. Each process's stack and heap generally requires only a single page each. If the system must allocate an entire page to hold this single valid pagetable entry, as SPUR does, then the minimum process size is twice what is theoretically necessary. As with object-oriented systems, the internal fragmentation wastes up to 50% of the physical memory.

Both the high translation miss ratio and the space inefficiency problems hinge on the organization of the pagetable. Because the pagetable is implemented as an array mapped into the virtual space, the pagetable entries for the bottom of each segment map to the bottom of the cache. Since this region of the cache tends to have the greatest activity, the pagetable entries are frequently knocked out. By changing the pagetable to a different data structure, we can change the placement of pagetable entries in the cache. Similarly, changing the pagetable's data structure improves the space efficiency by reducing internal fragmentation.

One traditional solution to pagetable fragmentation is to use base and bounds registers to limit the size of the array. While this works well for densely populated address spaces, it does not help with sparse allocation. Also, it is more expensive to implement because it

[1]We assume that the pagetable entry must be resident in main memory if the page is resident. This is required by most operating systems.

requires addition rather than simply concatenation.

A second approach, commonly used in microprocessor designs, supports a multi-level pagetable with variable size pages. This data structure is similar to a B$\star$-tree, except that each node in the tree contains an array which is indexed with the appropriate bits from the virtual address. When translating an address, we traverse the tree beginning at the root and ending at a leaf. Using small pages improves the space efficiency of the data structure. Conversely, smaller pages increase the depth of the tree, increasing the translation time. This approach allows the operating system to make the time/space trade-off. A drawback of this approach is that reducing the space requirement significantly slows the performance of translation.

A better solution is used in the IBM System/38[21], also implemented in the IBM RT PC[39, 14]. In this approach, the pagetable is implemented using a hashtable. Hashtables efficiently support sparse representations, providing fast random lookups with low space overhead. Because the hash function randomizes the placement of pagetable entries into the cache, the translation miss ratio is less sensitive to how processes allocate their address spaces.

7.2 Inverted In-Cache Translation

In this section, we propose a variation on in-cache translation, called *inverted in-cache address translation*, that uses a hashtable representation for the pagetable (also known as an *inverted pagetable*). We first describe the translation process, by examining the cases that occur while looking for a hashtable entry. Then we describe the hardware needed to implement the translation algorithm. Finally, we describe the procedures for inserting and deleting hashtable entries.

Under inverted in-cache translation, cache hits continue as normal without translation. On a cache miss, rather than indexing into an array, we hash the virtual page number and probe into the hashtable. There are three cases that arise on a probe. First, we find the entry in the cache (a hit) and the hash keys are equal. In this case, translation is complete and we transfer the requested data from memory. In the second case, we find the entry in the cache, but the keys do not match (a hash collision). We resolve the collision by *reprobing*: computing a new index into the hashtable and trying again. To take advantage of the relative costs of cache hits and misses, we simply increment the old index (modulo the hashtable size) to compute the new hash index. The final case occurs when the hashtable entry is not in the cache. To resolve the miss, we must determine the physical address of the hashtable entry. Since the hashtable is small relative to the physical memory size, we simply require that the hashtable be contiguous in physical space, and point to its base with a special register. Thus resolving the miss simply requires concatenating the hash index with the base register contents, and transferring the hashtable entries from main memory. At this point we repeat the probe (and possible reprobes) as above.

Finding an entry in the hashtable is straight-forward, unless it isn't there. If it's not,

we must detect this case and trap to software to handle the page fault. To determine that a page is not in the hashtable, we must be able to detect the end of a collision chain. We do this by including an *end-of-chain* bit in each entry. If the keys do not match and the end-of-chain bit is set, then the controller generates a page fault. Figure 7.1 presents pseudo-code for the translation process.

```
Translate(targetVA)
{
    addr pteVA;

    pteVA = Hash(targetVA);
    while(true) {
        if (Hit(pteVA)) {
            /* Hash table entry is in the cache */
            /* Check that its the right one */
            if (Tag(targetVA) == PteVtag(pteVA) && PteValid(pteVA)) {
                /* Translation is complete, Get the Data */
                ReadDataFromMemory(TargetPA(pteVA));
            }
            else if (EndOfChain(pteVA)) {
                PageFault(pteVA);
            }
            else {
                pteVA = Rehash(pteVA);
            }
        }
        else {
            /* Hashtable entry isn't in the cache; Fetch it from Memory */
            ReadDataFromMemory(HashPA(pteVA));
        }
    }
}
```

Figure 7.1: Pseudo-Code for Inverted In-Cache Translation Algorithm

Figure 7.2 illustrates the pagetable address computation. The hash operation is a simple Boolean function (discussed below) of the virtual page number (the high order bits) of the global virtual address. The *mask* register specifies the size of the hashtable by masking off the unused high-order bits of the hash index. Note that since we use bit masking, the hashtable size is restricted to powers of two. The *vbase* register contains the virtual

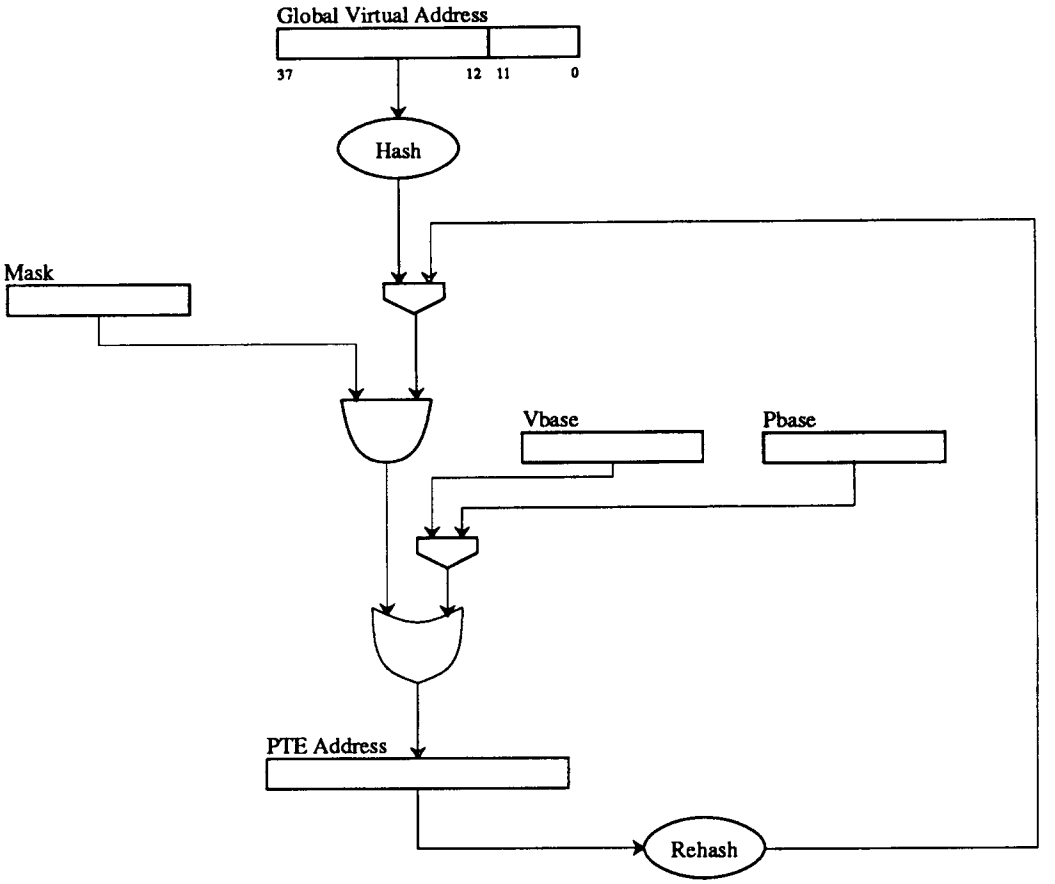


Figure 7.2: Inverted In-Cache Translation Address Mapping

This figure illustrates the inverted in-cache translation address computation. The virtual page number is first hashed, then masked and concatenated with the base virtual address of the hashtable. If a hash collision occurs, the rehash function (an incrementer) computes a new offset into the hashtable. If the entry is not in the cache, then the physical address of the hashtable entry is constructed by concatenating the base physical address to the hash offset.

address of the hashtable. We concatenate this with the hash index to form the pagetable entry virtual address. Again, to simplify the hardware we restrict the pagetable to be properly aligned. The *rehash* function simply increments the hash index: to handle the end condition correctly we must re-mask the result. Finally, the *pbase* register contains the physical address of the hashtable. We simply concatenate the contents of this register with the hash index to locate the hashtable entry in main memory.

The mask register and associated and/or logic permits the hashtable size to vary under control of the operating system. This is necessary to keep the average number of reprobes low. A well-known theoretical result shows that the expected number of probes to find an entry is $(2-\alpha)/(2-2\alpha)$, where α is the fill factor[43]. If the hashtable is 4 times larger than the data it contains, then α is 0.25 and the average number of probes is less than 1.2. Of course, this assumes that the hash function produces uniformly distributed indices for all inputs. Earlier research has shown that division by a prime number produces high quality hash indices. Unfortunately, it is hardly feasible to perform division on the critical path of a cache controller. Instead, we chose a simple bit-folding function that combines bit fields with the exclusive-or operator ($\oplus$):

$$hash(gva) = (gva \gg (p-3)) \oplus (gva \gg (p+7)) \oplus (gva \gg 15) \tag{7.1}$$

where *gva* is the global virtual address, p is the log_2 of the page size, and $\gg$ is the right-shift operator. We designed this function for the SPUR address space, and it produces acceptable results for the SPUR traces. Because the hash function includes the high-order bits, pages from different segments are less likely to collide in the hashtable. However, as we show in Section 7.4, this function does not produce good results for all traces.

A software handler is responsible for all insertion and deletion operations, just as in SPUR in-cache translation. To prevent inconsistencies, we must synchronize updates to the pagetable. The handler must change the hashtable carefully, since other processors read the table during the update. To insert a new entry, the handler simply follows the collision chain until it reaches the end. If the next entry is free, it uses it. If not, it turns off the end-of-chain bit (combining two chains together) and continues searching for a free entry. When it finally finds a free entry following the end of the chain, it allocates this entry, sets its end-of-chain bit, then clears the end-of-chain bit in the preceding entry.

To delete an entry, the handler locks the table then marks the entry invalid. If the entry is at the end of a chain greater than length one, we simply mark the preceding entry as the new end-of-chain, and return. If the entry is in the middle of a chain, then the procedure is more complicated. The basic idea is to copy the end-of-chain entry to the newly freed location, then deallocate the tail. Unfortunately, it isn't quite that simple since a chain may have multiple entry points (this occurs when we combine two chains during allocation). By moving the end-of-chain entry to the free slot, we could incorrectly move the entry past its entry point. Thus, we must carefully check each entry's hash index to make sure we don't make this mistake, and to split the collision chain when necessary. There is a small race condition where one processor searches the chain for a particular entry while the handler

Module	Spur In-cache	Inverted In-cache
Sequencer	41 inputs	43 inputs
	36 outputs	37 outputs
	80 states	76 states
	207 product terms	206 product terms
Decoder	14 inputs	14 inputs
	56 outputs	52 outputs
	103 product terms	99 product terms
Datapath	11 Registers	5 Registers
	226 Bits of Register	130 Bits of Register
	10 Bit Shifter	Hash Function
		Rehash Function
		Mask Logic

Table 7.1: Implementation Statistics for Inverted and SPUR In-cache Translation

This table compares the implementation statistics for inverted in-cache translation to SPUR in-cache translation. We assume that the hashtable size may vary from a low of 4 kilobytes up to 1 megabyte.

is moving it. Fortunately, this race condition only causes an unnecessary page fault, which the operating system can easily handle.

7.3 Implementation Complexity

Inverted in-cache translation has similar implementation complexity to regular in-cache translation. Both the datapath and control complexity require roughly the same number of transistors. Both forms of in-cache translation are simpler to implement than translation lookaside buffers.

The datapath component of inverted in-cache does not require the shift and concatenate circuit used in SPUR. In addition, it needs 6 fewer registers, totaling 130 bits rather than 226 bits. On the other hand, inverted in-cache requires both the hashing and rehashing functions, the later being an incrementer, and the mask logic. The total number of transistors is roughly the same in both designs, and both datapaths are small.

The control logic for inverted in-cache requires fewer states, reducing the sequencer from 80 to 76 states. But this only reduces the product terms from 207 to 206, because one

state must detect an additional case (reprobe)[2]. The area of the sequencer PLA actually increases because it needs two additional inputs and one additional output. The decoder PLA decreases by 4 outputs and 4 product terms (103 to 99). These two factors tend to cancel out each other, leaving the control area of the two approaches roughly equal.

Inverted in-cache translation must also compare the hash keys. This requires a 26-bit comparator, to determine if the virtual page number of the requested address matches the pagetable entry. This also requires that pagetable entries be 8 bytes, to accommodate the virtual page number. We assume that the cache access path is 64 bits, as in SPUR, so only one access is necessary to read the entire pagetable entry.

The differences between the two designs are summarized in Table 7.1. Both designs are simple to implement, and require only a small amount of chip and board resources.

7.4 Performance of Inverted In-cache Translation

In this section, we use trace-driven simulation to analyze the performance of inverted in-cache translation. We use the same traces and methodology described in Chapter 4. We compare both the miss ratio and effective access time metrics of the new scheme to the original algorithm. We show that inverted in-cache translation reduces the translation miss ratio by over 50% for the SPUR traces, and over 10% overall. However, this has only a small effect on system performance, and the two schemes have effective access times within 1% of each other. Also, we show that although the original hash function performs well for the SPUR and Synapse traces, it requires many reprobes for the ATUM traces. We present an alternative hash function that performs better for these traces.

In this comparison, we mark the inverted translation metrics with a prime ($'$) character. For example, m_{trans} and m'_{trans} are the translation miss ratios for regular in-cache translation and inverted in-cache translation, respectively.

Table 7.2 summarizes the performance of inverted in-cache for the SPUR cache configuration. For 10 of the 14 traces, the translation miss ratio is lower for inverted in-cache. The ratio of the translation miss ratios, m'_{trans}/m_{trans}, ranges from a low of 0.37 to as high as 1.36. On average, the ratio of ratios is 0.86. The greatest improvement occurs for the SPUR traces: the ratio of translation miss ratios ranges from 0.37 to 0.67, with an average of 0.47. In other words, for these traces, the translation miss ratio for inverted in-cache translation is only half that for SPUR in-cache translation. The ATUM traces do not perform as well; m'_{trans} is 36% higher than m_{trans} for the *Mul8* trace.

Because translation misses are a small component of the miss ratio, the difference in the total miss ratio is less significant. The ratio of the total miss ratios $m'_{incache}/m_{incache}$ averages 0.99. The SPUR traces benefit most: from as much as 10% better for *Rsim*, to as little as 1% better for *Weaver*. Overall, the SPUR traces average 4% better under inverted

[2]Also, the state assignment was optimized for the original specification. Re-assigning the state values would probably reduce the number of terms slightly.

Trace	m_{ideal}	m_{trans}	m'_{trans}	$\frac{m'_{trans}}{m_{trans}}$	$\frac{m'_{incache}}{m_{incache}}$
mul2	0.02427	0.00162	0.00171	1.01410	1.00353
mul8	0.02237	0.00168	0.00214	1.35814	1.01883
savec	0.01076	0.00071	0.00080	1.02800	1.00787
dec0	0.01334	0.00055	0.00064	1.03504	1.00644
forf	0.01783	0.00123	0.00112	0.96937	0.99446
rsim	0.00180	0.00044	0.00023	0.39836	0.90429
rsim2nd5M	0.01288	0.00086	0.00066	0.59798	0.98530
slc	0.01863	0.00230	0.00124	0.46187	0.94940
slc2nd5M	0.01329	0.00183	0.00095	0.42835	0.94196
weaver	0.00781	0.00059	0.00051	0.66667	0.99105
weaver2nd5M	0.00916	0.00069	0.00031	0.36612	0.96210
devel	0.00263	0.00038	0.00030	0.71372	0.97418
prod5M	0.01106	0.00083	0.00069	0.76475	0.98882
prod2nd5M	0.01211	0.00114	0.00089	0.71651	0.98117
ATUM_MP	0.01913	0.00134	0.00155	1.16466	1.01033
ATUM_UP	0.01559	0.00089	0.00088	0.99013	0.99951
SPUR	0.01059	0.00112	0.00065	0.47456	0.96013
Synapse	0.00860	0.00078	0.00063	0.73385	0.98366
Average	0.01348	0.00103	0.00093	0.86202	0.99282

Table 7.2: Performance of Inverted In-cache Translation

This table compares the miss ratios of in-cache translation, m_{trans} and $m_{incache}$, to the miss ratios of inverted in-cache translation, m'_{trans} and $m'_{incache}$. The first section of the table gives the results for each of the individual traces, separated into their respective groups. The second section of the table presents the averages for each group, and the final section gives the overall average. The results show that inverted in-cache performs very well for the SPUR and Synapse traces, but has comparable miss ratios for ATUM_UP, and worse miss ratios for ATUM_MP.

in-cache translation. The ATUM traces, particularly the ATUM_MP traces, generally perform better for regular in-cache translation. Nonetheless, the total miss ratio for inverted in-cache is less than 2% worse for any of the traces.

These results are not surprising given the results of Chapter 4. We showed that set conflicts cause 95% of the translation misses for the SPUR traces. This compares to 81% for the Synapse traces, 67% for the ATUM_UP traces, and only 33% for the ATUM_MP traces. This explained why increasing the associativity improves the performance of the SPUR traces more than the others. Since hashing the pagetable address also tends to reduce set conflicts, the SPUR traces also benefit more from inverted in-cache translation than the other traces.

As always, miss ratio is not the whole story. The effective access time of inverted in-cache is also a function of the average number of hash probes.

$$\begin{aligned} t'_{incache} &= t_{cache} \\ &\quad + m'_{cpu}((1 - m'_{pte})(t'_{pte-hit} + t_{memory} + m'_{reprobe}t'_{reprobe}) \\ &\quad + m'_{pte}(t'_{pte-miss} + 2t_{memory} + m'_{reprobe}t'_{reprobe})) \end{aligned} \tag{7.2}$$

where $m'_{reprobe}$ is the average number of reprobes per lookup, and $t'_{reprobe}$ is the time to execute the reprobe. If we assume equivalent implementations for both SPUR and inverted in-cache, then their effective access times differ by the factor $m'_{cpu}m'_{reprobe}t'_{reprobe}$ (as well as the differing miss ratios). Thus the relative performance depends on the values of $t'_{reprobe}$ and $m'_{reprobe}$.

In an aggressive implementation, $t'_{reprobe}$ takes only a single cycle. Under the SPUR implementation philosophy, however, $t'_{reprobe}$ requires 2 cycles because of the control pipeline delay. In this chapter, we assume an aggressive implementation to facilitate comparisons with the analysis in Chapter 4. The average number of reprobes $m'_{reprobe}$ depends upon the quality of the hash function. A function which uniformly distributes the pagetable entries in a random fashion tends to minimize the number of reprobes.

As we see in Table 7.3, our hash function achieves mixed results. The number of reprobes per lookup averages 0.05 for the SPUR traces and 0.11 for the Synapse traces. But the ATUM traces perform much worse. Both the ATUM_UP and the ATUM_MP traces average more than 1.5 reprobes per lookup, with a worst case (for *mul8*) of 3.6 reprobes per lookup.

The number of reprobes significantly affects the effective access time. The SPUR traces perform 0.5% better under inverted in-cache translation, and the Synapse traces perform almost equally well under the two schemes. The large number of reprobes, coupled with the slightly higher translation miss ratio, makes inverted in-cache perform worse for the ATUM traces. The *mul8* trace performs over 6% worse with inverted in-cache translation. On average the ATUM_UP and ATUM_MP traces perform 2% and 2.8% worse, respectively.

We examined whether an alternative hash function would improve the performance for the ATUM traces. This function exclusive-ors random pairs of bits from the virtual page number to form the hash index. As shown in Table 7.4, this new function does a better job

Trace	$t_{incache}$	$t'_{incache}$	$m'_{reprobe}$	$\frac{t'_{incache}}{t_{incache}}$
mul2	1.43567	1.45664	0.85195	1.01460
mul8	1.41956	1.50619	3.60774	1.06103
savec	1.19466	1.20265	0.65102	1.00669
dec0	1.23506	1.25127	1.05946	1.01312
forf	1.33817	1.37347	2.13260	1.02638
rsim	1.04123	1.03852	0.05463	0.99739
rsim2nd5M	1.20931	1.20685	0.00057	0.99797
slc	1.30857	1.29760	0.14922	0.99162
slc2nd5M	1.22703	1.21651	0.05913	0.99143
weaver	1.12092	1.11994	0.00316	0.99912
weaver2nd5M	1.14282	1.13808	0.00354	0.99586
devel	1.04505	1.04416	0.00177	0.99915
prod5M	1.18265	1.18325	0.21600	1.00051
prod2nd5M	1.20654	1.20435	0.10517	0.99818
ATUM_MP	1.34996	1.38849	1.70357	1.02854
ATUM_UP	1.28661	1.31237	1.59603	1.02002
SPUR	1.17498	1.16958	0.04504	0.99541
Synapse	1.14475	1.14392	0.10765	0.99928
Average	1.23908	1.25359	0.86307	1.01171

Table 7.3: Effective Access Time of Inverted In-cache Translation

This table compares the effective access times of inverted in-cache translation to the original algorithm. In addition to having higher miss ratios, the ATUM traces also require large numbers of hashtable reprobes. This makes inverted in-cache perform worse for these traces. Conversely, the SPUR and Synapse traces generally perform better, although never by more than 1%.

Trace	m_{trans}	m'_{trans}	$m'_{reprobe}$	m''_{trans}	$m''_{reprobe}$
mul2	0.00162	0.00171	1.85195	0.00266	1.12501
mul8	0.00168	0.00214	4.60774	0.00286	1.21382
savec	0.00071	0.00080	1.65102	0.00106	1.22270
dec0	0.00055	0.00064	2.05946	0.00182	1.08469
forf	0.00123	0.00112	3.13260	0.00180	1.11013
ATUM_MP	0.00134	0.00155	2.70357	0.00219	1.18718
ATUM_UP	0.00089	0.00088	2.59603	0.00181	1.09741

Table 7.4: Miss Ratio and Reprobes for Alternative Hash Function

This table compares the performance of the two hash functions for the ATUM traces. The alternative hash function decreases the average number of reprobes for the ATUM_MP traces from 1.7 to 0.19. However, the translation miss ratio increases from 0.16% to 0.22%. Metrics computed from the new hash function are marked with two prime characters ($''$), while metrics from the first hash function are marked with only one ($'$).

Trace	$t_{incache}$	$t'_{incache}$	$t''_{incache}$	$\frac{t''_{incache}}{t_{incache}}$
mul2	1.43567	1.45664	1.44991	1.00992
mul8	1.41956	1.50619	1.43717	1.01241
savec	1.19466	1.20265	1.20144	1.00568
dec0	1.23506	1.25127	1.25247	1.01409
forf	1.33817	1.37347	1.34542	1.00542
ATUM_MP	1.34996	1.38849	1.36284	1.00954
ATUM_UP	1.28661	1.31237	1.29894	1.00958

Table 7.5: Effective Access Time for Alternative Hash Function

This table compares the effective access time of inverted in-cache translation using the alternative hash function to the original hash function and SPUR in-cache. Metrics computed from the new hash function are marked with two prime characters ($''$), while metrics from the first hash function are marked with only one ($'$).

of randomizing the virtual address, which in turn reduces the number of reprobes ($m''_{reprobe}$). The ATUM_MP traces average only 0.19 reprobes per lookup and the ATUM_UP traces average 0.10. This brings $m''_{reprobe}$ for the ATUM traces in line with $m'_{reprobe}$ for the other traces.

Unfortunately, this new hash function also increases the miss ratio. In the worst case, the *dec0* trace, the translation miss ratio with the new function, m''_{trans}, is nearly 3 times worse than with the old. On average, m''_{trans} is over a factor of two worse than m'_{trans}.

However, reducing the number of reprobes is more important than the increase in the miss ratio. As shown in Table 7.5, the alternative hash function reduces the effective access time for 4 of the 5 traces. The effective access time for *dec0* increases by only 0.1%. The worst case for the new hash function is 1.4% worse than regular in-cache translation, compared to 6% with the old hash function. On average, the effective access time is less than 1% worse than regular in-cache translation.

These results show that the performance of inverted in-cache translation is very sensitive to the choice of hash function. But with a good hash function, the performance is within 1% of regular in-cache translation for the ATUM traces, and 0.5% better for the SPUR traces. Additional work is needed to determine whether a better hash function could improve the performance of the ATUM traces.

7.5 Summary

Inverted in-cache translation is a variation on the original design. It uses a different data structure for the pagetable: a hashtable rather than an array. The hashtable greatly reduces the physical memory requirements of the pagetable when the workload consists either of many small processes or of processes with large sparse address spaces. By improving the utilization of physical memory, inverted in-cache translation can prevent unnecessary paging and thereby improve performance.

In this chapter, we show that inverted in-cache is no harder to implement than regular in-cache translation. The datapath is still very simple, requiring only 5 registers and a simple function unit. The cache controller requires very few changes to support inverted in-cache, and the area of the controller remains essentially constant.

We also show that the two schemes have very similar performance when we ignore paging behavior. For the SPUR cache configuration, the translation miss ratio for the SPUR traces is 50% lower under inverted in-cache translation. With the original hash function, the effective access time of the SPUR traces averages 0.5% less than regular in-cache, while the Synapse traces are essentially equal. The ATUM traces perform poorly with the original hash function, generating a large number of hash collisions. But an alternative hash function eliminates this problem, and brings the performance within 1% for the ATUM traces.

These results show that the two schemes perform within 1% of each other, and are equally simple to implement. Since inverted in-cache translation is less sensitive to how

processes use the address space (with a good hash function) and the pagetable uses physical memory more efficiently, we believe that this alternative may be superior to the original design. Since the measurements of C programs in Chapter 6 showed higher translation miss ratios than the SPUR Lisp traces, we hypothesize that the actual benefit of inverted in-cache is greater for these programs. Further research is needed to test this hypothesis and develop high quality hash functions that are easily implemented in hardware.

Chapter 8

Support

In the preceding chapters we focussed on in-cache address translation, analyzing its performance, studying its implementation, and examining a variation of the original algorithm. In this chapter we broaden our focus, and study an issue – reference and dirty bits – relevant to all virtual address caches, rather than specific to in-cache translation. We apply our experimental approach to this problem, going through the entire design-analyze-prototype-evaluate process.

Most virtual memory systems use reference and dirty bits to help optimize page replacement policies. A page's reference bit is logically set on each reference to the page, and its dirty bit is logically set on each write to the page. Operating systems use reference bits to keep the pages in approximate least-recently-used order, to prevent the page daemon from replacing frequently used pages. Dirty bits tell the page daemon whether it must write a page back to secondary storage before replacing it, thus reducing the write-back traffic.

Systems with physical address caches usually use a translation lookaside buffer to translate virtual addresses to physical addresses. This buffer, accessed on every reference, provides a convenient place to cache reference and dirty bits[1]. Systems with virtual address caches do not have this luxury; since they only access the translation information on cache misses they require additional hardware support to maintain accurate reference and dirty bits[94].

In this chapter we examine several different approaches to maintaining reference and dirty bits in systems with virtual address caches. These schemes require different levels of hardware and software support, and their performance depends upon the workload. In the next section we study 4 alternative ways to implement dirty bits, using a simple analytic model, confirmed by measurements from the SPUR prototype, to analyze their performance. We show we can efficiently emulate dity bits using the standard protection mechanism. In Section 8.2 we examine 3 alternatives to maintaining reference bits. We use measurements

[1]Although logically set on each reference or write, most systems check the bits first and only set them when necessary. This reduces the overhead of updating the pagetable entry in memory.

from SPUR to show that the miss bit approximation, which updates the reference bit only on cache misses, has the best performance.

8.1 Dirty Bit Alternatives

This section presents four approaches to maintaining dirty bits in a virtual address cache. Section 8.1.1 describes the alternatives and qualitatively analyzes their hardware complexity. Section 8.1.2 uses a simple analytic model to estimate their performance. Section 8.1.3 uses measurements from the SPUR prototype to confirm the results of the analytic model. Finally, in Section 8.1.4 we use measurements from several Sprite systems to show that the benefit of dirty bits decreases with memory size, and suggest that operating systems may someday eliminate them.

8.1.1 Dirty Bit Implementation Trade-offs

Dirty bits are set infrequently: none of the workloads used in Chapter 6 set a dirty bit more often than once every 150,000 references. Since maintaining dirty bits is a small component of total system performance, we are not interested in finding the optimal approach. Rather, our goal is to determine the simplest implementation that yields acceptable performance.

One of the lessons we have learned from RISC processor designs[63] is to implement frequent cases in hardware and trap to software for the infrequent ones. We can apply this lesson to dirty bits. A page's dirty bit must be checked frequently, perhaps as often as every processor write, but only needs to be set infrequently, on the first write to each page. Thus while we must dedicate some hardware to check the bit, we can trap (or, in SPUR nomenclature, *fault*) to software to update the bit. Moving infrequent functions from hardware into software reduces the hardware complexity and may improve performance by reducing the cycle time.

Setting the dirty bit in software is very desirable in shared-memory multiprocessors. Because pagetable entries are shared between processors, updates must either be atomic or controlled by some higher-level synchronization mechanism, such as a semaphore. Performing the pagetable entry updates in software can therefore substantially simplify the memory management unit design. Throughout the rest of the chapter we assume that all pagetable updates are performed under software control.

Since we update the dirty bit using a software fault handler, it is natural to consider combining dirty bit checking with the mechanism that checks protection. After all, both functions cause a fault to software on the first write to a page. To emulate dirty bits with protection, the operating system initially marks writable pages as read-only, then when the first write causes a fault, it sets a software dirty bit and increases the protection level to read-write. This approach requires no special hardware support, and only minor changes to the operating system.

This approach, which we refer to as the *FAULT* alternative, is very promising and is used

in several commercial machines, e.g., the MIPS R2000[22]. However, there is a drawback to using it in systems with virtual address caches. In a virtual address cache like SPUR's, a copy of the protection information is stored with each cache block[2]. Thus there may be several blocks from a particular page in the cache, each with its own copy of the protection level. Changing the protection in the pagetable entry does not affect blocks already brought into the cache, as illustrated in Figure 8.1. Thus, even though the first write to a page causes a fault and makes the page writable, subsequent writes to other resident cache blocks will also fault. Under extreme conditions, these *excess faults* could degrade performance by 50% or more[3]. A reasonable goal is to keep the overhead of maintaining dirty bits below 0.1%.

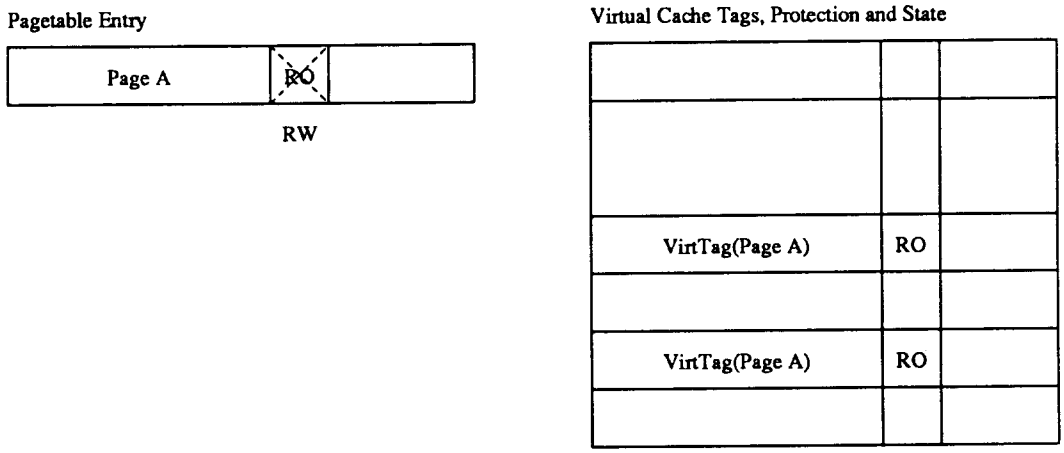


Figure 8.1: Example of Multiple Blocks in the Cache

This figure illustrates the problem with emulating dirty bits with protection in a virtual address cache. Most virtual address caches store a copy of the protection with each cache line, since they only reference the pagetable entry (translation lookaside buffer) on cache misses. Updating the protection in the pagetable entry does not affect these cached copies. Thus even though the page has been made writable, there may be blocks with the old protection value that cause excess faults.

One approach to eliminating excess faults is to flush the page from the cache when the first fault occurs. This guarantees that no blocks from the page remain in the cache with the old protection level. This alternative, called *FLUSH*, incurs less overhead than the FAULT policy if the time to flush the page is less than the expected overhead due to excess

[2]The protection bits are unnecessary if the operating system flushes the cache on each context switch and system call. However, most systems try to minimize cache flushing because it degrades performance.

[3]Consider a program that generates a necessary dirty bit fault once every 100,000 references, with a software handler that requires 1000 references to process the fault. If there are an average of 100 excess faults per necessary fault (128 blocks per page), then each necessary fault has 100,000 references of excess overhead.

faults. On the other hand, if flushing a page from the cache is inefficient, or excess faults are infrequent, then this alternative is slower.

In the design of SPUR, we took another approach to reducing the performance penalty of previously cached blocks. Rather than emulate the dirty bit with protection, the pagetable entry contains an explicit, hardware-defined dirty bit. And just as we cache the page's protection with each cache block, we also cache the page's dirty bit. Note that this bit is distinct from the *block dirty bit* that indicates that a particular cache block has been modified (this difference is illustrated in Figure 8.2). When a block is brought into the cache, both the protection and page dirty bit are copied from the pagetable entry into the cache. Thus if the page has already been modified, the cached page dirty bit will be a one.

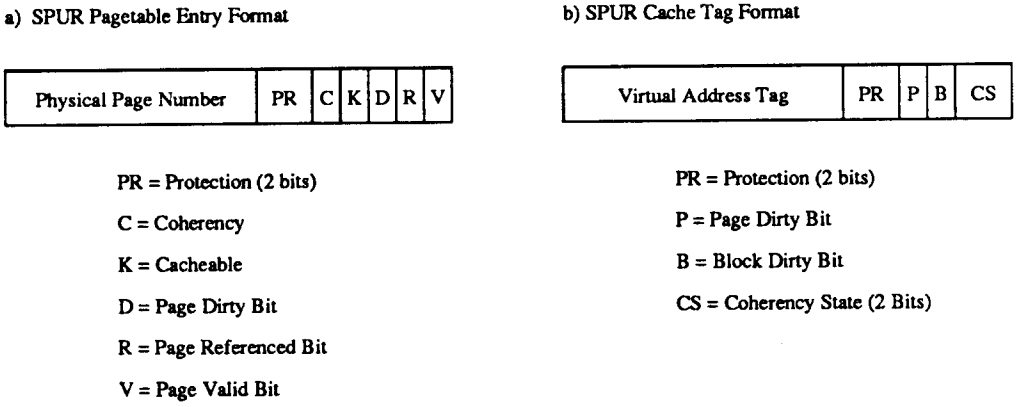


Figure 8.2: SPUR Pagetable and Cache Line Format

This figure illustrates the pagetable entry format and cache line (block frame) format in the SPUR prototype. Note that the cache line contains two dirty bits: a *block dirty bit*, that indicates that the cache block has been modified while in the cache, and a *page dirty bit*, that indicates that the page has been modified. When a block is brought into the cache, the page dirty bit and protection are copied from the pagetable entry into the cache line. Since the pagetable entry may change while a block is in the cache, the cached copies of the page dirty bit and protection may become inconsistent with respect to the pagetable entry.

Each time a processor writes to a cache block, the hardware checks the cached copy of the page dirty bit. The first time a page is written, the cached copy indicates that the page is clean. To verify that this is the first write to a page, the hardware checks the pagetable entry. If the pagetable entry dirty bit indicates that the page is still clean, then this is the first write to the page and the hardware generates a "dirty bit fault" to a software handler. If a write finds that the cached copy of the dirty bit indicates the page is clean, but the pagetable entry is marked dirty, then we have already faulted on another cache block and don't need to fault again. Instead, the hardware merely updates the cached copy

of the page dirty bit, and the write proceeds normally. All subsequent writes to that block find the cached copy of the page dirty bit set, and proceed without delay. In the SPUR implementation, we update the cached copy of the page dirty bit by forcing a cache miss; this leads to the name *dirty bit miss*, which we use for the rest of the chapter.

It is important to emphasize the similarities and differences between the SPUR scheme and that of emulating dirty bits with protection (the FAULT alternative). In both approaches, the "dirty bit information" (dirty bit or protection) is cached with each block on a cache miss, and the first write to a page results in a fault to a software routine that actually modifies the pagetable entry. The key difference is that any subsequent writes to other cache blocks (from the same page) that were brought in while the page was still clean cause faults when emulating with protection, but only dirty bit misses in the SPUR scheme. Since a fault takes at least one order of magnitude longer than a dirty bit miss (as discussed in the next section), the SPUR scheme performs significantly better when there are many writes to previously cached blocks.

Note that while SPUR implements an explicit dirty bit, the same idea could be applied directly to the protection. Instead of immediately faulting to software when the cached copy of the protection indicates an access violation, the hardware first checks the pagetable entry. If the cached copy is out of date, the hardware refreshes it (with a "protection bit miss") and permits the access to proceed. Since the performance of this scheme is identical to what we implemented in SPUR, we will not discuss it separately.

Finally, we also consider a fourth alternative, called WRITE, which is similar to the approach used in the Sun-3 architecture[83]. In this scheme, the hardware checks the pagetable entry dirty bit on the first write to a cache block. There are two cases to consider. First, when a write misses in the cache, the controller must examine the pagetable entry to obtain the physical address, so checking the dirty bit incurs no additional penalty. In the second case, a write hits on a clean cache block, which is indicated by the block dirty bit. In this case, some additional overhead is needed to check the pagetable entry's dirty bit. If the page is dirty, then execution continues, but if it's clean, the hardware generates a fault to a software handler[4]. Since this scheme always checks the pagetable entry before faulting it never generates excess faults.

Table 8.1 summarizes the four alternatives. Both FAULT and FLUSH emulate dirty bits using protection, and therefore require no additional hardware support. The SPUR alternative uses hardware to check the pagetable entry before faulting, thus reducing the performance penalty from excess faults. The WRITE alternative uses hardware to check the pagetable entry on the first write to every cache block, regardless of whether the page was clean or dirty when the block was brought into the cache.

[4]The Sun-3 architecture uses this approach, but implements the pagetable update in hardware.

Name	Description
FAULT	Emulate dirty bits with protection. Writes to previously cached blocks cause excess faults.
FLUSH	Emulate dirty bits with protection. When a fault occurs, flush all blocks in that page from the cache, preventing excess faults.
SPUR	Store a copy of the dirty bit with each cache block. Check the PTE before faulting; if the cached copy is merely out of date, update it with a dirty bit miss.
WRITE	Check the PTE on the first write to each cache block.
MIN	Minimal policy. Includes only overhead intrinsic to all policies.

Table 8.1: Dirty Bit Implementation Alternatives

8.1.2 Analysis

In this section we analyze the design alternatives to determine which has the best performance. Emulating dirty bits with protection makes the least demands upon the hardware, and is therefore the most attractive to implement. But do excess faults present a performance problem? If so, does flushing the page prove an efficient alternative or does the performance gain from SPUR's dirty bit miss mechanism justify the implementation complexity?

We evaluate the different approaches by comparing mean-value models of their overhead, which weight the frequency of events by their duration. We can express the performance overhead of the four alternatives:

$$O(policy) = \text{overhead of dirty bit policy} \tag{8.1}$$

in terms of the following parameters:

N_{ds} Number of necessary dirty bit faults.

$N_{ef} = N_{dm}$ Number of previously cached blocks that cause *excess faults* or *dirty bit misses*.

N_{w-hit} Number of blocks brought into cache by a read that are later modified.

N_{w-miss} Number of blocks brought into cache by a write miss.

t_{ds} Time required to handle a dirty bit fault.

t_{dm} Time required to handle a dirty bit miss.

t_{flush} Time to flush a page from the cache. Assumes 10% of blocks from the page are in cache and are clean.

t_{dc} Time to check the dirty bit when writing to a clean block in the cache.

In the first alternative (FAULT), the hardware faults when it is necessary to set the dirty bit (i.e., on the first write to a page) and also on previously cached blocks.

$$O(FAULT) = (N_{ds} + N_{ef})t_{ds} \tag{8.2}$$

In the SPUR implementation, the fault handler must switch to the kernel stack, read a cache controller status register to determine the type of fault, then decode the instruction to determine the address of the pagetable entry. Because of the high overhead incurred on a fault, roughly 1000 cycles[5], the actual time to update the pagetable entry is a small fraction of the total time. Thus we assume there is no difference in the time to handle necessary and excess faults.

The second alternative (FLUSH) calls for flushing the page from the cache on a fault, preventing excess faults from occurring.

$$O(FLUSH) = N_{ds}(t_{ds} + t_{flush}) \tag{8.3}$$

SPUR's flush mechanism flushes a single cache block regardless of its virtual address tag. Thus flushing a page from the cache requires 128 flush operations, one for each block. The SPUR cache controller does not check the address tag, so it may unnecessarily flush blocks from other pages, substantially increasing the bus traffic and the total overhead. However, a flush operation that checks the address tag could easily be implemented, and we assume this operation exists for a more generic comparison. Even with this operation, flushing a page from the cache will still take approximately 500 cycles (128 blocks to check, two instructions for loop overhead, 90% of blocks at 1 cycle per block, 10% must be flushed at 10 cycles per block). This is approximately half the overhead of an excess fault, not counting the time to reread blocks that are accessed again. Therefore, FAULT is superior to FLUSH if there are at least twice as many necessary faults as excess faults.

The SPUR alternative greatly reduces the overhead of writes to those memory blocks that are already cached. A dirty bit miss takes 25 cycles, on average, compared to 1000 cycles for an excess fault.

$$O(SPUR) = N_{ds}(t_{ds} + t_{dm}) + N_{dm}t_{dm} \tag{8.4}$$

[5]The current implementation of the fault handler has not been tuned. We believe that it can be improved, but doing so will not affect our conclusions.

Parameter	Cycle Count	Description
t_{ds}	1000	Time for handler to set dirty bit
t_{flush}	500	Time to flush page from cache
t_{dm}	25	Time to update cached dirty bit
t_{dc}	5	Time to check pagetable entry dirty bit

Table 8.2: Values for the Model Time Parameters

If a large fraction of the blocks in a page are read before they are written, then this scheme will greatly reduce the overhead to maintain dirty bits.

The last alternative (WRITE) checks the pagetable entry on the first write to a cache block.

$$O(WRITE) = N_{ds}t_{ds} + N_{w-hit}t_{dc} \tag{8.5}$$

We assume that translation is done using SPUR's in-cache translation algorithm, and that the probability the pagetable entry is in the cache is the average across all pagetable entry references. This assumption is pessimistic, but, as we shall see below, does not affect our conclusions. It takes 3 cycles to check the pagetable entry if it is in the cache, plus a weighted miss penalty of about 2 cycles. Thus t_{dc} is approximately 5 cycles. Table 8.2 summarizes the time parameters discussed above.

Finally, we also consider the minimal policy (MIN) for comparison.

$$O(MIN) = N_{ds}t_{ds} \tag{8.6}$$

This alternative includes only the overhead to update the dirty bit in software, and is optimal in the sense that it incurs no additional overhead to check the dirty bit or to handle excess faults.

The overheads of the first three alternatives strongly depend upon the frequency of excess faults relative to necessary faults. If excess faults occur frequently, then the SPUR policy is best because it handles this case in hardware. If they are relatively rare, then the FAULT policy will perform nearly as well as the SPUR alternative, but without requiring any special hardware support. The FLUSH policy falls in between; it will never perform as well as the SPUR policy but also requires no additional hardware and has bounded performance degradation.

To estimate the frequency of excess faults, we construct a simple analytic model. Excess faults only occur when blocks that have been read into the cache while the page is clean are later modified. Thus we can consider only those blocks that will be modified while in the cache. First, we make several assumptions to simplify the model:

(a) Read and write misses to blocks that will be written while in the cache are uniformly

distributed over time with parameter p_w, where p_w is the probability that a block that is modified while in the cache is brought into the cache by a write miss.

(b) Pages are infinitely large (i.e., each page has an infinite number of blocks).

(c) Necessary dirty bit faults only occur on write misses.

With these assumptions, we can model the number of blocks that are read before they are written. The probability that n blocks are in the cache is simply:

$$P\{X = n\} = p_w(1 - p_w)^n \tag{8.7}$$

But this is simply the geometric distribution with parameter p_w. Since an excess fault is caused each time one of these blocks is written, the expected number of excess faults is just the mean of the geometric distribution.

$$\begin{aligned} X &= \text{Number of excess faults per necessary fault} \\ X &\sim \text{Geometric}(p_w) \\ E[X] &= \frac{1 - p_w}{p_w} \end{aligned} \tag{8.8}$$

From our measurements in Chapter 6, we know that the parameter p_w usually ranges from 0.7 to 0.9. For these values, the model predicts that for each necessary fault, we can expect from 0.1 to 0.4 excess faults to occur.

This is a very small number of excess faults per necessary fault, and suggests that they rarely occur. If this result holds true in practice, then we will clearly favor the FAULT policy over either FLUSH or SPUR. Assumptions (b) and (c) both tend to make the model overpredict the true frequency of excess faults. However, assumption (a) is key. If the misses are not uniformly distributed, then the model could have an extremely large error. To validate this model, we must perform additional measurements, which we discuss in the next section.

The performance of the WRITE alternative depends upon the frequency with which blocks are read into the cache before being modified. These events obviously occur much more often than excess faults, but require far fewer cycles to handle. The WRITE policy has lower overhead than FAULT if excess faults occur at least once for every 200 of these events:

$$\begin{aligned} O(WRITE) &\overset{?}{\leq} O(FAULT) \\ N_{ds}t_{ds} + N_{w-hit}t_{dc} &\overset{?}{\leq} (N_{ds} + N_{ef})t_{ds} \\ \frac{N_{w-hit}}{N_{ef}} &\overset{?}{\leq} \frac{t_{ds}}{t_{dc}} \\ \frac{N_{w-hit}}{N_{ef}} &\overset{?}{\leq} 200 \end{aligned}$$

However, while we can estimate whether this relationship holds using back-of-the-envelope calculations, we have no theoretical basis for this comparison. Instead, we leave this evaluation to the next section, where we use measurements from the SPUR prototype.

8.1.3 Dirty Bit Evaluation

In this section we describe an evaluation of the dirty bit implementation alternatives using measurements from the SPUR prototype. We used a more limited workload in this evaluation, consisting of only two synthetic workloads. The first is a multiprogramming script (called *WORKLOAD1*) designed to reflect a moderately heavy load for a CAD tool developer. This script includes the compilation of several C program modules plus the link and debug of a 12000 line CAD tool (espresso). The same CAD tool runs in the background optimizing a large PLA. Other edit, compile, and miscellaneous commands manipulate files and directories. In addition, two performance monitor programs periodically report status of the virtual memory system and CPU performance[6]. The second workload (called *SLC*), uses the SPUR Common Lisp[95] system and the SPUR Lisp compiler to compile a set of benchmark programs. These two workloads are representative of the types of applications originally intended to run on SPUR.

Table 8.3 summarizes the event frequencies for these two workloads, measured on the SPUR prototype. It is immediately clear from this table that *excess faults occur very infrequently*. At 6 and 8 megabytes of main memory, both workloads cause less than 8% as many excess faults as necessary faults (i.e., for each necessary fault, they generate less than 0.08 necessary faults). At 5 megabytes, the fraction climbs to 16% for WORKLOAD1 and 10% for SLC. However, in all cases excess faults are infrequent, suggesting that pages that will be modified are modified quickly. The ratio of N_{w-hit} to N_{w-miss} supports this hypothesis: roughly one-fifth (from 16% to 24%) of the modified cache blocks are read before they are written.

However, further investigation uncovered another reason for the low percentage of excess faults. Like UNIX, the Sprite operating system initializes newly allocated stack and heap pages to zero. The kernel maps the initialized page into the process's address space with the dirty bit turned off. But since programs rarely want to read stack and heap pages before they are written (i.e., read a zero), the first operation to these pages is almost always a write, resulting in a dirty bit fault. If we exclude these zero-fill pages from the necessary dirty bit faults, the fraction of excess faults increases, ranging from 15% to 34%. While still a small percentage, this range is closer to our model's prediction.

Table 8.4 summarizes the overhead for the implementation alternatives. Because they are not intrinsic, we exclude the zero-fill pages from the calculations[7] (i.e., the difference

[6]This workload lacks any window activity, a major deficiency for a workstation environment. Unfortunately, no window system currently runs on SPUR, so it is not possible to include this behavior.

[7]Note that Sprite will always write a zero-filled page to swap the first time it is replaced, even if the program has not modified it.

Workload	Memory Size (MB)	N_{ds}	N_{zfod}	N_{ef} N_{dm}	N_{w-hit} (million)	N_{w-miss} (million)	$t_{elapsed}$ (seconds)
SLC	5	2349	905	237	1.27	7.38	948
	6	1838	905	143	0.839	5.11	502
	8	1661	905	120	0.612	3.68	341
WORKLOAD1	5	9860	5286	1534	6.15	34.0	3016
	6	7843	5181	456	4.92	20.4	2535
	8	7471	5182	364	4.10	17.3	2555

Table 8.3: Event Counts Measured on the SPUR Prototype

This table summarizes the measurement results for the two synthetic workloads for each of 3 memory sizes. The first two metrics are N_{ds}, the number of dirty bit faults, and N_{zfod}, the number of zero-fill page faults. The number of excess faults, N_{ef}, and the number of dirty bit misses, N_{dm}, are two names for the same value. N_{w-hit} is the number of times blocks are read into the cache before being written, and N_{w-miss} is the number of times blocks are brought into the cache by a write miss.

Workload	Memory Size (megabytes)	MIN	FAULT	FLUSH	SPUR	WRITE
		Fraction of total cycles (relative to MIN)				
SLC	5	0.00031 (1.00)	0.00036 (1.16)	0.00046 (1.50)	0.00032 (1.03)	0.00167 (5.41)
	6	0.00031 (1.00)	0.00036 (1.15)	0.00046 (1.50)	0.00032 (1.03)	0.00169 (5.50)
	8	0.00035 (1.00)	0.00041 (1.16)	0.00052 (1.50)	0.00036 (1.03)	0.00176 (5.05)
WORKLOAD1	5	0.00022 (1.00)	0.00029 (1.34)	0.00033 (1.50)	0.00023 (1.03)	0.00169 (7.72)
	6	0.00015 (1.00)	0.00018 (1.17)	0.00023 (1.50)	0.00016 (1.03)	0.00155 (10.2)
	8	0.00013 (1.00)	0.00015 (1.16)	0.00019 (1.50)	0.00013 (1.03)	0.00128 (9.95)

Table 8.4: Overhead of Dirty Bit Alternatives (Excluding Zero-Fills)

$N_{ds} - N_{zfod}$ is substituted for N_{ds} in the models). Because excess faults occur infrequently, we expect the FAULT policy to perform well. Flushing the page from the cache, the FLUSH policy, increases the overhead by 26% over FAULT on average. Only for WORKLOAD1 at 5 megabytes does FLUSH start to come close, dropping to only 12% worse. The SPUR scheme has the best performance, requiring only 3% more than the minimum (MIN). But since excess faults are rare, SPUR's overhead is only 16% less than FAULT's; this difference amounts to less than 0.01% of total system performance. In addition, the SPUR scheme requires one additional state bit per cache block[8], plus an additional 14 product terms in the controller's main PLA (193 vs 207, or 7%). We believe the minor performance improvement does not justify the increase in chip and board complexity.

The WRITE policy performs worst of all by a large margin. The ratio N_{w-hit}/N_{ef} exceeds 4000 for all workload and memory size combinations; this is a factor of 20 larger than the break-even point discussed in the last section. Even if we reduce the time to check the pagetable entry dirty bit to only 1 cycle, this alternative still has the worst performance by a large margin. Since this policy has higher overhead despite special hardware support, it is clearly inferior to the FAULT policy.

To summarize the results of this section, it is clear that we can efficiently emulate dirty bits using protection even when using a large virtual address cache. Based on our synthetic benchmarks and measured events on the SPUR hardware, the frequency of excess faults ranges from 16% to 34% of the frequency of necessary faults. Additional hardware support can improve dirty bit performance by at most 34%, which amounts to much less than 1% of overall system performance. Simply tuning the fault handler would probably achieve a larger improvement. This is good news to hardware designers because it further simplifies the logic they must implement. Eliminating the dirty bit support in the SPUR prototype reduces the number of product terms in the cache controller Sequencer PLA by 7%.

8.1.4 Benefits of Dirty Bits

In the first half of this section, we looked at the cost and performance impact of different implementations of dirty bits. In the remainder of the section, we take a step back and look at what we gain from implementing dirty bits. Although we have shown that dirty bits require no special hardware support, they do add some complexity to the operating system.

Dirty bits improve system performance by eliminating unnecessary page-outs; clean pages need not be written back to secondary storage when displaced from main memory. Dirty bits only help for pages which can be modified; since most systems disallow direct updates of code, dirty bits provide no information for code pages. During times of heavy paging, pages do not stay in memory long and thus are unlikely to be modified. Under these conditions dirty bits can potentially reduce the page-out traffic. However, memory prices have dropped by a factor of 100 over the last 10 years [60] prompting a rapid increase in main memory sizes. Workstations are now commonly sold with at least 16-24 megabytes of

[8]The generalized version, using the protection field, eliminates the need for an extra bit.

Hostname	Memory Size (MB)	Time (hours)	Number of Page-Ins	Potentially Modified Pages	Not Modified Pages	Percent Not Modified	Percent Add'l Paging
mace	8	70	15203	2681	488	18%	2.8%
sloth	8	37	10566	2146	129	6%	1.0%
mace	8	46	48722	5198	814	16%	1.4%
sage	12	45	5246	544	14	3%	0.2%
fenugreek	12	36	8556	1154	58	5%	0.6%
murder	16	119	23302	12944	895	7%	2.5%

Table 8.5: Page-Out Results from Sprite Development Machines

This table presents measurements from several Sun-3 workstations running the Sprite operating system. The paging statistics presented here are the number of pages actually read from the file server, the number of pages selected for replacement that *could* have been modified, and the number of pages selected for replacement that could have been modified, but were not.

memory. With the relatively low price of memory, most users will increase their memory size rather than sustain consistently high paging rates. Many computing environments also provide compute servers in addition to workstations; jobs which page heavily on a workstation are usually moved to these larger machines.

In conjunction with the increase in memory size, many workstations have gone to large page sizes: the Sun-3's pages are 8 kilobytes, the MIPS R2000's and SPUR's are 4 kilobytes. Both factors, large memories and large pages, suggest that most modifiable pages will be modified while in memory. Thus dirty bits may be of marginal utility, and perhaps could be excluded from future operating systems. To examine this hypothesis, we looked at the page-out performance of our Sun-3's running Sprite. The Sprite developers use these systems to enhance and maintain the Sprite operating system, as well as other tasks such as reading mail, and writing papers and dissertations. The workload on these machines is similar to many other software development environments.

Even though some of the systems have only 8 megabytes of memory, the paging rates are still relatively low. There is also a certain amount of self-scheduling; users tend to run programs with very large memory demands on the systems with more physical memory.

Table 8.5 displays the measurements from the development machines. The second to last column holds the main result: with 8 megabytes of memory at least 80% of all modifiable pages are modified. With 12 megabytes or more, the fraction increases to at least 90%. Also, as shown in the last column, the total number of additional pages that would be written out without dirty bits only increases the total number of paging I/Os by at most

3%. Since the paging rate is already low, a 3% increase will have a negligible impact on the total system performance.

These results support our hypothesis, indicating that dirty bits provide only negligible performance improvement for this class of workload. While these results may not apply to all applications, it is clear that for an important class of workloads dirty bits provide little benefit, which will decline further with increasing memory size.

8.2 Reference Bits

In this section, we examine the trade-offs in reference bit implementations. We begin by presenting three alternative approaches and qualitatively discussing their relative merits. Then, in Section 8.2.2 we evaluate their performance using measurements from the SPUR prototype.

8.2.1 Reference Bit Policies

Reference bits are used to maintain a pseudo-LRU ordering of resident pages. A *page daemon* periodically clears the reference bits and reclaims unreferenced pages. As with dirty bits, the approach generally taken in systems with translation lookaside buffers is not directly applicable to systems with virtual address caches. In traditional implementations, the reference bits are cached in the translation lookaside buffer and checked on each processor reference. However, in a system like SPUR, it is impractical to check the bits this frequently. Instead, SPUR only checks the reference bit on cache misses. Since the hardware already must access the pagetable entry on a cache miss, there is no additional penalty to check the reference bit. As with dirty bits, SPUR generates a fault to a software handler when the bit must be set. While SPUR supports an explicit reference bit, we could just as easily emulate them with the valid bit, as done in BSD Unix on the VAX architecture[9].

Checking the reference bit on cache misses reduces the overhead, but it does not provide exactly the same functionality. When the page daemon, or allocate procedure, clears the reference bit, it does not affect blocks from that page that are already in the cache. Thus the processor can continue to reference those blocks without setting the reference bit. Under this policy, which we call the *MISS bit approximation*, or simply the *MISS* policy, the page daemon may incorrectly replace pages that have actually been recently referenced, but have not recently caused a cache miss.

We can eliminate these incorrect replacements if the page daemon flushes the page from the cache when it clears its reference bit. This guarantees that the next reference to the page will cause a cache miss, setting the reference bit. Under this policy, called *true reference bits*, or simply the *REF* policy, the reference bits are accurately maintained and should result in fewer page faults than the MISS policy. However, the REF policy does not come for free. Flushing a page from the cache is an expensive operation relative to the cost of clearing the reference bit. This is especially true in a multiprocessor, which must flush the

page from all the caches. Not only does the flush take a long time, but it disrupts the cache, forcing additional cache misses to refetch some of the blocks.

For small caches, the MISS policy is probably a good approximation to true reference bits. When the cache miss rate is high then the average residency of a block is short, and the reference bit for an active page will be set fairly soon after it is cleared. But as caches increase in size, we expect the approximation to become worse. Consider a cache of infinite capacity. Once a block is brought into the cache it never leaves without being explicitly flushed. Thus the MISS policy never sets the reference bit once the entire page is resident in the cache. At this extreme, the MISS bit approximation provides no benefit; eliminating reference bits all together, the *NOREF* policy, would be superior since it eliminates the overhead of checking, setting, and clearing the bits.

The NOREF policy should also be superior with large memory sizes. It has been observed that large systems spend lots of time searching for unreferenced pages[59]. With large enough main memories, the overhead to maintain pseudo-LRU ordering may exceed the overhead of additional faults incurred by not maintaining reference bits.

We consider a very simple NOREF policy, primarily to minimize changes to the Sprite virtual memory system[61]. Under this policy, the basic replacement algorithm is unchanged, however the machine dependent routine that reads the hardware reference bit always returns false. Conversely, the routine that clears the hardware reference bit has no effect, leaving the hardware bit always set (thus preventing reference faults). While we consider this policy to be reasonable, we believe there may be better replacement algorithms that do not maintain reference bits. Nonetheless, if we get acceptable results under this policy, then it supports the proposition that we can eliminate reference bits.

8.2.2 Reference Bit Evaluation

To evaluate these three policies, we modified Sprite to use each of the them under control of run-time flags. We ran our synthetic workloads on the SPUR prototype, with 5, 6, and 8 megabytes of memory. We ran five repetitions of each data point, using a randomized experiment design to minimize bias. The main results are summarized in Table 8.6.

Running WORKLOAD1 at 8 megabytes of main memory generates only a small amount of paging activity, so the overhead of maintaining reference information exceeds the benefits. The REF policy requires 3% fewer page-ins than the MISS policy, but takes 6% longer to execute due to the flush overhead. The NOREF policy generates 5% more page-ins than MISS, but because it spends no time maintaining reference bits runs 2% faster.

As we would expect, the reference bits provide more benefit as memory becomes more scarce. At both 5 and 6 megabytes, the NOREF policy generates significantly more page-in traffic, 134% and 143% respectively. Note, however, that at 6 megabytes, the performance degradation is still only 1%. Finally, at 5 megabytes paging becomes heavy and the REF policy results in 7% fewer page-ins than MISS. Nonetheless, MISS still requires 5% less time to execute because of its lower overhead.

Workload	Memory Size (megabytes)	Policy	Page-Ins		Elapsed Time (seconds)	
SLC	5	MISS	4647	(100%)	948	(100%)
		REF	4738	(102%)	1020	(108%)
		NOREF	8230	(177%)	1341	(141%)
	6	MISS	1833	(100%)	502	(100%)
		REF	1866	(102%)	534	(106%)
		NOREF	3465	(189%)	703	(140%)
	8	MISS	1056	(100%)	341	(100%)
		REF	1062	(101%)	342	(101%)
		NOREF	1512	(143%)	382	(112%)
WORKLOAD1	5	MISS	11959	(100%)	3016	(100%)
		REF	11119	(93%)	3153	(105%)
		NOREF	16045	(134%)	3214	(107%)
	6	MISS	3556	(100%)	2535	(100%)
		REF	3617	(102%)	2677	(106%)
		NOREF	5073	(143%)	2555	(101%)
	8	MISS	1837	(100%)	2555	(100%)
		REF	1790	(97%)	2701	(106%)
		NOREF	1926	(105%)	2505	(98%)

Table 8.6: Reference Bit Measurement Results

This table uses measurements from the SPUR prototype to compare the three page replacement strategies. The MISS policy approximates reference bits by only checking them on cache misses; the REF policy maintains accurate reference bits by flushing the page from the cache when clearing the reference bit; the NOREF policy does not maintain even an approximate reference bit.

The SLC workload displays much more uniform behavior. The MISS policy always results in the fewest page-ins and the shortest elapsed time. For this workload, the NOREF policy never comes closer than 12% slower; for the two smaller memory sizes, it is 40% slower. The REF policy has comparable performance to MISS at 8 megabytes, when there is little paging. But at smaller memory sizes, despite selective cache flushing on reference bit clears, it still generates more page-ins than MISS; consequently, REF has a longer execution time than MISS.

In summary, implementing true reference bits may reduce the number of page-ins at low memory sizes, but the CPU overhead required always exceeds the benefit of the lower fault rate. The MISS bit approximation has the best overall performance, generating significantly fewer page-ins than the NOREF policy but without the high overhead of true reference bits. For WORKLOAD1, however, eliminating reference bits altogether (NOREF) has the best performance at 8 megabytes, and only slightly worse performance at 6 megabytes. Clearly these results do not apply to all systems, but certainly for this workload reference bits provide at best a small increase in performance. As memory sizes increase, this benefit will tend to decrease and may eventually become a hindrance.

8.3 Summary

In this chapter, we have examined alternative implementations of reference and dirty bits for virtual address caches. Virtual address caches provide faster access times than physical address caches, but make it more difficult to maintain bookkeeping information because we no longer access the translation lookaside buffer on every reference.

We have shown that simpler is better: we can efficiently emulate dirty bits with standard protection mechanisms. The excess faults that occur when multiple blocks from a clean page are brought into the cache and then modified account for only 19% of all dirty bit faults, on average. The small performance penalty from these additional faults does not justify the increased hardware complexity of SPUR's dirty bit miss mechanism or the Sun-3's first-write mechanism. No special hardware is necessary to efficiently support dirty bits.

Approximating reference bits by checking only on cache misses can result in more page faults at smaller memory sizes. However, the overhead of maintaining true reference bits, by flushing a page when clearing the reference bit, far exceeds the benefit of a lower fault rate.

We also examined the possibility of eliminating reference and dirty bits entirely. In measurements on machines used for software development, more than 80% of all writable pages are dirty when replaced; eliminating dirty bits would have increased the total paging activity by at most 3%. Similarly, for some workloads maintaining reference bits, even with the miss bit approximation, may become a liability at larger memory sizes. At 8 megabytes, not a large memory for today's workstations, the overhead to maintain reference bits exceeds the benefits for one of the two workloads presented. These results strongly suggest that the benefits of reference and dirty bits decline as memory size increases, and may eventually

degrade rather than improve performance.

Chapter 9

Conclusion

In this dissertation, we study in-cache address translation, which combines the functions of the traditional translation lookaside buffer with a virtual address cache. Rather than dedicating hardware resources specifically to hold pagetable entries, in-cache translation lets the pagetable share the regular cache with instructions and data. By combining the two mechanisms we simplify the memory system design, and potentially reduce the cycle time. In addition, by eliminating the translation lookaside buffer, we simplify the translation consistency problem in multiprocessors: the operating system can use the regular data cache coherency protocol to maintain a consistent view of the pagetable.

We employed an experimental approach to this research, going beyond the usual design and analysis steps to build a prototype implementation. Measurements of real programs on the prototype validated many of the key simulation results. When unexpected behavior arose, we formulated new hypotheses and tested them with further experiments.

We have extensively studied the performance of in-cache translation using trace-driven simulation. We used 14 address traces to determine when in-cache translation performs better than traditional translation lookaside buffers. We compared its performance to 11 translation lookaside buffers that characterize the range of commercial implementations. Only 2 of the largest buffers have better performance for the SPUR cache configuration (128 kilobytes, direct-mapped, and 32-byte blocks), and these are no more than 1% better. In-cache translation performs more than 5% better than the worst of the buffers.

While this performance improvement is small, these figures assume a constant cache access time. However, virtual address caches often have faster access times than caches with translation lookaside buffers, because they only require translation on cache misses. When we include this implementation-dependent factor, in-cache translation has the potential to improve performance by as much as 50%.

We tested the sensitivity of these simulation results to the SPUR cache parameters. We found that in-cache translation has superior performance when the cache contains 64 kilobytes or more (assuming 32-byte blocks, direct-mapped). For smaller caches, the interference between pagetable entries and the processor's instructions and data degrades

the performance of in-cache translation, so it falls below that of typical translation lookaside buffers. However, modern memory technologies make large caches both feasible and attractive, and this limitation should not be a factor in modern and future machines.

We also found that the effective access time is sensitive to the time in-cache translation requires to look for a pagetable entry in the cache. For the SPUR cache configuration, each cycle adds 0.013 cycles per reference to the effective access time or approximately 1%. Thus we must clearly implement this operation in hardware. Conversely, because we usually find the pagetable entry, we could trap to software when its not in the cache without seriously degrading performance. This is a promising area for further research.

We implemented in-cache translation as a central feature of the SPUR prototype; a five-processor system convincingly demonstrates the feasibility of this new mechanism. Because of the constraints of the academic environment, the prototype does not run as fast as many current commercial machines. However, we described a set of optimizations that could improve the memory system performance in a more aggressive implementation.

The SPUR prototype allowed us to validate the simulation results, by measuring the performance of in-cache translation for a large set of important programs. We proposed a model for the effective access time using the event counters imbedded in the SPUR cache controller chip. We demonstrated that our original model was accurate in user mode, but inaccurate in kernel mode. We then revised the model to account for two memory system features that were used unexpectedly often: non-cacheable pages and cache controller register accesses. Incorporating both factors reduced the model's average error to less than 1%.

With the revised model, we showed that in-cache translation consumes less than 4% of all cycles, even with a conservative implementation. This result compares favorably with published results for two translation lookaside buffers: the VAX 8800 spends 4.6% of all cycles in translation, while the VAX 11/780 spends 6%. While the difference in cycle count is small, coupling this with the decrease in cycle time permitted by the virtual address cache leads to a large performance improvement.

Both simulations and measurements indicated a sensitivity of in-cache's performance to the pagetable placement. In addition, the pagetable organization, an array, inefficiently uses memory for sparsely populated address spaces. We showed that replacing the data structure with a hashtable solves these problems. This variation, called inverted in-cache translation, performs better than the original scheme in many cases, and no worse than 1% for any of the traces.

Finally, we examined alternative ways to support reference and dirty bits in a virtual address cache. We used an analytic model and measurements from the SPUR prototype to compare the performance of four dirty bit implementation alternatives. We showed that the novel approach implemented in SPUR has the best performance, but that emulating dirty bits with protection is not much worse and requires no additional hardware. Also, the *miss bit approximation* to reference bits, where the bit is only checked on cache misses, performs better than true reference bits, which require a partial cache flush when the bit is

cleared. Both the reference and dirty bit results apply to all virtual address cache designs, rather than just in-cache translation.

In summary, in-cache translation is a new approach to virtual address translation, which combines the functions of the translation lookaside buffer with the virtual address cache. By integrating the two mechanisms we improve performance by reducing the cycle time and the total number of cache and translation misses. In addition, by eliminating a separate translation lookaside buffer we simplifying the memory system design. Our results convincingly show that in-cache translation is an attractive alternative to translation lookaside buffers over a large region of the design space.

Appendix A

Trace-Stitching

In this appendix, we present the results of an experiment that tests the accuracy of *trace-stitching*, a technique proposed by Agarwal for constructing long traces from shorter samples[4]. Although commonly used, no one has analyzed how accurately these composite traces predict the true miss ratio. We present data showing that trace-stitching is more accurate than the standard alternative of averaging the independent samples, particularly for large caches. However, the relative error still exceeds 20% for a 256-kilobyte cache, on average. We also show that the similarity ratio ρ, a metric Agarwal proposed as a indicator of the accuracy of the approximation, is a poor predictor of the relative error.

In the first section, we describe the trace-stitching approximation. In Section A.2 we present experimental results comparing the trace-stitching approximation to the true miss ratio and the alternative approximation. Finally, in Section A.3 we test for a correlation between the similarity ratio and the relative error.

A.1 Trace-Stitching

In trace-driven simulation the cache is initially empty, a condition that rarely occurs in most real systems. The *cold-start behavior* or *start-up distortion* that results from an empty cache, discussed in Chapter 4, causes a simulated miss ratio higher than the true miss ratio. If a trace is sufficiently long, then cold-start misses do not significantly bias the final result. Alternatively, we can use the initial portion of a trace to warm-up the cache, computing the *warm-start miss ratio* from the remainder of the trace.

But for short traces, such as the 400,000 reference samples gathered using the ATUM technique[4], cold-start behavior dominates the miss ratio. Averaging the miss ratio of the independent samples tends to over-predict the true miss ratio, defined as the steady-state miss ratio for the underlying workload. This is particularly true for large caches, since they require more references to warm-up.

Agarwal proposes a technique called *trace-stitching* to reduce the cold-start effects for

short traces[2]. Trace-stitching calls for concatenating several short samples of a particular workload to form a composite, or stitched, trace. If the samples are taken close together in time, then they will share some of the same addresses and reduce the start-up distortion of the subsequent samples.

Agarwal shows that the miss ratios of these stitched traces is lower than the average miss ratio when the samples are simulated separately. He argues that the composite traces more closely approximate the true miss ratios. Unfortunately, Agarwal presents no data to support this claim. Therefore, although our intuition suggests that stitched traces probably are better predictors of the true miss ratio, we don't know *just how well* they predict.

A.2 Accuracy of Trace-Stitching Approximation

We conducted a simple experiment to test the accuracy of the trace-stitching approximation. Using two of the 5 million reference traces described in Chapter 4, *Prod5M* and *Slc2nd5M* we constructed a set of stitched traces and compared their miss ratios to the miss ratios of the full traces.

We broke each of the long traces into 400,000 reference samples, $T_1, T_2, T_3, T_4, \ldots$. Then we created all traces possible by skipping N samples, for N ranging from 1 to 4. In other words, for N equal to 1, we concatenated every other sample, producing two traces: *trace1a* consisting of samples $T_1, T_3, T_5, \ldots$, and *trace1b* consisting of samples $T_2, T_4, T_6, \ldots$. Thus for stride N, there are $N + 1$ composite traces from each base trace.

Using these stitched traces, we simulated direct-mapped caches with 32-byte blocks and capacities ranging from 16 kilobytes to 256 kilobytes. We also simulated the full traces to obtain accurate estimates of the *true* miss ratios, and separately simulated the independent samples. We average the miss ratio for all the traces with the same base trace and same stride.

Figure A.1 plots the results of the simulations. As expected, the average miss ratio of the independent samples is an accurate approximation for small caches, but becomes much worse as the caches become larger. On average, the stitched traces predict the true miss ratio better than the independent samples; however, the prediction still degrades for larger caches. For a 256-kilobyte cache, the average of the independent samples over-predicts the true miss ratio by 50% for *Prod5M*, and 36% for *Slc2nd5M*. For a 64-kilobyte cache, the error drops to 4% and 7% respectively.

The traces derived from *Prod5M* show a clear decrease in accuracy as the interval between samples increases. This is not surprising, since the longer the stride, the more likely the working set has changed substantially. The traces derived from *Slc2nd5M* are more closely clustered and less orderly: the average of the traces with stride 4 most closely approximates the true miss ratio for large caches. The difference in the working sets appears to explain the different behavior as the stride increases. Adjacent samples of *Prod5M* share an average of 75,000 addresses, while the adjacent samples from *Slc2nd5M* share only 30,000

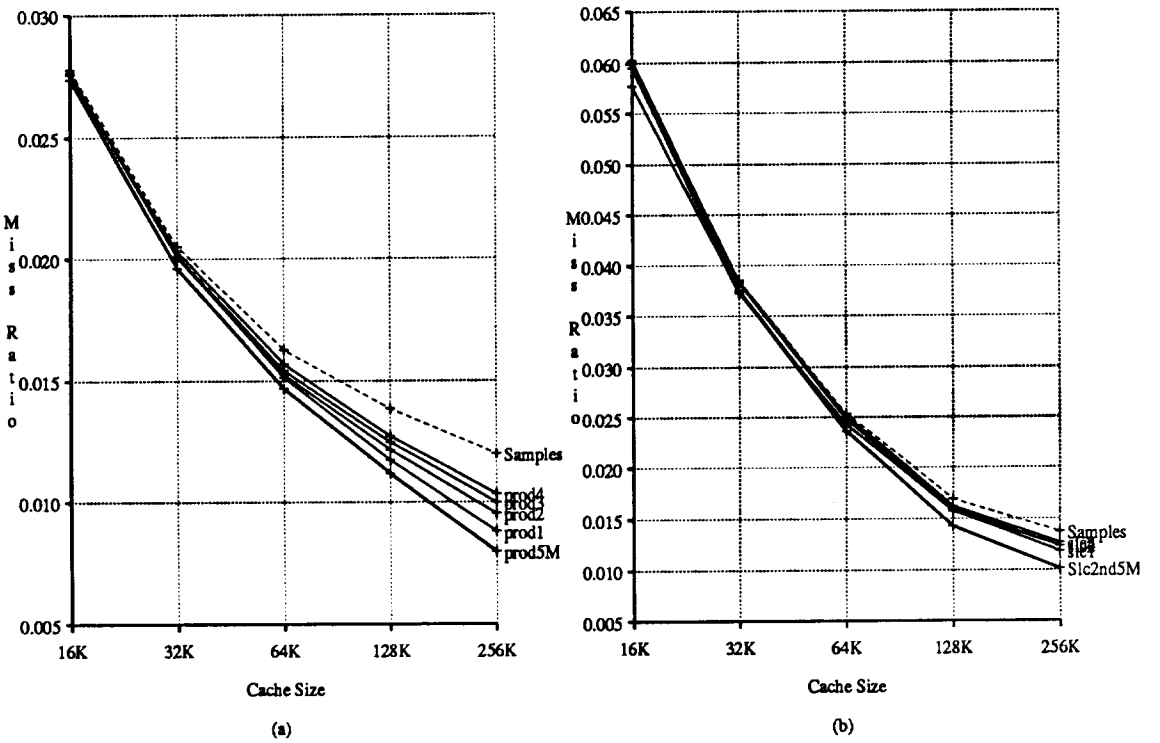


Figure A.1: Comparison of Composite Traces to Full Trace

This figure plots the miss ratios for the stitched traces against the base traces and the alternative approximation. Graph (a) plots the results for the *Prod5M* traces, and graph (b) plots the results for the *Slc2nd5M* traces. The dashed line in each graph is the average of the independent samples; the bold line is the miss ratio of the base trace. The cache is direct-mapped with 32-byte blocks.

addresses, on average. Thus the *Prod5M* trace has "more to lose" from a long stride then the *Slc2nd5M* trace.

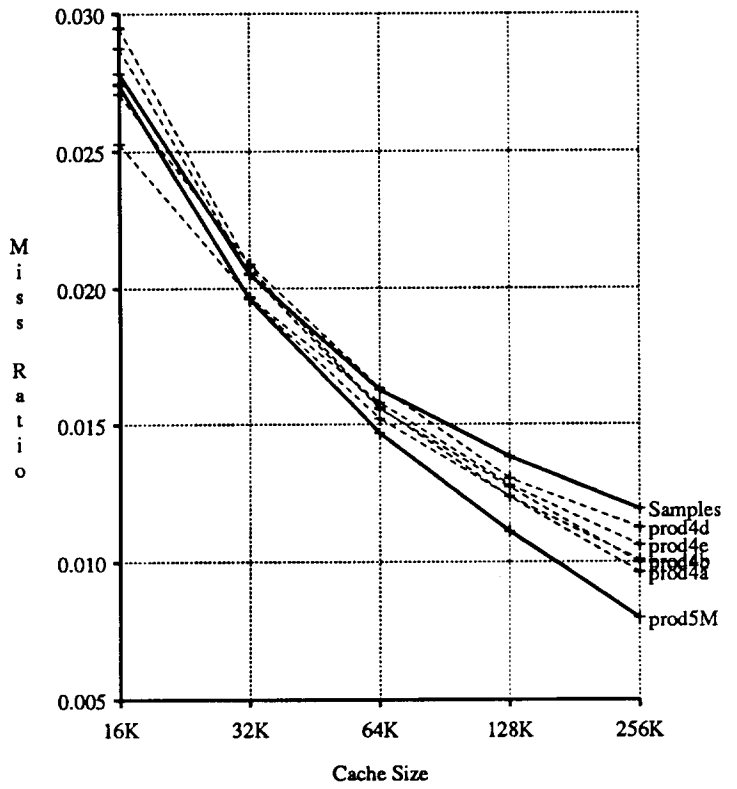


Figure A.2: Variance of Composite Traces with Stride 4

This figure shows the variance of the miss ratios for the *Prod5M* composite traces with stride 4. For large caches, the stitched traces are always better than the independent samples, and always over-predict the true miss ratio. However, for a 16-kilobyte cache the miss ratio of two of the composite traces under-predicts the true miss ratio. The cache is direct-mapped with 32-byte blocks.

As the stride increases, the variance in the miss ratio of the traces with the same stride also increases. Figure A.2 plots all 5 traces derived from *Prod5M* with stride 4 against the true miss ratio and the average miss ratio of the samples. While some of the traces are good predictors of the true miss ratio, others are not. Most importantly, the miss ratios of the composite traces are not necessarily an upper bound on the true miss ratio. This is possible if the samples selected for a particular composite trace come from well-behaved sections of the trace. 14 of the 28 composite traces underpredict the true miss ratio for at least one cache size. 24% of all the simulation combinations (composite trace and cache size)

underpredict the true miss ratio by some amount, and 17% of the combinations underpredict it by more than 5%. Clearly, stitched traces do not always accurately predict the true miss ratio.

A.3 Similarity Ratio

Agarwal proposes that only *similar* samples should be concatenated, to reduce the error of the approximation. He defines two samples as similar if the change in the working sets of the two trace samples is less than the change in the working set within any one trace sample. He measures this with the similarity ratio ρ; if the similarity ratio of two samples is greater than one, then the two traces are similar. The metric ρ is defined as:

$$\rho = \frac{2cs(T_1, T_2)}{as(T_1) + as(T_2)} \tag{A.1}$$

where the *cosimilarity* $cs(T_1, T_2)$ is simply the number of unique references the two samples have in common, and the *auto-similarity* $as(T_1)$ is just the co-similarity of the two halves of sample T_1.

We extend this metric by defining $\bar{\rho}$ as the average of the ρ's for the adjacent samples in a stitched trace. It follows from Agarwal's argument, that if $\bar{\rho}$ is large then the trace should be a good predictor of the true miss ratio. However, our simulations show only a weak correlation between $\bar{\rho}$ and the relative error. Figure A.3 plots the relative error for each simulation against the value of $\bar{\rho}$ for the corresponding stitched trace. Visual inspection shows that the data is not strongly correlated. Using the least squares method, we fit the linear function:

$$\text{relative error} = -0.014\bar{\rho} + 0.15 \tag{A.2}$$

This shows that an inverse relationship exists, as Agarwal predicts, but the magnitude of the slope is very small. In addition, the error in the fit is large: the t-value is only 1.075, indicating a 28% chance that the data is actually random. The 95% confidence interval for the slope is (-0.038, 0.011), indicating that it might even be positive. These results clearly show that $\bar{\rho}$, and hence ρ, is a poor predictor of how well a particular stitched trace approximates the true miss ratio.

A.4 Summary

These results show that stitched traces generally approximate the steady-state miss ratio more accurately than separate simulations of the independent samples. The relative error from trace-stitching is lower in over 75% of the simulations, and within 10% for 90% of the cases. In addition, as the cache size increases, trace-stitching becomes a better approximation: for caches of 128 kilobytes and larger, trace-stitching is more accurate for over 90% of the simulations. Nonetheless, the relative error of trace-stitching is still quite high for large

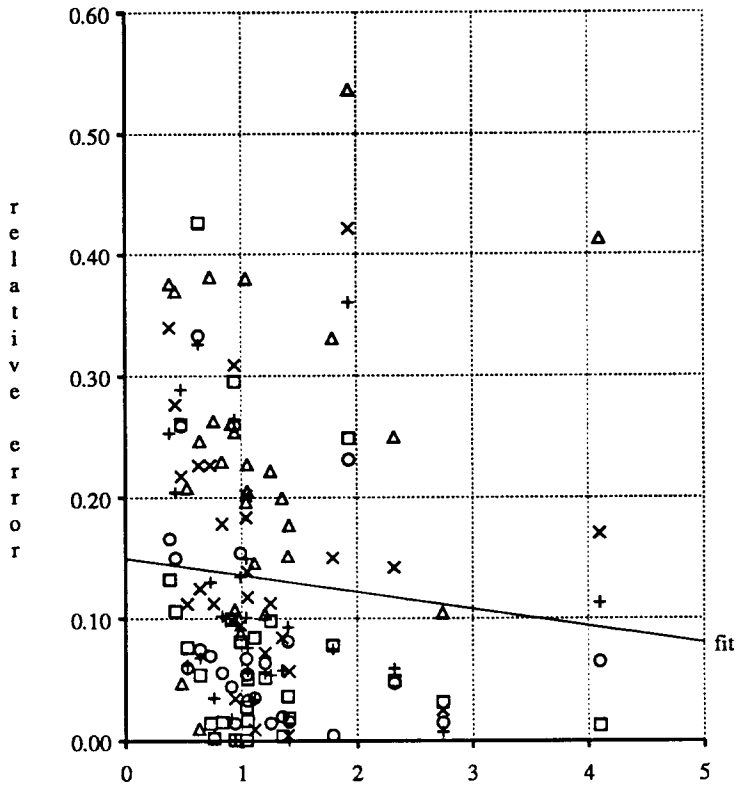


Figure A.3: Scatter plot of $\bar{\rho}$ vs. Relative Error

This scatter plot shows that there is not a strong correlation between $\bar{\rho}$ and the relative error.

caches. The average relative error over all stitched traces is over 20% for a 256-kilobyte cache. Even for those traces that skip only a single sample, the error averages 10% for the *Prod5M* traces and 18% for the *Slc2nd5M* traces. In addition, the similarity ratio ρ does not help predict which stitched traces have the lowest error.

In summary, trace-stitching appears to be better than the obvious alternative of simply averaging the independent samples, and we employ it in Chapter 4. However, it is important to remember that it is only an approximation to the true miss ratio, and that substantial error occurs for large caches. The quality of the approximation will obviously improve as the sample size increases. Determining the trade-off between sample size, sample spacing, and relative error is an interesting area for further research.

Bibliography

[1] Advanced Micro Devices. *Am29000 User's Manual*, 1987.

[2] Anant Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* PhD thesis, Computer System Laboratory, Stanford Univ., May 1987. Available as Technical Report CSL-TR-87-332.

[3] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–432, November 1988.

[4] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 119–127, June 1986. Tokyo, Japan.

[5] C. A. Alexander, W. M. Keshlear, and Faye Briggs. Translation buffer performance in a UNIX environment. *Computer Architecture News*, 13(5), December 1985.

[6] D. P. Anderson and D. Ferrari. The DASH project. In *Proceedings of the ACM SIGOPS Workshop on Distributed Systems*, September 1985. Amsterdam.

[7] Anon, et al. A measure of transaction processing power. *Datamation*, 31(7), April 1985.

[8] Ozalp Babaoglu. *Virtual Storage Management in the Absence of Reference Bits.* PhD thesis, Computer Science Division (EECS), Univ. of California at Berkeley, November 1981. Available as ERL Memo. UCB/ERL M81/92.

[9] Ozalp Babaoglu and William Joy. Converting a swap-based system to do paging in an architecture lacking reference bits. *Operating Systems Review*, 15(5):78–86, December 1981.

[10] C. G. Bell, A. Newell, M. Reich, and D. Siewiorek. The IBM system/360, system/370, 3030, 4300: A series of planned machines that span a wide performance range. In D. Siewiorek, C. Gordon Bell, and Allen Newell, editors, *Computer Structures: Principles and Examples*, pages 856–892. McGraw-Hill, 1982.

[11] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory: Concepts and design. *Communications of the ACM*, 15(5), May 1972.

[12] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron. Translation lookaside buffer consistency: A software approach. In *Proceedings of the Third Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 113–122, April 1989. Boston, MA.

[13] Paul Borrill and John Theus. An advanced communication protocol for the proposed 896 Futurebus. *IEEE Micro*, 4(4), August 1984.

[14] Albert Chang and Mark F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[15] Ray Cheng. Virtual address cache in UNIX. In *Proceedings of the 1987 Summer USENIX Conference*, pages 217–224, 1987.

[16] David R. Cheriton, Anoop Gupta, Patrick D. Boyle, and Hendrik A. Goosen. The VMP multiprocessor: Initial experience, refinements, and performance evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 410–421, May 1988.

[17] David R. Cheriton, Gert A. Slavenburg, and Patrick D. Boyle. Software-controlled caches in the VMP multiprocessor. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 366–374, June 1986. Tokyo, Japan.

[18] Douglas W. Clark. Pipelining and performance in the VAX 8800 processor. In *Proceedings of the Second Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 173–177, October 1987. Palo Alto, California.

[19] Douglas W. Clark, Peter J. Bannon, and James B. Keller. Measuring VAX 8800 performance with a histogram hardware monitor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 176–185, June 1988. Honolulu, HA.

[20] Douglas W. Clark and Joel S. Emer. Performance of the VAX 11/780 translation buffer: Simulation and measurement. *ACM Transactions on Computer Systems*, 3(1), February 1985.

[21] S. H. Dahlby, G. C. Henry, D. N. Reynolds, and P. T. Taylor. The IBM system/38: A high-level machine. In D. Siewiorek, C. Gordon Bell, and Allen Newell, editors, *Computer Structures: Principles and Examples*, pages 533–546. McGraw-Hill, 1982.

[22] M. DeMoney, J. Moore, and J. Mashey. Operating system support on a RISC. In *Proceedings 1986 IEEE Computer Society International Conference (COMPCON)*, pages 138–143, March 1986.

[23] P. J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3), September 1970.

[24] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6(1):64–84, January 1980.

[25] Malcom C. Easton and Ronald Fagin. Cold-start vs. warm-start miss ratios. *Communications of the ACM*, 21(10), October 1978.

[26] S. J. Farnham, M. S. Harvey, and K. D. Morse. VMS multiprocessing on the VAX 8800 system. *Digital Technical Journal*, (4):111–119, February 1987.

[27] Craig R. Fink and Paul J. Roy. The cache architecture of the Apollo DN4000. In *Proceedings 1988 IEEE Computer Society International Conference (COMPCON)*, pages 300–302, March 1988. San Fransisco, CA.

[28] Michael J. Flynn, Chad L. Mitchell, and Johannes M. Mulder. And now a case for more complex instruction sets. *IEEE Computer*, 20(9), September 1987.

[29] J. Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, October 1961.

[30] S. J. Frank. Tightly coupled multiprocessor system speeds memory-access times. *Electronics*, pages 164–169, January 1984.

[31] John Fu, James B. Keller, and Kenneth J. Haduch. Aspects of the VAX 8800 C box design. *Digital Technical Journal*, (4):41–51, February 1987.

[32] B. Furht and V. Milutinovic. A survey of microprocessor architectures for memory management. *IEEE Computer*, 20(3):48–67, March 1987.

[33] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press Series in Computer Science. M.I.T. Press, Cambridge, MA, 1985.

[34] Armando Garcia, David J. Foster, and Richard F. Freitas. The advanced computing environment multiprocessor workstation. Technical Report RC 14491, IBM Research Report.

[35] Garth A. Gibson. SpurBus specification. Technical Report UCB/CSD 88/480, Computer Science Division (EECS), Univ. of California at Berkeley, December 1988.

[36] E. L. Glasser, J. F. Couleur, and G. A. Oliver. System design of a computer for time sharing applications. In *Proceedings of AFIPS 1965 FJCC*, volume 27, pages 197–202, 1965.

[37] James R. Goodman. Coherency for multiprocessor virtual address caches. In *Proceedings of the Second Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 72–81, 1987.

[38] R. N. Gustafson and F. J. Shapiro. IBM 3081 processor unit: Design considerations and design process. *IBM Journal of Research and Development*, 26(1):12–21, January 1982.

[39] P. D. Hester, R. O. Simpson, and A. Chang. The IBM RT PC ROMP and memory management architecture. In F. Waters, editor, *RT Personal Computer Technology*, pages 48–56. IBM, 1986. IBM Form No. SA23-1057.

[40] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout, and D. A. Patterson. Design decisions in SPUR. *IEEE Computer*, 19(11):8–22, November 1986.

[41] Mark D. Hill. The case for direct-mapped caches. *IEEE Computer*, 21(12):25–41, December 1988.

[42] Mark Donald Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, Computer Science Division (EECS), Univ. of California at Berkeley, November 1987. Available as Technical Report UCB/CSD 87/381.

[43] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures.* Computer Science Press, Potomac, Maryland, 1976.

[44] IEEE Standards Board. *NuBus - A Simple 32-Bit Backplane Bus*, December 1986. P1196 Specification, Draft 2.0.

[45] Deog-Kyoon Jeong. *Clocking and Synchronization Circuits in Multiprocessor Systems.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Univ. of California at Berkeley, April 1989. Available as Technical Report UCB/CSD 89/505.

[46] R. Joobami. *WEAVER: An Application of Knowledge-Based Expert Systems to Detailed Routing of VLSI Chips.* PhD thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon Univ., July 1985.

[47] Norman P. Jouppi. Architectural and organizational tradeoffs in the design of the MultiTitan CPU. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 281–289, June 1988. Honolulu, Hawaii.

[48] Gerry Kane. *Mips RISC Architecture.* Prentice Hall, 1987.

[49] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 276–283, June 1985. Boston, Mass.

[50] T. Kilburn. One-level storage system. *I.R.E. Transactions on Electronic Computers*, EC-11(2):223–235, April 1962.

[51] V. Knapp. *Virtually Addressed Caches for Multiprogramming and Multiprocessing Environments.* PhD thesis, Dept. of Computer Science, University of Washington, June 1985. Available as Technical Report 85-06-02.

[52] V. Knapp and J. L. Baer. Virtually addressed caches for multiprogramming and multiprocessing environments. In *Proceedings of the 18th Annual Hawaii Int'l Conference on System Sciences*, pages 477–486, 1985.

[53] A. E. Knowles, S. S. Thakkar, J. V. Woods, P. B. Lomas, and D. B. G. Edwards. Virtual memory management within MU6-G. Technical Report CS/E 85-004, University of Manchester, January 1985.

[54] Alan Knowles and Shreekant Thakkar. The MU6-G virtual address cache. Technical Report CS/E 84-007, University of Manchester, October 1984.

[55] Les Kohn, et al. Description of Intel i860 64-bit RISC-based microprocessor. *IEEE Micro*, 9(4):15–30, August 1989.

[56] J. Letz and J. Slingwine. Living with RISC: Software issues in the Regulus architecture. In *Proceedings of 1987 IEEE International Conference on Computer Design*, pages 549–557, 1987.

[57] E. M. McCreight. The Dragon computer system. In *Proceedings of the NATO Advanced Study Institute on Mircoarchitecture of VLSI Computers*, pages 83–101. Martinus Nijhoff Publishers, Dordrecht, 1985.

[58] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the Third Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 183–191, April 1989. Boston, MA.

[59] M. Kirk McKusick, Mike Karels, and Sam Leffler. Performance improvements and functional enhancements in 4.3BSD. Technical Report UCB/CSD 85/245, Computer Science Division (EECS), Univ. of California at Berkeley, June 1985.

[60] G. J. Myers, A. Y. C. Yu, and D. L. House. Microprocessor technology trends. *Proceedings of the IEEE*, 74(12), December 1986.

[61] Mike Nelson. Virtual memory for the Sprite operating system. Master's thesis, Computer Science Division (EECS), Univ. of California at Berkeley, June 1986. Available as Technical Report UCB/CSD 86/301.

[62] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[63] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[64] Roger G. Peterson. *Design and Analysis of Experiments.* Marcel Dekker, Inc., 1985.

[65] Bart Prieve. Personal Communication.

[66] Steven Przybylski, Mark Horowitz, and John Hennessey. Performance trade-offs in cache design. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 290–298, May 1988.

[67] Steven A. Przybylski. *Performance-Directed Memory Hierarchy Design.* PhD thesis, Computer System Laboratory, Stanford Univ., September 1988. Available as Technical Report CSL-TR-88-366.

[68] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 31–41, October 1987.

[69] S. A. Ritchie. TLB for free: In-cache address translation for a multiprocessor workstation. Master's thesis, Computer Science Division (EECS), Univ. of California at Berkeley, May 1985. Available as Technical Report UCB/CSD 85/233.

[70] Bryan S. Rosenburg. Low-synchronization translation lookaside buffer consistency in large-scale shared-memory multiprocessors. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 137–146, December 1989.

[71] M. Satyanarayanan and D. Bhandarkar. Design trade-offs in VAX-11 translation buffer organization. *IEEE Computer*, 14(12):103–111, December 1981.

[72] M. Schroeder. Performance of the GE-645 associative memory while Multics is in operation. In *Proceedings of the SIGOPS Workshop on System Performance Evaluation*, April 1971. Harvard University.

[73] S. Sims and J. Benkual. Regulus: A high performance VLSI architecture. In *Proceedings 1988 IEEE Computer Society International Conference (COMPCON)*, pages 48–50, March 1988. San Fransisco, CA.

[74] Jaswinder Pal Singh, Harold Stone, and Dominique F. Theibaut. An analytical model for fully associative cache memories. Technical Report RC 14232 (#63678), IBM Research Report, November 1988.

[75] Alan Jay Smith. Bibliography on paging and related topics. *Operating Systems Review*, 12(4):39–56, October 1978.

[76] Alan Jay Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*, SE-4(2):121–130, March 1978.

[77] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[78] Alan Jay Smith. Cache evaluation and the impact of workload choice. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 64–73, June 1985. Boston, Mass.

[79] Alan Jay Smith. Bibliography and readings on CPU cache memories and related topics. *Computer Architecture News*, 14(1):22–42, January 1986.

[80] Alan Jay Smith. Line (block) size choice for CPU caches. *IEEE Transactions on Computers*, C-36(9), September 1987.

[81] James E. Smith. Characterizing computer performance with a single number. *Communications of the ACM*, 31(10):1202–1206, October 1988.

[82] Stanford University. MIPS-X: A high performance computer, March 1985. Technical Progress Report.

[83] Sun Microsystems, Inc. *Sun-3 Architecture Manual*, July 1985.

[84] P. Teller, R. Kenner, and M. Snir. TLB consistency on highly parallel shared memory multiprocessors. In *Proceedings of the 21st Annual Hawaii Int'l Conference on System Sciences*, pages 184–192, 1988.

[85] Chris J. Terman. Simulation tools for digital LSI design. Technical Report TR-304, MIT Laboratory for Computer Science, Cambridge, MA, September 1983.

[86] S. S. Thakkar and A. E. Knowles. A high-performance memory management scheme. *IEEE Computer*, 19(5):8–22, May 1986.

[87] S. S. Thakker. *A High Performance Virtual Memory Management Unit for a Supermini Computer.* PhD thesis, University of Manchester, 1982.

[88] Michael Y. Thompson, J. M. Barton, T. A. Jermoluk, and J. C. Wagner. Translation lookaside buffer synchronization in a multiprocessor system. In *USENIX Winter Conference*, pages 297–302, February 1988.

[89] Shin-Yuan Tzou. *Software Mechanisms for Multiprocessor TLB Consistency.* PhD thesis, Computer Science Division (EECS), Univ. of California at Berkeley, 1989. Available as Technical Report UCB/CSD 89/551, Computer Science Division (EECS), Univ. of California at Berkeley, December 1989.

[90] Wen-Hann Wang, Jean-Loup Baer, and Henry M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 140–148, May 1989.

[91] D. A. Wood, S. J. Eggers, G. A. Gibson, M. D. Hill, J. M. Pendelton, S. A. Ritchie, G. S. Taylor, R. H. Katz, and D. A. Patterson. An in-cache address translation mechanism. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 358–365, June 1986. Tokyo, Japan.

[92] David A. Wood, Susan Eggers, and Garth Gibson. SPUR memory system architecture. Technical Report UCB/CSD 87/394, Computer Science Division (EECS), Univ. of California at Berkeley, December 1987.

[93] David A. Wood, Garth A. Gibson, and Randy H. Katz. Verifying a multiprocessor cache controller using random case generation. *IEEE Design and Test of Computers*, 1990. Available as Technical Report UCB/CSD 89/490, Computer Science Division (EECS), Univ. of California at Berkeley, January 1989.

[94] David A. Wood and Randy H. Katz. Supporting reference and dirty bits in SPUR's virtual address cache. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 122–130, May 1989. Jerusalem, Israel.

[95] Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Larus. SPUR Lisp: Design and implementation. Technical Report UCB/CSD 87/373, Computer Science Division (EECS), Univ. of California at Berkeley, September 1987.