Use of Superpages and Subblocking in the Address Translation Hierarchy

Madhusudhan Talluri

Technical Report #1277

August 1995

USE OF SUPERPAGES AND SUBBLOCKING IN THE ADDRESS TRANSLATION HIERARCHY

by

Madhusudhan Talluri

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Science)

at the

UNIVERSITY OF WISCONSIN - MADISON

1995

Abstract

Most computers that support virtual memory translate virtual addresses to physical addresses using a *translation lookaside buffer* (TLB) and a *page table*. Time spent in TLB miss handling—number of TLB misses times average TLB miss penalty—is increasing due to workload, architectural, and technological trends. This thesis studies TLB architectures that reduce the number of TLB misses by increasing TLB reach—the maximum address space mapped by a TLB— and page table designs that decrease TLB miss penalty or support new TLB architectures without increasing TLB miss penalty.

First, this thesis evaluates two TLB architectures in commercial use—*superpages* and *complete subblocking*. This thesis studies the benefits of superpages and the issues involved in modifying operating systems and page tables to support superpages. Complete subblocking allows processor designers to use larger chip areas to build large TLBs within cycle time constraints. Simulation results show that for comparable chip area, complete-subblock TLBs have faster access times and incur fewer TLB misses than single-page-size TLBs without requiring operating system changes.

Second, this thesis proposes a new TLB architecture, *partial subblocking*, that combines the best features of complete subblocking and superpages. Simulation results show that superpage and subblock TLBs, for comparable chip area, incur fewer TLB misses than single-page-size TLBs. Further, partial-subblock TLBs require simpler operating systems and incur fewer misses than superpage TLBs.

Third, superpage and partial-subblock TLBs are ineffective without operating system support. This thesis identifies the policies and mechanisms required to support these TLBs. In particular, this thesis proposes a physical memory allocation algorithm, *page reservation*, that makes partial-subblock TLBs effective or eliminates page copying in superpage creation.

Fourth, this thesis suggests modifications to conventional page tables to support superpage and subblock TLBs and proposes a new page table structure, *clustered page table*, that augments hashed page tables with subblocking. Simulation results show that clustered page tables are smaller and have a faster access time than conventional page tables when using single-page-size TLBs. A clustered page table improves on these advantages when storing superpage and subblock PTEs.

Acknowledgments

Graduate study at the University of Wisconsin-Madison has helped provide me with a solid education base and a great start in my career. I thank the admissions committee, faculty, secretarial staff, TAs and others at UW. I especially thank the faculty on the Wisconsin Wind Tunnel project and my thesis committee for their guidance, Mark Hill, David Wood, Guri Sohi, Marvin Solomon, James Smith, and James Larus.

Mark Hill, my advisor, supported my research assistantship with funds from his Presidential Young Investigator grant, Wisconsin Wind Tunnel project, and external grants from Sun Microsystems. David Ditzel, Shing Kong, Dock Williams, Emil Sarpa, John Entenmann, Karl Danz, Don Jaworski, Deepak Bhagat, Yousef Khalidi, Robert Roessler and others at Sun Microsystems were instrumental in funding my research through various grants and summer internships.

I thank Mark Hill for guiding and goading me through six years of fruitful research. I owe our papers that were published in three premier conferences to his patient pruning and polishing of my exceptionally long and wordy drafts. Shing Kong of Sun Microsystems started me off on the path of superpages. David Wood suggested the trap simulation method I use in my simulations. Robert Yung of Sun Microsystems suggested partial-subblocking. Konrad Lai of Intel and John Mashey of Silicon Graphics encouraged me to study complete-subblocking. Dock Williams of Sun Microsystems helped me study the operating system support issues and build Foxtrot. Norm Jouppi of DEC WRL helped me with the TLB access time model. Vikam Joshi, Adrian Caceras, and Yousef Khalidi of Sun Microsystems are co-inventors of the clustered page table. Praveen Seshadri, John Reppy, Babak Falsafi, and Prasad Wagle helped me with the workloads I used for my research. James Smith suggested the metric I use in my TLB studies. Yousef Khalidi helped me file patents and guide my research while at Sun Microsystems.

The computer architecture, Wisconsin Wind Tunnel, and Paragon project members at UW-Madison helped with their suggestions, criticisms, comments, and reviews. Many project groups at Sun Microsystems helped shape my research and develop Foxtrot—Advanced Architecture group at Sun Laboratories, UltraSPARC architecture group at SMCC/Sparc Tech, Kernel-MT and Kernel-VM groups at SunSoft, and Desktop and Server Software groups at SMCC.

Many people—officemates, roommates, study groups, wind tunnel project group, architecture group, and "other" friends too many to list—helped make my stay at Madison a very special one. A special thanks goes to Guhan for doing the paperwork and filing my thesis in Wisconsin while I was away in California, and to Shivku for taking care of my bills and mail on my many internship visits to Sun.

Moving to a foreign country is often a traumatic experience. I would like to thank the people that made the transition pleasant. The International Student Office, the Wollner family—my host family through MFIS, Dr. Sham Sundar Rao and family, Gopu, Chitluri and Kumudan families, and many UW students and families. The dance department at UW, the University of Wisconsin Ballroom Dance Association (UWMBDA) and many dancing buddies take credit for introducing me to social dancing and some of the best times of my life.

I thank my family for exceptional support during my education. My parents for sending me to a good school out-of-town and a great university in a foreign country; my grandparents for bringing me up and guiding me through the school years; many aunts, uncles, and cousins for their support over the years; and finally, my wife Sridevi for all the help and support during the final years of my Ph.D. and for putting up with many late nights at work.

TABLE OF CONTENTS

Chapter 1 Introduction

1.1 Problem Description

Paged virtual memory distinguishes addresses used by programs (virtual addresses) from the real memory addresses (physical addresses). On every memory access the system translates a virtual address to a physical address. This indirection allows access to more memory than physically present, transparent relocation of program text and data, and protection between processes [Denn70]. A *page table* stores the translation and protection information and a *translation lookaside buffer*[1] (TLB) caches recently used translations to accelerate the translation process [Lee69, Smit82, Mile90]. The TLB and page table make up the *address translation hierarchy* that is the focus of my study.

Time spent in TLB miss handling is an overhead of virtual memory and is equal to the number of TLB misses incurred times the average time to traverse the page table (TLB miss penalty). In the early 1980s, TLB miss handling was a small fraction of the processor's cycles-per-instruction (CPI) [Clar85, Wood86]. Many workload, technology, and architecture trends have combined to increase both the number of TLB misses and the TLB miss penalty, increasing the amount of time spent in TLB miss handling.

One long-standing computer trend is that programs' memory usage doubles each year or two [Henn90]. To support the larger program working set sizes [Denn68], workstations with more than 100MB of physical memory are becoming common. This places pressure on the TLB to map an increasingly larger amount of memory. Innovative uses of virtual memory to implement new functionality, such as distributed shared memory, is increasing the address space that a TLB should map. Programs incur a large number of TLB misses if their working set is larger than the *TLB reach*—maximum amount of address space mapped by a TLB. This is analogous to an increase in disk paging if the working set is larger than the amount of available physical memory. There are at least two ways of increasing TLB reach—by increasing the number of TLB blocks (or entries) or by increasing the address space mapped by each TLB block.

The number of TLB blocks is not increasing significantly as TLBs are now implemented on the microprocessor chip where chip area, access time, and cycle time constraints limit TLB size. First, access time constraints limit TLB designers from using the larger number of transistors and chip area available today [Gels89] to increase the number of TLB blocks. Larger TLBs are slower to access and affect cycle time. TLB access time is an important metric as TLBs are often in the cache-access critical path. Cache consistency management by the operating system has complicated efforts to remove the TLB from the critical path using virtually-tagged caches [Whee92]. Physically-tagged caches continue to be common. Second, the trend toward wider superscalar processor implementations [Joup89] requires TLBs to support multiple translations per cycle through multi-porting or replication. TLBs that support multiple transactions per cycle are slower to access and occupy larger chip area. This reduces the number of TLB blocks that can be implemented for fixed chip area or access time constraints. Third, larger virtual and physical address sizes, *e.g.*, 64-bit virtual addresses, increase the number of bits stored in a TLB, further reducing the number of TLB blocks for a fixed transistor count. Thus, there is need for innovative ways to increase TLB reach with little or no in-

1. Also known as Translation Buffer (TB), Directory LookAside Table (DLAT), Address Translation Cache (ATC) or Memory Management Unit (MMU)

crease in the number of TLB blocks.

One way to increase the address space mapped by a TLB block is to increase the page size. Doubling the page size, for example, doubles TLB reach. Large page sizes, however, increase physical memory usage due to internal fragmentation [Denn70] because the page size is larger than what the program needs. While the smallest supported page sizes have increased modestly from 512 bytes in VAX 11/780 to 8KB in Alpha, virtual and physical memory sizes have increased by orders of magnitude, *e.g.*, from 640KB to 64MB. Two factors restrict the page size choice. First, microprocessors are designed to be used in a variety of computer systems from large-memory servers to small-memory laptops. A large page size will restrict the processor to large-memory machines. Extra paging in small-memory machines from use of larger pages makes the large pages unattractive. Increasing the page size from 4KB to 64KB, for example, doubles the working set size for some programs [Tall92] and can increase paging. Second, the page size is an architectural feature that changes only during major transitions in processor architecture, such as from VAX [Levy82] to Alpha [Site93] or from SPARC V8 [SPAR91] to SPARC V9 [SPAR94]. On the other hand, cache line size is an implementation parameter more easily changed.

TLB miss penalty is also increasing due to many reasons. First, as processors become faster relative to main memory accesses [Henn90], page table traversal—the main component of TLB miss penalty—becomes relatively slower. Second, page table size has been increasing due to larger address spaces and larger *page table entry* (PTE) size, *e.g.*, four bytes to eight bytes. This increases cache pollution and reduces the likelihood of completing page table traversal within the CPU caches. Third, many processors support TLB miss handling in software, *e.g.*, ZS-1 [Smit87], AMD29000 [John87], MIPS [Kane92], Alpha [Site93], UltraSPARC [Yung95], PA7100 [Aspr93], that incurs higher overhead than hardware state machines. A small five cycle overhead to drain the processor pipeline before trapping to software has an opportunity cost of twenty instructions in a four-way superscalar processor. Memory system designers are addressing the increasing TLB miss penalty with more levels in the address translation hierarchy by using a second-level *software TLB*, *e.g.*, swTLB [Huck93], TSB [Yung94], STLB [Bala94].

My thesis looks at increasing TLB reach through use of variable block size and subblocking techniques to map more address space per TLB block. My thesis addresses the trend toward higher TLB miss penalties by proposing page table designs that are better than conventional page tables or proposing modifications to conventional page tables to support the new TLB architectures without increasing TLB miss penalty. I make four significant contributions in the areas of TLB and page table design.

- I evaluate two TLB architectures in commercial use today that have a larger TLB reach than single-page-size TLBs of equivalent chip area and access time—*superpage*[2] (Chapter 3) and *complete-subblock* TLBs (Chapter 4).

- I propose a new TLB architecture—*partial-subblock* TLB—that is more effective at reducing the number of TLB misses than single-page-size, medium-size superpage, and complete-subblock TLBs of comparable implementation complexity (Chapter 5).

- I identify the operating system policies and mechanisms required to support superpage and partial-subblock TLBs. Further, I have implemented some policies and mechanisms in So-

2. I use the term first suggested by Mogul [Mogu93].

laris 2.1, a commercial operating system (Chapter 6).

- I propose a new page table structure, *clustered page table*, that has a lower page table access time, occupies less memory, and stores superpage (and partial-subblock) mappings more efficiently than conventional page tables (Chapter 7).

Section 1.2 explains, in brief, the new TLB and page table architectures and key results of my thesis—Chapters 3-7 include a detailed description and evaluation. Section 1.3 includes references to published literature of other research in this area. Section 1.4 brings out the relationship between my previous published papers and my thesis. Section 1.5 explains a typical hardware implementation of conventional single-page-size TLBs and TLB miss handling. Later chapters show extensions required to support the new TLB architectures. Table 1-1 includes a definition of terms and naming conventions that I use throughout the thesis.

Table 1-1: Definition of terms

Term	Definition
Address	An address is virtual unless explicitly identified as a physical address
Aligned	A region of contiguous memory of size B is aligned if it starts at a virtual or physical address that is a multiple of B
Page	A page is a contiguous region of address space, virtual or physical, that is power-of-two aligned, *e.g.*, 4KB
Base page	Base page size is the smallest page size supported in a system, *e.g.*, 4KB. A base page is a page of that size
Page block	A page block is a contiguous region of address space, virtual or physical, that is aligned to a power-of-two multiple of the base page size, *e.g.*, 64KB
Subblock factor	Subblock factor is the number of base pages in a page block
Superpage	A superpage is a page block where all the base pages have superpage compatible mappings (see below)
VPN (PPN)	VPN (PPN) is the virtual (physical) page number—the virtual (physical) address divided by the base page size
VPBN (PPBN)	VPBN (PPBN) is the virtual (physical) page block number—the virtual (physical) address divided by the page block size
Virtual (Physical) block offset	Virtual (Physical) block offset is the virtual (physical) page number mod subblock factor (mod is the modulus operator)
Mapping	A mapping stores the translation and protection information for one base page, superpage, or page block
TLB block	A TLB block consists of one or more valid bits and a VPN or VPBN (and process ID) as tag and one or more mappings as data. Also known as a TLB entry
Subblock-	A subblock- prefix identifies a property of a base page within a TLB-block or page block, *e.g.*, a subblock-valid bit refers to the valid bit corresponding to a base page within a TLB block that maps multiple base pages
Page table entry	A page table entry (PTE) consists of one or more mappings, and optionally, a VPN or VPBN (and process ID) as tag
Page block aligned	A base page mapping is page block aligned if the virtual and physical block offsets are equal, *i.e.*, VPN(p) mod s = PPN(p) mod s, where mod is the modulus operator and s is the subblock factor

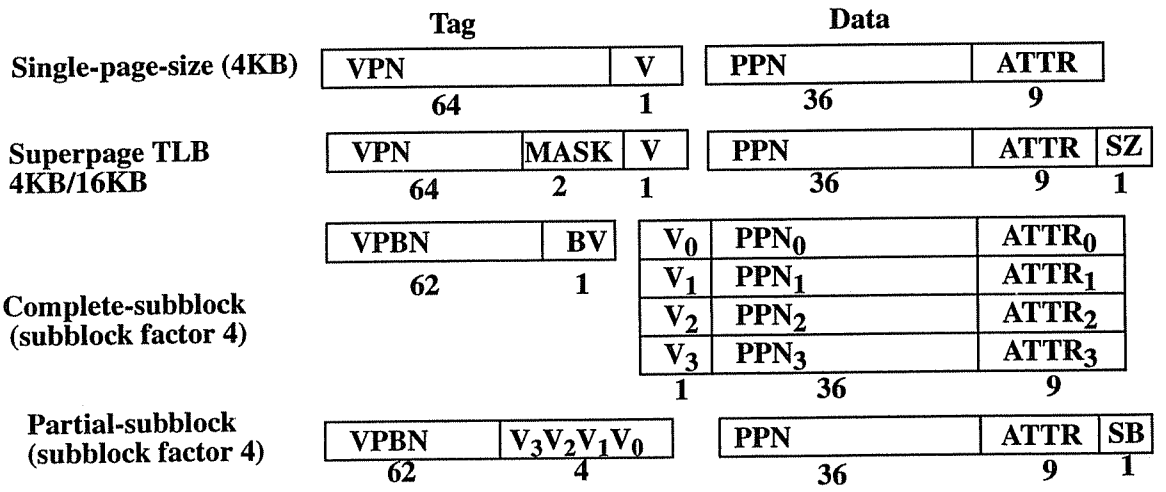
Table 1-1: Definition of terms

Term	Definition
Properly placed	Base pages x and y are properly placed if they are placed in the same virtual and physical page blocks, *i.e.*, VPBN(x) = VPBN(y) and PPBN(x) = PPBN(y), and are both page block aligned, *i.e.*, VPN(x) mod s = PPN(x) mod s and VPN(y) mod s = PPN(y) mod s, where s is the subblock factor (see page 56)
Compatible	Two (or more) mappings are compatible to share a TLB block (or PTE) if their virtual addresses fall within the same virtual page block, *i.e.*, VPBN(p1) = VPBN(p2), and satisfy a compatibility constraint that depends on the TLB type
Superpage compatible	Mappings for a virtual page block are superpage compatible if they all are valid, all have the same attributes, and all are properly placed with respect to each other
TLB reach	The maximum amount of address space that a TLB can map, *i.e.*, the number of TLB blocks times the maximum page block size for each TLB block
Bit ordering	I use little-endian notation for numbering bits within a word, *i.e.*, bit_0 is in the least significant bit

1.2 Summary of thesis

A conventional, or single-page-size, TLB block stores a translation for a fixed-size *base page* using the virtual page number (VPN) and valid bit (V) for a tag and physical page number (PPN) and page attributes (ATTR) as data. My thesis explores three ways to increase TLB reach by allowing a single TLB block to map multiple base pages. Figure 1-1 illustrates the TLB block formats for the different TLB architectures with length of the bit fields in bits. I discuss the details of each field in the appropriate chapters (Chapters 3-5). The first approach allows each TLB block to map a variable sized "page", *i.e.*, superpages. To benefit from superpages, however, requires significant operating system changes. The second approach uses naive subblocking, complete-subblocking, where each TLB block maps several base pages but includes individual mappings for base pages. The third approach, partial-subblocking, maps several base pages per TLB block but includes in each TLB block only individual subblock valid bits. Both varieties of subblocking require simpler operating system changes than superpages.

Figure 1-1: Comparison of TLB blocks of different TLB architectures



All three types of TLBs attempt to improve TLB performance by storing in a single TLB block mappings to multiple base pages more efficiently than separate single-page-size TLB blocks. To be effective, however, base pages belonging to the same virtual page block must be simultaneously active in the TLB. These TLBs can hold mappings to more base pages than a monolithic single-page-size of comparable implementation cost. Prefetching mappings for neighboring base pages on a single TLB miss further reduces the number of TLB misses. These techniques are effective when spatial locality [Denn75] makes it likely that consecutive base pages are in contemporaneous use.

The three TLB architectures differ in the conditions under which mappings to base pages within a page block can share a single TLB block (Table 1-2). Superpage TLB blocks are used only when all base pages within the page block are valid, properly placed in physical memory, have the same attributes, and the operating system has promoted the page block to a superpage. Partial-subblock TLB blocks can store mappings to multiple base pages even if just two or more base pages have valid, properly placed mappings with the same attributes. A complete-subblock TLB block can store multiple valid mappings for base pages within a page block without any restrictions.

Table 1-2: Summary of when base pages within a page block can share a single TLB block

valid mappings	properly placed in physical memory	same attributes	Operating system page promotion	Superpage TLB block	Partial-subblock TLB block	Complete-subblock TLB block
all	all	all	YES	X	X	X
all	all	all	NO		X	X
	some[a]		N/A		X	X
1 or more	N/A	N/A	N/A			X

a. some => two or more valid, properly placed base page mappings with the same attributes

Superpages have sizes that are power-of-two multiples of the *base page size* and must be aligned in both virtual and physical memory (Chapter 3). Many processors now support superpages, *e.g.*, MIPS [Kane92], UltraSPARC [Yung95], Alpha [Bann95], PowerPC [Silh93], HP-PA RISC [Hunt95]. A fully-associative TLB can easily include support for superpages. An example is the MIPS R4000, which supports a 4KB base page size and superpages of 16KB, 64KB, 256KB, 1MB, 4MB, and 16MB with a fully-associative TLB. Set-associative TLBs typically use the least significant bits of the VPN as index bits and are not trivial to extend to support variable superpage sizes [Tall92]. Large superpages, 256KB and larger, are most useful for unpageable memory and devices, *e.g.*, kernel text, frame buffer, and database buffer pools. If there are only a few large superpages in use, their mappings may be set up with limited changes to existing operating systems. As using superpages results in larger memory usage and I/O costs due to internal fragmentation, programs are more likely to use medium-size superpages in the range of 16KB to 64KB. However, medium-size superpages or generic uses of larger superpages require substantial operating system changes in mechanisms to support them and policies for choosing appropriate page sizes. Chapter 3 explores how to build superpage TLBs and handle TLB misses in superpage TLBs. TLB simulation results show that, with an operating system that uses superpages, medium-size superpage TLBs result in significant execution time speedup for the workloads described in Section 2.4.

Subblocking associates mappings for multiple base pages with each TLB tag, thus increasing TLB reach (Chapter 4). With a *subblock factor* of sixteen and 4KB base pages, for example, each tag covers a 64KB page block. Each subblock-TLB block also includes multiple subblock-valid bits that allow individual base page mappings to be loaded into the TLB. Chapter 4 shows a naive implementation of subblock TLBs, *i.e., complete-subblocking*, that stores in each TLB block the full base page mappings for base pages in the page-block. Complete subblock TLBs use similar implementation technology as subblock caches, do not require any operating system support, can use prefetching without displacing other useful translations in the TLB, and incur fewer TLB misses than superpage TLBs with the same TLB reach. Complete-subblock and single-page-size TLBs that have the same TLB reach have the same number of bits in the data memory. Complete-subblock TLBs, however, use tag memory more efficiently than single-page-size TLBs. Simulation results in Section 4.3 comparing alternate implementations for a fixed chip area show that complete-subblock TLBs nearly always incur fewer TLB misses than single-page-size TLBs. A disadvantage of complete-subblocking is that each TLB block's data area is large, because it contains multiple mappings. The next design attempts to address this issue by using the data memory more efficiently also.

Chapter 5 introduces a third way to improve TLB reach—*partial subblocking*—that requires less operating system support than medium-size superpages and uses less chip area than complete-subblock TLBs. Partial-subblock TLBs use less area than complete-subblock TLBs by storing only one set of page attributes and a single PPN per TLB block. To make effective use of a partial-subblock TLB requires base virtual pages in a page block to be placed in a single, aligned block of physical memory, *i.e., properly placed*. Pages not properly placed are allowed but use multiple TLB blocks. Thus, if the operating system can implement a good physical memory allocation algorithm, partial-subblock TLBs can be as effective as complete-subblock TLBs but use significantly smaller chip area. Superpages require more complicated operating systems than partial-subblock TLBs, because the hardware requires the operating system guarantee that base pages within a page block are superpage compatible. By storing subblock valid bits, partial-subblock TLBs require only a best-effort by the operating system. Further, the operating system need not wait for all base pages within a page block to be present in memory to share TLB blocks. A single partial-subblock TLB block suffices, for example, if only ten of sixteen pages of a page block are memory resident whereas using a superpage TLB block requires all sixteen to be resident. Chapter 5 studies alternate ways to build partial-subblock TLBs and TLB miss handling techniques, including subblock prefetching. Simulation results comparing single-page-size, superpage, and both types of subblock TLBs show that for fixed chip area partial-subblock TLBs result in the best execution time speedups for the workloads I consider (Section 5.5).

In comparing different TLB configurations, TLB access time and chip area are important metrics. I estimate TLB access time and chip area cost using analytical models adapted from similar models developed for caches. Section 2.2 describes the area model adapted from Mulder's model [Muld91] and Section 2.3 describes the access time model adapted from Jouppi and Wilton's model [Wilt93]. In Chapters 3 to 5, I compare TLB configurations of comparable chip area to show that the new TLB architectures not only improve execution time but often result in a TLB with faster access time[3].

I illustrate the effectiveness of the new TLB architectures by comparing three alternate fully-associative TLBs that occupy comparable area to a 64-block fully-associative TLB (the ac-

3. The access times estimates in this thesis for superpage and subblock TLBs are pessimistic and real implementations can be expected to be faster.

cess times are also comparable)—a 62-block superpage TLB that supports a 4KB base page size and a 32KB superpage size with the page size decisions made as described in Section 2.7.1, a 57-block partial-subblock TLB with subblock factor 16 and preloading in the TLB miss handler as described in Section 5.3.3, and a 35-block complete-subblock TLB with subblock factor four without preloading. The superpage and subblock TLBs have fewer TLB blocks than the single-page-size TLB but have a larger TLB reach and better performance. Table 1-3[4] shows the normalized execution time speedup (defined in Section 2.5) relative to using a 64-block fully-associative single-page-size (4KB) TLB.

Table 1-3: Key TLB performance results—normalized execution time speedup relative to using 64-block fully-associative single-page-size (4KB) TLB

64-block Single-page-size (4KB) TLB	62-block Superpage (4KB/32KB) TLB	57-block partial-subblock TLB (subblock factor 16)	35-block complete-subblock TLB (subblock factor 4)
1.00	1.18	1.21	1.04

The important conclusion from Table 1-3 is that there are alternate TLB designs to a monolithic single-page-size TLB that are of comparable implementation complexity but deliver good execution time speedups. The speedups are not gigantic (4% to 21%) even with my overemphasis on workloads that spend significant time in TLB miss handling (Section 2.4). Future workloads, 64-bit and object-oriented, that spend more time in TLB miss handling can get higher execution time speedups. To realize these speedups, however, requires low-overhead operating system support for superpage and partial-subblock TLBs, and page table support to keep the TLB miss penalties comparable or smaller.

Operating system support for paged virtual memory with a single fixed page size is substantial but well-understood (*e.g.*, UNIX [Thom74, Bach86, Leff90], VMS [Levy82], NT [Cust93], MACH [Acce86, Rash88], OS/2 [Koga88]). Most facets of paged virtual memory operating system policies and mechanisms require modifications to support superpages effectively. A new policy—*page-size assignment*—and upto six new mechanisms may also be required. A page-size assignment policy decides when to use superpages, what size superpages, and for which address space regions. Chapter 6 describes the operating system support required, discusses alternate page-size assignment policies, the mechanisms required to support the policies, and interaction with other operating system policies. Operating system support for partial-subblock TLBs does not include page-size assignment, but requires a different physical memory allocator to optimize TLB usage. Section 6.2.5 describes *page reservation*, a new physical memory allocation algorithm, that commonly allocates properly placed physical pages for partial-subblocking. Page reservation also helps in efficient creation of superpages. Section 2.7 describes the specific policy I use in superpage and partial-subblock TLB simulations.

Reducing the number of TLB misses addresses only part of the time spent in TLB miss handling. Reducing TLB miss penalty is equally important. TLB miss penalty is dependent on the page table structure. In Chapter 7, I propose a new page table structure, *clustered page table*, that extends a hashed page table with subblocking—a clustered page table is a complete-subblock hashed page table. My results on a single-page-size system comparing clustered page tables with conventional page tables show that clustered page tables use less memory and are faster to access. Clustered page tables have additional advantages when supporting superpage or subblock TLBs.

4. Section 5.5 includes a comparison with more alternate TLBs and chip areas.

If page tables do not properly support superpages and subblocking, increases in TLB miss penalty can offset some or all of the gains from reduction in the number of TLB misses. In Sections 3.3 and 5.3, I show that superpage and partial-subblock TLB miss handling is most efficient if the operating system can construct and store in the page table superpage and partial-subblock PTEs that coalesce multiple base page PTEs into a single PTE. Chapter 7 shows how popular page tables can be extended to support superpage and partial-subblock PTEs. In particular, by replicating them at every base page PTE site the TLB miss penalty is no worse than in a single-page-size system but using the new PTEs reduces the number of TLB misses. It is often desirable for operating systems to store a single copy of a superpage PTE, *e.g.*, for efficient update in multi-threaded operating systems. Clustered page tables store superpage and partial-subblock PTEs without replication using the same superpage and partial-subblock techniques used in TLBs. In addition, clustered page tables support the new TLB architectures using less memory, and are often faster to access than other page tables. Section 7.5 discusses how to store superpage and partial-subblock PTEs in an operating system data structure that maintains aliases to physical pages, a *synonym table.*

My thesis shows that superpages and subblocking are effective ways to increase TLB reach, discusses the hardware implementation issues, and operating system support required to make the new TLBs effective, and proposes a page table that again uses superpages and subblocking to reduce page table memory usage and TLB miss penalty.

1.3 Related Work

In this section, I review related work on segments, TLBs, caches, superpages, and page tables. The next section reviews my previous work. While few published literature exist on the use of superpages and subblocking in TLBs, the basic ideas of superpages and complete subblocking appear in earlier systems.

Pure segmented systems allow allocation of arbitrary sized regions of memory and were popular in early computer systems, *e.g.*, Multics [Orga72] and Burroughs B5000 [Bur61]. Segments use a two-dimensional address space, may be arbitrarily long, and may start at arbitrary physical addresses. Supporting superpages is easier than supporting segments because superpages have alignment restrictions that allow hardware to use bit steering instead of adders that segments require. Further, simpler versions of the algorithms used in segmented operating systems may be applicable in superpage operating systems, *e.g.*, memory allocation [Knut68a], segment-size assignment [Redd75], and variable-sized segment and page replacement [Prie76, Fran74, Turn81]. Smith compiled a bibliography of early virtual memory research that includes research on segmented systems [Smit78c]. Operating systems still use segments to represent objects in address spaces, but most current operating systems treat all physical memory as fixed-size frames or pages, allowing portions of segments to be present in memory. This segmentation can be either invisible to hardware using a linear address space model, *e.g.*, VAX [Leon82] and MIPS [Kane89], or visible to hardware using a paged-segmentation model. In a paged-segmentation model, programs generate a <segment identifier, segment offset> tuple that first translates to a global effective virtual address before translating to a physical address [Knig81, Dall92]. Examples include Honeywell 645 [Glas65], SPUR, [Hill86], HP-PA RISC [Lee89b], IBM RS/6000 [Chan90], and PowerPC [May94]. TLBs and page tables translate the virtual address to physical address, and superpages or subblocking are equally applicable as described in this thesis. If segmentation is visible to the hardware, base page protection and attributes may be enhanced through support for segment protections, protection lookaside buffers [Kold92], page-groups [Wilk92], or capabilities

[Fabr74].

Various researchers have studied TLB design and extensions to improve TLB performance that may complement use of superpages or subblocking. I list some literature relating to TLBs that may be useful for future reference. A survey paper on cache memories by Smith also describes TLBs and related design parameters [Smit82]. The TLB in the VAX 11/780 system is the focus of some studies [Saty81, Clar85, Alex85, Alex86]. Their studies show that time spent in TLB miss handling is less than 5%, for the workloads (including multiprogrammed[5] workloads) used in early 1980s. Workload changes have made TLBs more important, as shown in later studies [Chen92, Tall92]. Some innovative designs that attempt to reduce TLB access time include the TLB-slice [Tayl90], micro-TLB [Chen92], lazy address translation [Chiu92], and fast address calculation [Aust95]. TLB misses are often handled by hardware that traverses page tables. Some processors support TLB miss handling in software and Nagle *et al* discuss issues in software TLB miss handling using MIPS processors as examples [Nagl94b]. In multi-processor systems TLB coherence becomes an issue. Teller describes many strategies for maintaining TLB coherence [Tell90]. Many operating systems use a conservative TLB shootdown algorithm, *e.g.*, [Blac89]. The SPUR [Wood86] and Fugu [Mack94] machines combine TLB coherence with existing cache coherence mechanisms. Systems that support paged-segmentation typically include two translation buffers, a TLB and a SLB (segment lookaside buffer), that are accessed one after another. Dally shows a scheme that combines the segment and page translation [Dall92].

TLBs have traditionally been a second-order performance concern, as programs often incur a higher overhead in cache miss handling. With the use of large multi-megabyte caches [Kess91] and innovative uses of virtual address spaces [Appe91a, Blum94], some applications now incur more TLB misses than cache misses. Fortunately, there is a large body of research in cache design [*e.g.*, Smit86, Smit91] that is largely applicable to TLBs also—TLBs have a structure similar to caches (Section 1.5).

In particular, my thesis applies to TLBs and page tables three techniques borrowed from cache design—variable block size [Dubn92], subblocking [Lipt68, Bell74, Good83, Hill84], and subblock prefetching [Smit78b, Hill87]. Superpage TLBs implement a variable block size design with the policy decisions on when to use superpages made by the operating system. A subblock-cache associates with each address tag several data subblocks that each have their own valid bits so that they can be loaded independently. A complete-subblock TLB uses the same techniques as subblock-caches. The partial-subblock design optimizes a subblock design using specific knowledge about the structure and content of the data stored in a TLB.

A key motivation for my thesis was the introduction of superpage support in many microprocessor TLBs, *e.g.*, MIPS [Kane92], UltraSPARC [Yung95], Alpha [Bann95], PowerPC [Silh93], HP-PA RISC [Hunt95]. Commercial operating systems I am aware of, however, do not support general use of superpage mappings. Many operating systems include special mechanisms to use large superpages for unpageable memory and devices. While some have suggested uses for superpages [Chen92, Mogu93], I believe my thesis (and my previous work) is the first to study the issues involved in building superpages TLBs and supporting them. My results show that superpage TLBs are largely ineffective and a waste of hardware resources if operating systems and page tables do not support them.

5. By multiprogramming, I mean execution of multiple concurrently active processes.

Operating system support for superpages involves implementing some mechanisms (described in Section 6.2), *e.g.*, variable-size memory allocation [Knut68a], and a page-size assignment policy (described in Section 6.1). Romer *et al.* [Rome95] study the use of competitive algorithms for page-size assignment among multiple superpage sizes. I use working-set based page-size assignment (described in Section 6.1) in my work [Tall92, Tall94a].

Three styles of page tables are popular[6]—linear (*e.g.*, VAX [Levy82]), forward-mapped (*e.g.*, SPARC [SPAR91]), and hashed/inverted (*e.g.*, PowerPC [May94], IBM System/38 [IBM78]). Many forward-mapped page table implementations and guarded page tables [Lied95] support certain superpage sizes at their intermediate nodes. Hashed page tables are being increasingly used to support sparse 64-bit address [Houd68, Abra81, Thak86, Rose92, Huck93, May94] but none support superpage mappings. Page table management algorithms in a multi-threaded multiprocessor operating system [Bala92, Khal94] also affect system performance, however, they execute infrequently compared to TLB misses.

1.4 My Previous Work

A paper titled *"Tradeoffs in Supporting Two Page Sizes"* by Talluri *et al.* [Tall92] first addresses the costs and benefits of using large page sizes. Using larger page sizes increases the working set size but reduces the number of TLB misses. The paper suggests the simultaneous use of two page sizes with an operating system page-size assignment policy to decide the appropriate page size for every virtual address. It also shows how to build fully-associative and set-associative TLBs to support two page sizes. Using results from trace-driven simulations the paper shows that using two page sizes can reduce the number of TLB misses with only a small increase in the working set size. Chapter 3 gives an updated presentation of this material using results from larger workloads, a real operating system implementation, a chip area model, and an access time model.

A paper titled *"Surpassing the TLB performance of Superpages with Less Operating System Support"* by Talluri and Hill [Tall94a] summarizes the new TLB architectures that my thesis presents. It proposes partial-subblock TLBs, compares the TLB performance of superpage, complete-subblock, and partial-subblock TLBs, and the operating system support required for these TLBs. Chapters 3-5 describe in detail the new TLB architectures, how to build such TLBs, how to handle TLB misses, and include detailed performance studies.

This thesis includes two significant changes from the above paper that offset each other to leave the conclusions unchanged. First, in doing the TLB simulations for this thesis, I uncovered a bug that overestimated the number of TLB misses for superpage TLBs reported in the paper. Second, in designing partial-subblock TLBs in Chapter 5 and page tables for them in Chapter 7, I found a simple extension to partial-subblock TLBs, preloading, that results in a simpler hardware implementation, smaller TLB miss penalty for many page tables, and significantly reduces the number of TLB misses. The net effect of these two changes is that the conclusions of the paper remain the same—partial-subblock TLBs incur fewer TLB misses than superpage TLBs and require less operating system support. For the workloads used in the paper, superpage TLBs incur fewer TLB misses than partial-subblock TLBs without preloading but more TLB misses than partial-subblock TLBs with preloading.

A paper titled *"Virtual Memory Support for Multiple Page Sizes"* by Khalidi *et al.* [Khal93b] explains the importance of operating system support for superpage TLBs and lists the issues

6. For lack of a standard page table terminology in literature, I use the same terminology as Huck and Hays [Huck93]

that need to be addressed in operating system implementation or design to support superpage TLBs. Chapter 6 identifies the mechanisms and policies that need to be implemented in an operating system to support superpage or partial-subblock TLBs. While I suggest some page-size assignment policies and algorithms for the various mechanisms, this area merits further operating system research.

A paper titled *"A New Page Table for 64-bit Address Spaces"* by Talluri *et al.* [Tall95] reviews the suitability of conventional page tables—linear, forward-mapped and hashed—for 64-bit address spaces and superpage mappings. The paper then proposes a new page table, clustered page table, that extends hashed page tables using the same subblocking techniques that this thesis uses to make TLBs more effective. Clustered page tables can have a faster access time, occupy less memory, and are more efficient at storing superpage and partial-subblock PTEs than conventional page tables. Chapter 7 is an expanded version of this paper. Solaris 2.5, a commercial operating system, implements clustered page tables as explained by Khalidi *et al.* [Khal95a] and in patent applications [Tall93, Khal93a].

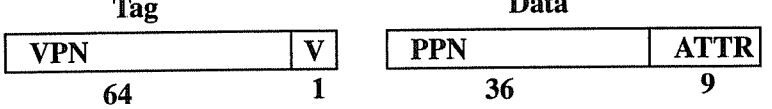
A technical report titled *"Improving the Address Translation Performance of Widely Shared Pages"* by Khalidi and Talluri [Khal95b, Tall94b] addresses TLB performance and page table size in the presence of large number of aliases for physical pages, *e.g.*, shared libraries. It suggests a *common-mask scheme* for TLBs and hashed page tables where "correctly-placed" aliases share a single TLB block or page table entry. This approach increases TLB reach and reduces page table size in an orthogonal way to the use of superpages or subblocking and the two approaches can be combined. I do not describe this work further.

1.5 Mechanics of a single-page-size TLB

A TLB, being a cache of virtual-to-physical address translations, is constructed similar to a CPU data or instruction cache [Smit82]. A processor or the memory system accesses a TLB with a virtual address (VA) to translate it to a physical address (PA)—typically before or in parallel to accessing a physically-tagged cache or main memory. If the TLB has a matching translation—a TLB hit—it outputs the physical address and memory access attributes. If the TLB does not have a matching translation—a TLB miss—special hardware or software fetches the missing translation by traversing a page table—TLB miss handling—and loads it into the TLB.

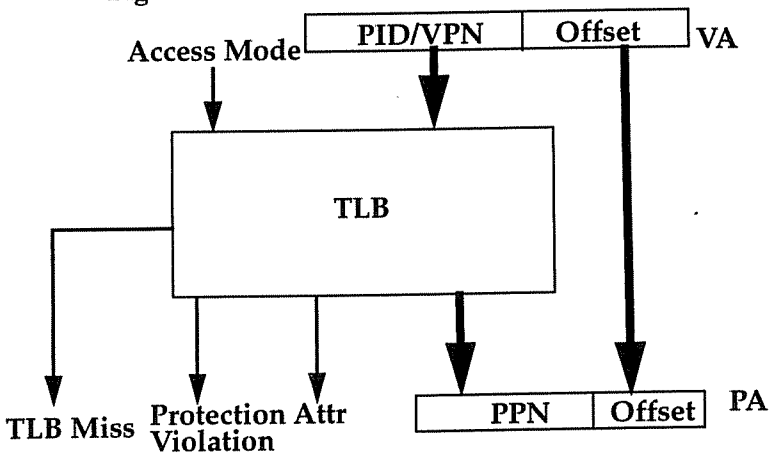
A TLB stores translations in TLB blocks, each containing a tag and a data part. The tag contains the virtual page number (VPN) of the translation and a valid bit (V). The data part stores the corresponding physical page number (PPN) bits and page attributes (ATTR), *e.g.*, protection, cacheability, referenced/modified bits. Figure 1-2 shows a sample TLB block for 64-bit virtual addresses with the length of the fields in bits—the VPN includes a 12-bit process identifier (PID) [Dekk87]. Note that the TLB tag has more bits than the data. This is a significant difference from cache designs where tags, *e.g.*, four to eight bytes, are much smaller than a cache block size, *e.g.*, 32-256 bytes.

Figure 1-2: Structure of a single-page-size (4KB) TLB block



Many such TLB blocks can be combined in either a fully-associative or set-associative form as explained in Sections 1.5.1 and 1.5.2. In either case, a tag array stores all the tags and includes comparators to compare them with the input VA. A random-access-memory (RAM) stores the data parts of the TLB blocks.

Figure 1-3: Structure of a conventional TLB



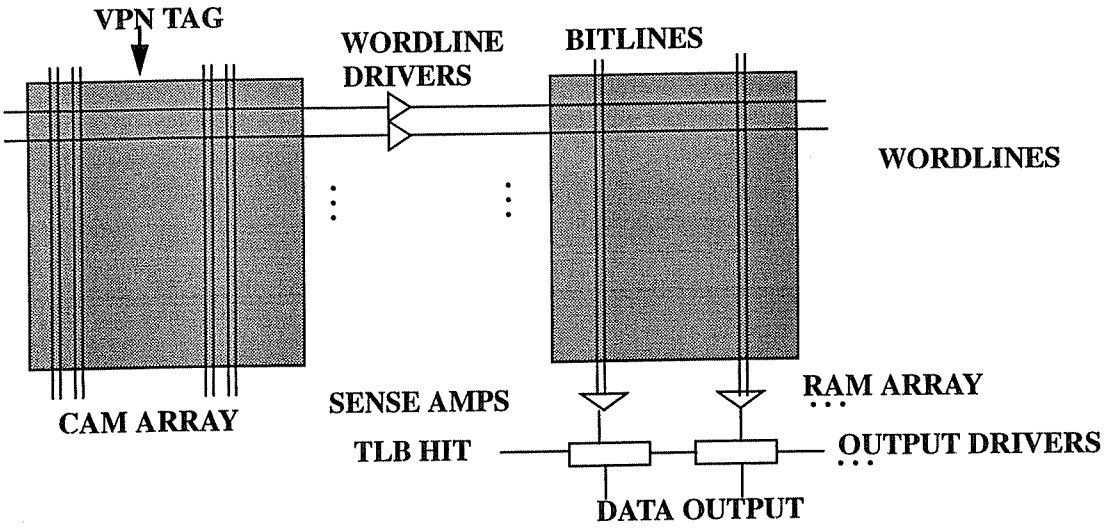
During a TLB lookup, the input VA is split into two parts based on the page size—VPN and Offset. The Offset field, without any translation, appends to the PPN output from the TLB. The page size has to be a power of two to use this bit steering approach. The TLB compares the VPN stored in the tags with the input VPN. Only TLB blocks that contain a valid translation participate in the comparison. The valid bit differentiates between valid and invalid translations. The result of tag comparison selects one word from the RAM as the matching translation and outputs the correct PPN and Attributes (Attr). If no TLB block has a matching tag, the TLB generates a TLB miss signal. Separate control logic generates a protection violation signal if the attributes do not match the intended mode of access.

1.5.1 Fully-associative TLB

In a fully associative TLB (Figure 1-4), the tag array uses a content-addressable-memory (CAM) and the data array uses a randomly-addressable-memory (RAM). Each word, or tag, in the CAM includes a comparator. The tags compare their contents, all in parallel, with the input VPN and signal a match by asserting a match line corresponding to the matching word. The match line, amplified by the wordline driver, selects one data word in the RAM. The bitlines, sense amps, and output drivers output the selected word to the physical address generation and protection check circuits. If none of the tags match, the CAM generates a TLB miss signal as the logical-NOR of all the match lines. Appendix A shows sample VLSI circuits for some components.

Hardware or software must guarantee that only a single tag can match a given VPN. If multiple tags match, more than one RAM word will be enabled onto the bitlines, overloading the circuits. A logical AND of the match signal and the valid bit in each TLB block prevents invalid TLB blocks from generating a spurious match. There are at least three ways to implement valid bits as explained in Appendix A. The first uses a special valid-bit CAM cell to extend the CAM array. The other two store the valid bit in separate storage and combine it with the match signal using either logic or pass gates.

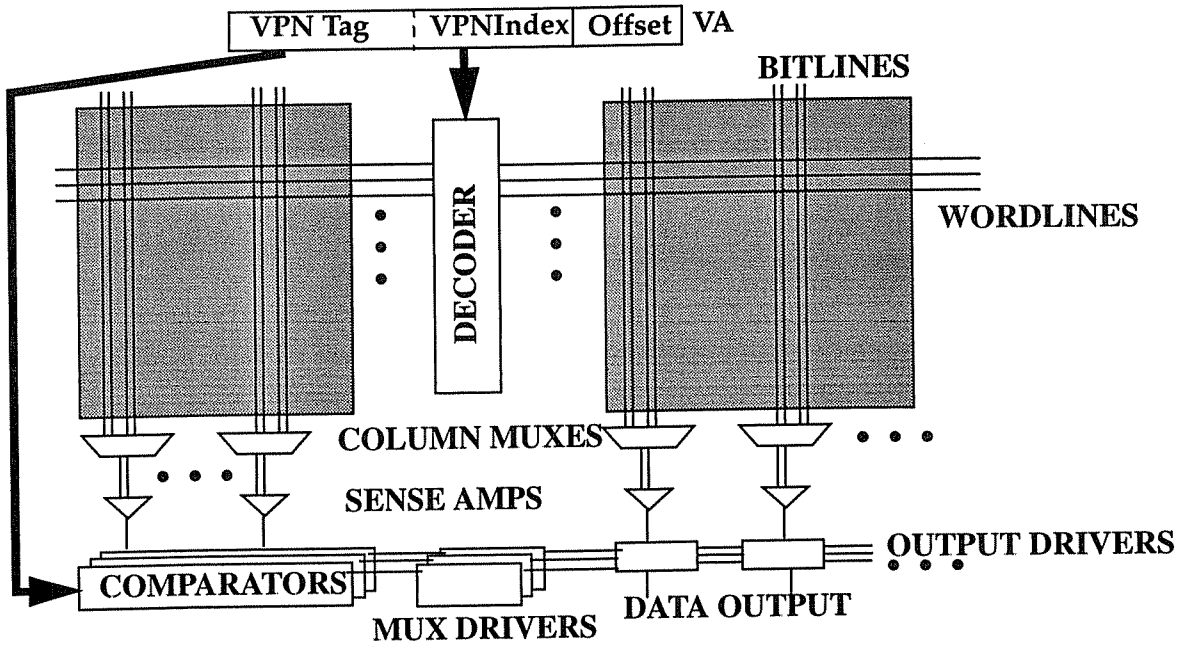
Figure 1-4: Structure of a fully-associative TLB.



1.5.2 *Set-associative TLB*

In a set-associative TLB, as in a set-associative cache [Smit78a], both tag and data arrays use RAMs [Wilt93]. In a typical implementation of an *a*-way set-associative TLB, a single row of the data RAM stores *a* data words and a single row of the tag RAM stores *a* tag words. The low-order bits of the VPN typically[7] index both the RAMs to read out one row from each. Tag comparators compare the *a* tags read out with the high-order bits of the VPN. The output of the tag comparators, amplified by the multiplexor drivers, enables one of *a* output drivers on a TLB hit.

Figure 1-5: Structure of a set-associative TLB



7. TLBs often use a hash function that includes bits from the PID or VPN Tag. Logic not shown in Figure 1-5 combines these bits with the VPN Index bits before being input to the decoder.

As in a fully-associative TLB, the valid bits can be part of the associative compare or can combine with the comparator output as part of the multiplexor driver logic (Appendix A). Direct-mapped TLBs do not require the multiplexors [Hill88] and the data can be used before the tag array access is complete. If tag comparison fails subsequently, the processor can undo the instruction(s) and cause a precise interrupt [Smit88, Wang93]. Figure 1-5 shows column multiplexors associated with the RAMs. Column multiplexors are required in set-associative designs only if the RAM layout has more than *a* words per row—RAM designers often optimize the access time by changing the layout of the array to be roughly square. Bits from the VPNIndex field control the column multiplexors.

1.5.3 TLB miss handling

TLB miss handling, follow-up action on a TLB miss, consists of three different actions. First, locate the PTE corresponding to the faulting address, possibly setting reference and modified bits. Second, choose a victim TLB block to store the PTE. Third, load the PTE into the chosen TLB block. A typical TLB miss handler is as follows:

```
PTE = Find_mapping(VPN);        /* Includes reference/modified bit setting */
load_TLB(PTE, blocki);          /* blocki is replacement victim */
```

The cost of locating the PTE largely determines the TLB miss penalty and involves traversing a page table structure. The traversal can be done by hardware, *e.g.*, SPARC Reference MMU [SPAR91], some PowerPC implementations [May94, Levi95, Beck93], or by software, *e.g.*, MIPS R4x00 [Kane92], UltraSPARC [Yung95], Alpha [Site92]. With software TLB miss handling, there is typically some hardware assist to speed up page table traversal. Some processors generate specialized TLB exceptions and hardware generated page table hint pointers as in MIPS R4x00 [Kane92], UltraSPARC [Yung95], and PowerPC [Ogde95]. Further, software TLB miss handlers incur trap entry/exit costs not shown here. Chapter 7 describes popular page table data structures and their access times.

Many operating systems require the TLB miss handler to set reference and modified bits in the page table. Reference bits are set on TLB misses for loads and stores to a page that has the reference bit clear. Modified bits are set only on TLB misses for stores to a page that has the modified bit clear. A special TLB miss handler typically handles the situation where the TLB has a valid translation and only the modified bit in the page table needs to be set. This mod-bit update does not require TLB replacement but only updates the page table[8]. Mod-bit updates occur frequently as programs often read data from a page before writing to it. Software TLB miss handlers allow operating systems to implement optimizations in setting these bits [DeMo86].

A TLB replacement policy, like a cache replacement policy [Puza85], decides where to place a new translation by choosing a victim TLB block. In a direct-mapped TLB replacement is trivial—there is only a single TLB block that can store the new translation. For set-associative and fully-associative TLBs, hardware or software must implement a replacement policy. The TLB replacement policy impacts TLB performance because non-optimal replacement decisions would cause additional TLB misses. It is impossible to implement the optimal replacement policy (OPT) [Bela66, Matt70, Prie76], and it is impractical to maintain information for

8. Alternatively, the modified bit can be updated in the TLB, postponing the page table update till the next TLB replacement and leaves the page table in a stale state. The operating system often consults page table modified bits to flush dirty pages to disk and must instead use TLB probes to get the correct state.

true LRU (least recently used) replacement policy for large set sizes. Pseudo-LRU algorithms, that approximate LRU with limited information [Kess89, So88, Devi92], are often used. Policies that do not use reference information, *e.g.*, RANDOM [Kane89], are cheaper to implement but incur more TLB misses. Section 2.6 describes the pseudo-LRU replacement policy I use in my TLB simulations.

Finally, the new translation directly overwrites the victim TLB block by writing into the tag and data arrays. TLBs do not require write-backs, as in write-back caches, as the TLB miss handler often writes-through reference and modified bits to the page table. If the PTE format differs from the TLB block format, some transformations are required, *e.g.*, the size field in a superpage PTE may be decoded into a MASK field in the superpage TLB block. Often the TLB miss handler need load only the data part as hardware can infer the tag from the faulting virtual address.

1.6 Roadmap to rest of thesis

Chapter 2 separates out the simulation methodology, metrics, chip area model, access time model, and specific operating system policies used in reporting simulation results in the rest of the thesis. Chapters 3, 4, and 5, describe superpage, complete-subblock, and partial-subblock TLBs. Each chapter includes details of hardware implementation, TLB miss handling techniques, and comparison with other TLB architectures of equal TLB reach or comparable chip area. Appendices A-H include descriptions of alternate implementation ideas, extensions to the base TLB architectures, and handling error conditions. Chapters 3, 4, and 5, report normalized execution time speedup averaged over ten workloads. Appendix I includes execution time speedups for individual workloads. Appendix J shows the absolute number of TLB misses for different TLBs to allow readers to recompute execution time speedups with different assumptions for TLB miss penalty. Chapter 6 includes a discussion on the operating system support required for superpage and partial-subblock TLBs—these TLBs require proper operating system support to be effective. Chapter 7 discusses how conventional page tables may be extended to 64-bit virtual addresses, and to support superpage and subblock TLBs. It also proposes a new page table, *clustered page table*, that applies the same superpage and subblocking ideas used for TLBs to hashed page tables. Chapter 8 concludes reiterating the contributions of my thesis and pointers to future research avenues.

Chapter 2 Methodology

This chapter describes the TLB simulation methodology, metrics, workloads, and operating system support I use to evaluate the performance of single-page-size, superpage, and subblock TLBs. I describe these here so that I can combine performance results along with the description of the new TLB architectures in Chapters 3, 4, and 5, instead of placing the results in a separate chapter after all the TLB descriptions.

I built an operating system *Foxtrot* to evaluate the new TLB architectures. Foxtrot extends Solaris 2.1, a commercial operating system, in two ways. First, Foxtrot includes a TLB simulator that uses trap-driven simulation to simulate a *target TLB*, the TLB under study, different from the hardware TLB. Second, Foxtrot implements operating system policies and mechanisms required to support superpage and partial-subblock TLBs.

Section 2.1 explains the trap-driven simulation technique I use to measure the number of TLB misses, instead of traditional trace-driven simulation techniques. I include a chip area model and an access time model in my study to compare the costs of building different TLBs. The chip area model extends Mulder's model [Muld91] to accommodate superpage and subblock TLBs (Section 2.2) and the access time model extends Wilton and Jouppi's model [Wilt93] (Section 2.3). Section 2.4 describes ten workloads I use throughout the thesis. I use execution time speedup as the performance metric as explained in Section 2.5. Section 2.6 describes the TLB replacement policy I assume. Finally, as superpage and partial-subblock TLBs are ineffective without proper operating system support, Foxtrot implements a default page-size-assignment policy for superpage TLBs and physical memory allocation for partial-subblock TLBs as explained in Section 2.7.

2.1 Trap-Driven Simulation

I use trap-driven simulation to measure the number of TLB misses for single-page-size, superpage, and subblock TLBs. Trap-driven simulation manipulates valid bits in the operating system page table to invoke a TLB simulator on target TLB misses and *never* on target TLB hits. Trap-driven simulation can be faster than trace-driven simulation, as it does not have to process references that are target TLB hits, but cannot measure the number of TLB hits. Wisconsin Wind Tunnel [Rein93] and Tapeworm II [Uhli94] are examples of other systems that use trap-driven simulation for memory system simulations.

Foxtrot manipulates page tables to cause traps into the TLB simulator on target TLB misses as follows: The TLB simulator maintains a data structure corresponding to the target TLB and marks **valid** only those PTEs that correspond to a mapping present in the target TLB. PTEs initialized by the operating system but not resident in the target TLB are marked in a **fake** state—using an unused bit combination in the PTE format. The TLB simulator does not modify PTEs marked **invalid**. The hardware TLB caches a subset of the **valid** mappings. The native TLB miss handler handles hardware TLB misses by traversing the page table without invoking the TLB simulator. The native TLB miss handler causes two types of traps when the processor references a page that causes a target TLB miss. PTEs marked **invalid** result in invoking the operating system page fault handler and PTEs marked **fake** result in invoking the TLB simulator. The TLB simulator loads a mapping for the faulting address into the target TLB—can be a superpage or subblock mapping—and changes the corresponding PTE(s) from **fake** to **valid** state. On target TLB replacement, the simulator changes the PTE(s) corresponding to the vic-

tim TLB block from **valid** to **fake** state. Foxtrot hides these page table changes from the rest of the operating system with wrapper functions for the **read_pte** and **write_pte** routines in Solaris. Foxtrot implements this for superSPARC processors [Blan92] with SPARC Reference MMU [SPAR91] and hardware TLB miss handling. This technique is also applicable for other processors or page table structures. Tapeworm II [Uhli94], for example, implements trap-driven simulation for a MIPS RX000 processor with linear page tables. Further, the technique extends to support multiprocessor TLB simulations—by maintaining per-processor page tables. Foxtrot implements the uniprocessor version only.

Trap-driven simulation for TLBs has three advantages over trace-driven simulation. First, trap-driven simulation runs faster as it incurs overhead only on relatively infrequent target TLB misses. My simulations run three to four orders of magnitude faster than comparable trace-driven simulations. Second, trap-driven simulation naturally handles multiprogrammed workloads. Third, superpage and partial-subblock TLB simulations require dynamically changing operating system information that is hard to encapsulate in a trace, *e.g.*, page-size assignment, physical page numbers and attributes. The simulator has access to such information from the operating system page table. Trace-driven simulation techniques can simulate approximate page-size assignment and physical memory allocation, *e.g.*, as I did for a prior paper [Tall92].

Trap driven simulation has two disadvantages. First, it only calculates the number of target TLB misses and requires other techniques to measure the number of TLB hits, *e.g.*, profiling counters [Site93], external probes [Nagl92]. Second, trap-driven simulation requires separate runs for simulating multiple TLBs. It is possible to use techniques for simultaneous simulation of multiple TLBs [Matt70, Hill89, Kim91] in trap-driven simulators for TLBs that satisfy the inclusion property [Matt70]. Foxtrot does not implement these techniques and I use separate runs for each TLB simulation. The operating system introduces variation in physical memory allocation between multiple runs of a workload. I minimize such variations by flushing the physical memory to a quiescent state before each simulation. In my simulations, variability in the number of TLB misses between multiple runs of the same workload is statistically insignificant (< 1%).

Trap-driven simulation techniques can account for operating system behavior. By locking in the TLB or page table mappings to code and data required for simulation and trap handling, the simulator can simulate TLB or cache misses to addresses outside the locked regions. However, I was unable to isolate the trap handling code and data in Solaris into a few pages. I instead lock nearly all the kernel text and data mappings. Thus, Foxtrot is unable to simulate kernel TLB misses. It simulates only user TLB misses and kernel TLB misses incurred due to data copying during file I/O.

Trap-driven simulation is a very fast simulation technique when the miss ratio is very low [Uhli94, Lebe95], as it often is for TLBs. Other researchers instrument executables and operating systems to perform memory system simulations, *e.g.*, Active Memory [Lebe95], Epoxie [Chen93b, Chen93a], and ATOM [Sriv94]. I do not use this technique as I did not have access to an instrumentation system for Solaris that worked on dynamically linked libraries, operating system references, and supported multiprogramming. Finally, Foxtrot supports simulation of two-level TLBs, with or without multi-level inclusion [Baer88].

2.2 Area Model

The number of TLB misses incurred by a TLB can often be made arbitrarily small by increasing the number of TLB blocks to map the complete working set of a workload. However, most microprocessor designs include a TLB on-chip and chip area constraints limit the design options available to processor designers [Joup94, Nagl94a]. In later chapters, I show the effectiveness of the new TLB architectures by comparing execution time speedups for different TLBs that occupy comparable chip area. In this section, I describe the area model I use to estimate the chip area cost of a TLB.

The primary component of a TLB is the data path—tag and data arrays, drivers, multiplexors, and sense amps. I estimate the area cost using the model proposed by Mulder *et al.* [Muld91] for fully-associative and set-associative caches. The model calculates the area in units of *register bit equivalents* (**rbe**)—the number of register cells that can be implemented in the same area. Figures 2-1 and 2-2 illustrate how the model estimates chip area for TLBs. The formulae for fully-associative ($Area_{fac}$) and set-associative TLBs ($Area_{sac}$) are as follows, where #blocks is the number of TLB blocks:

$$Area_{fac} = PLA + RAM + CAM = 130 + 0.6 * (\#blocks + 6) * ((\#data\ bits + \#status\ bits) + 6) + 0.6 * (\sqrt{2} * \#blocks + 6) * (\sqrt{2} * \#tag\ bits + 6)$$

$$Area_{sac} = PLA + Data\text{-}RAM + Tag\text{-}RAM = 130 + 0.6 * (\#sets + 6) * ((\#data\ bits * associativity) + 6) + 0.6 * (\#sets + 12) * (((\#tag\ bits + \#status\ bits - \lfloor lg_2(\#sets) \rfloor) * associativity) + 6)$$

Figure 2-1: Fully-associative TLB area model assumptions

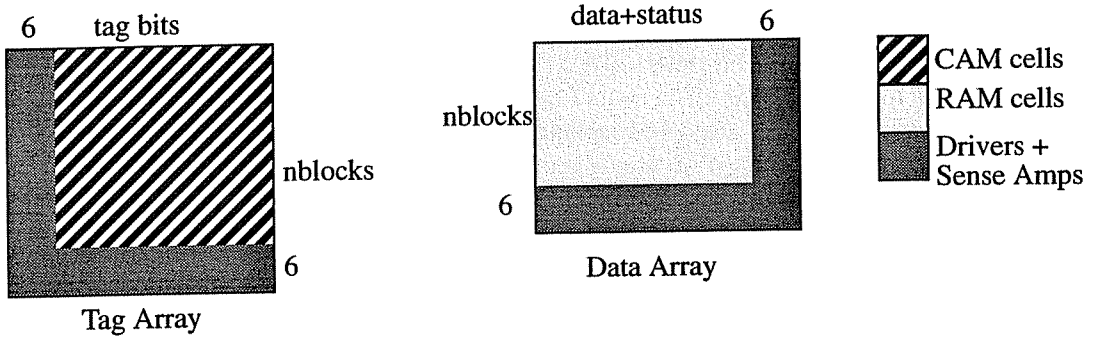
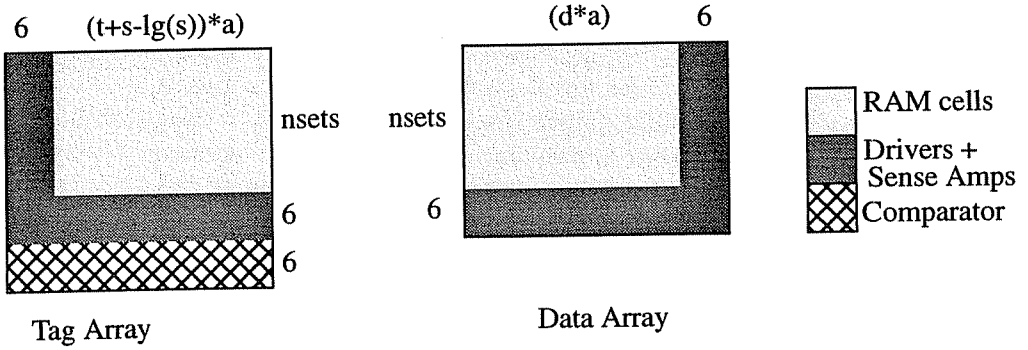


Figure 2-2: Set-associative TLB area model assumptions



I use the exact same estimates as in Mulder's model. The original model assumes that drivers, precharge circuitry, and sense amps for each array have an overhead equal to six bits wide

and six blocks high—solid areas in Figures 2-1 and 2-2. It assumes that set-associative TLB tag comparators occupy an area comparable to six tags—checked area in Figure 2-2. The model assumes square shapes for both RAM and CAM cells. In practice, fully-associative TLB implementations in custom VLSI match the pitches of the CAM and RAM arrays. This results in rectangular shaped cells that are different from the square cell assumption in the model. The model assumes that a RAM cell is 0.6 rbe and that a CAM cell is twice as large as a RAM cell. The model assumes overhead for control logic and decoders to be a constant 130. Besides the assumptions in the original model, I make the following assumptions about the number of bits in a TLB block:

#status bits is the number of status bits per TLB block. I assume one status bit per TLB block—the **used** bit (for pseudo-LRU replacement).

#data bits is the number of bits in the data part of a TLB block. I assume that the data bits include a 36-bit PPN (48-bit physical address - 12-bit base page offset) and nine attribute bits including one modified bit. A complete-subblock TLB has s, the subblock factor, times as many data bits. A superpage TLB block adds a SZ field that is $\lg_2$(number of supported page sizes) bits. A partial-subblock TLB block adds a one-bit SB attribute.

#tag bits is the number of bits in the tag part of a TLB block and is also the width of the tag comparator. The tag bits include a 12-bit Context ID, a 52-bit VPN (64-bit virtual address - 12-bit base page offset) and one or more valid bits[1]. There are $\log_2(s)$ fewer tag bits smaller in subblock TLBs as they store only the VPBN. In superpage TLBs $\log_2(s)$ bits of the VPN, the MASK field, are don't care bits that I model as two tag bits each. Single-page-size, superpage, and complete-subblock TLB blocks have one valid bit and partial-subblock TLB blocks have s valid bits in the tag.

This analytical model allows a simple back-of-the-envelope estimate of the chip area for various TLB parameters. In real implementations, the formulae must be adjusted for many VLSI process and implementation dependent parameters. Many optimizations I do not consider here also affect the accuracy of this model. The size of drivers is usually a function of the number of bits or blocks—the model assumes them to be of constant size; don't care bits may be smaller than two CAM cells; large TLB implementations are split into multiple arrays with separate drivers, decoders and multiplexors—the model assumes a single monolithic array. The original model gave estimates that were comparable to caches built around the time the paper was published in the early 1990s. I have not adjusted or validated the model for many VLSI process changes that have since occurred.

2.3 Access Time Model

TLB access time is also an important metric and design constraint as the TLB often lies on the processor critical path. Recent VLSI technology trends have increased the importance of access time relative to chip area. Chip designers have an increasing number of transistors or chip area available to them but cannot build large first-level caches or TLBs due to access time constraints. In real implementations circuit analysis tools, such as spice, predict TLB access time accurately. Due to lack of detailed circuit implementations and the large number of TLB configurations studied, I use an analytical model proposed by Wilton and Jouppi [Wilt93], which

1. The original model assumes tag valid bits to occupy a smaller area than CAM cells—($\sqrt{2}$ x 1) vs. ($\sqrt{2}$ x $\sqrt{2}$) times a RAM cell. I assume valid bits to have the same area as a CAM cell as I use CAM cells in partial-subblock TLBs (Section 5.1.2). In practice, valid bits are smaller and my model overestimates the area.

is an extension of the analytical model proposed by Wada *et al.* [Wada92]. The model derives simple equations that predict access and cycle times as a function of various cache and VLSI process parameters. I extend the model in several ways:

- I assume the number of tag and data bits as described in Section 2.2 for a 64-bit address space. The original model assumes that tag bits are significantly fewer than data bits and does not optimize the tag array drivers. TLBs for 64-bit address spaces may have more tag bits than data bits. I size the tag array drivers, in set-associative TLBs, using the same algorithm used in the original model to size the data array drivers.

- I model valid bits in the tag comparator. Appendix A includes alternate implementations for valid bits for single-page-size and superpage TLBs. Appendix B explains alternate ways to include multiple subblock-valid bits in subblock TLBs. I model the least efficient way to include subblock-valid bits in partial-subblock TLBs. Faster implementations are possible. I also model superpage TLBs to include don't care bits in the tag.

- I use a simpler multiplexor driver in set-associative TLBs if the number of data bits read out of the data array is same as the number of output bits. I use a 2-stage driver instead of a 3-stage driver used in the original model.

- I extended the data RAM model to support complete-subblock TLBs—with and without the use of column multiplexors as described in Appendix C. A complete-subblock TLB fits in the original model as a cache supporting word reads smaller than the block size.

- The original model does not support CAM arrays. I developed a model for a CAM array using the set-associative comparator and wordline models. I assume that the bitlines of a CAM array behave as wordlines during tag comparison. I also assume that the set-associative comparator models a CAM word comparator.

- I assume the use of a single RAM or CAM array in a rectangular shape. Other organizations may be faster to access. Wilton and Jouppi's model, for example, assumes that the arrays approximate a square shape. Large arrays may be faster to access when split into multiple smaller arrays and their results combined.

In a real implementation, many circuit optimizations are possible that can improve the access time by optimizing the critical path. Therefore, the absolute value of the access time is not significant and I present relative access times between the different TLB organizations throughout the thesis. There are some caveats for interpreting these results:

- The CAM model is a first order approximation and the access time for a fully-associative TLB from this model is not comparable to the access time for a set-associative TLB.

- Cycle times are also an important consideration in TLB design. Cycle time differs from access time by the time required to precharge various circuits. I use access time instead of cycle time as circuit designers have considerable leeway in varying the precharge times by proper sizing of precharge transistors. Note that changing the size of precharge transistors changes capacitances and will affect the access time calculated here.

- The model does not include in fully-associative TLBs an output multiplexor needed to implement MMU bypass mode or superpage and partial-subblock TLB physical address generation.

- Most importantly, the model has not been validated against any spice simulations. Many parts of the original model were validated [Wilt93] for greater than 100 rows. I, however, use the same RAM model even when the number of rows is less than 100.

The relative access time between two TLB configurations is an indication of the cost of implementing a TLB—another cost is the chip area calculated as in Section 2.2. The costs raise interesting questions about the relative merits of the different TLBs and any comparison using only number of TLB misses or execution time speedups would be incomplete.

2.4 Workloads

I use ten 32-bit single-user workloads from different application areas—scientific, integer, database, and functional programming—to evaluate the performance of various TLB implementations. I first describe the workloads and then discuss the consequences of concentrating on these workloads. **Nasa7, compress, wave5, spice,** and **gcc** are from the SPEC92 suite [SPEC91]; **fftpde** is a NAS benchmark [Bail91] operating on a 64X64X64 matrix; **mp3d** and **pthor** are uniprocessor versions from the SPLASH benchmark suite [Sing92]; **coral** [Rama93] is a deductive database executing a nested loop join; **ML** [Appe91b] is executing a stress test on the garbage collector [Repp94]. I use a Sun SPARCstation with a 40 MHz SuperSPARC processor for all the simulations.

Table 2-1: Workload characteristics (40MHz SuperSPARC processor)

Workload	total time (seconds)	user time (seconds)	#user TLB misses for SuperSPARC TLB (thousands)	% user time in TLB miss handling	#(user+kernel) cache misses SuperSPARC Ecache (thousands)	Peak Memory Usage (MB)
coral	177	172	85974	50%	71053	19.9
nasa7	387	385	152357	40%	64213	3.5
compress	99	77	21347	28%	21567	1.4
fftpde	55	53	11280	21%	14472	14.7
wave5	110	107	14510	14%	4583	14.3
mp3d	37	36	4050	11%	5457	4.8
spice	620	617	41923	7%	81949	3.6
pthor	48	35	2580	7%	6957	15.4
ML	945	917	38423	4%	314137	32.0
gcc	118	105	2440	2%	9980	5.6

Table 2-1 displays workload data, with the workloads sorted from most to least percent of user time spent on TLB miss handling for a SuperSPARC processor. Columns two and three give total and user execution time, showing that these workloads spend most of their time in user mode. This is important as my simulations do not account for operating system TLB misses. Columns four and five give the number of user TLB misses (for the SuperSPARC processor's TLB) and the percent of user time spent servicing these misses (assuming a forty cycle TLB miss penalty). I estimate the number of TLB misses by using Foxtrot to simulate SuperSPARC's TLB—64-block fully-associative single-page-size (4KB) TLB using the TLB replacement algorithm described in Section 2.6. Column six shows the number of cache misses (includes user and kernel cache misses) incurred by the SuperSPARC's level-two cache—1MB direct-mapped cache with 32-byte cache lines. I measure this using hardware counters in the cache controller. Note that some of these workloads incur more TLB misses than cache misses.

TLB misses may be more important than cache misses, because, in many systems, TLB miss penalty is larger than cache miss penalty. Finally, column seven displays the peak memory usage of each workload in megabytes.

I intentionally selected workloads that spend significant time in TLB miss handling. While many programs have negligible TLB miss ratios and do not benefit from TLB performance improvements of the new TLB architectures, my results are still relevant for two reasons.

First, today's microprocessors are used in a variety of computers, from laptops to multiprocessor servers. A TLB that can handle larger workloads makes a microprocessor viable for use in larger systems, even if it does not help average performance in smaller ones. The new TLB architectures reduce the number of TLB misses for even the small workloads, though the improvement in execution time is negligible.

Second, future workloads may place a greater pressure on TLBs than today. I use 32-bit workloads only. I expect future 64-bit and object-oriented programs to have even larger and sparser address spaces and spend more time in TLB miss handling. Such workloads would make TLB and page table effects more important. For example, Mogul *et al* report that modifying some 32-bit programs to use 64-bit pointers increases address space usage about 30% [Mogu95]. I am not aware of any published results about the effect on time spent in TLB miss handling.

By emphasizing on workloads for which TLB miss handling time is important, however, my results overestimate the potential speedup for workloads that include processes with small address spaces. Thus my results should be interpreted as a measure of the benefit for large workloads from the new TLB architectures.

There are two other limitations of my workloads—multiprogramming and working set size. I do not study the effects of multiprogramming several large programs. Multiprogramming can increase the number of TLB misses and make TLB miss handling more significant [Agar88]. Another effect of multiprogramming is the use of more physical memory. This can affect the proper placement of pages in physical memory for superpage and partial-subblock TLBs (Chapter 6). In chapters 3 and 5, I include TLB performance numbers with and without proper memory allocation support in the operating system for the new TLBs. I do not account for virtual memory paging in these simulations as I execute them on an otherwise idle system with 96MB of physical memory. Of my workloads, only **compress** and **gcc** are multiprogrammed.

Further, while my workloads are large enough to stress 64-block fully-associative TLBs, the working sets of many of them fit in 256 or 512-block TLBs and incur only compulsory misses (Appendix J). I include comparisons between large TLBs to emphasize the access time and chip area advantages of the new TLB architectures, though only some of my workloads exercise them, *e.g.*, **coral** and **ML**.

2.5 TLB Performance Metric

I use execution time of the workload as a measure of performance. I compare the performance effect of a TLB, TLB_{new}, relative to a base TLB, TLB_{base}, using execution time speedup as the metric. As I do not have hardware TLB implementations of the various TLB configurations, I could not measure execution time on a real system. I instead estimate the execution time for each workload and TLB configuration using the number of TLB misses from my TLB

simulator and a TLB miss penalty estimate, as explained below:

Speedup for workload $i = \frac{Time(i, baseTLB)}{Time(i, newTLB)}$, where *Time(i, T)* is the execution time for workload *i* and TLB configuration *T*.

$Time(i, T) = Tideal(i) + (TLBmisses(i, T) \times TLBmisspenalty)$, where *Tideal(i)* is the execution time for workload *i* if it spent zero time in TLB miss handling.

$Tideal(i) = RunTime(i, SPARCstation) - (TLBmisses(i, SuperSPARCTLB) \times TLBmisspenalty)$, where *RunTime(i, SPARCstation)* is the wall clock time to run workload *i* on the SPARCstation I use for my thesis. I then measure the number of TLB misses using my TLB simulator to simulate the processor (SuperSPARC) TLB configuration and replacement algorithm.

I further assume a constant TLB miss penalty of 40 cycles (on a 40 MHz processor) for all TLBs. I could not measure the TLB miss penalty on a SuperSPARC processor as the TLB miss handling happens in hardware. Further, the TLB miss penalty can vary from 2 to 80 cycles depending on the number of cache hits on page table accesses. I also assume that the new TLB architectures incur the same TLB miss penalty as a single-page-size system. In Chapter 7, I describe page table strategies for which this is true and some for which the TLB miss penalty can be higher.

In the main text of the thesis I only present normalized speedups, which allows me to compare the performance of two TLBs with a single number. Appendix I shows the individual workload speedups for more detailed study. Normalized speedup $= \frac{WTime(baseTLB)}{WTime(newTLB)}$, where *WTime(T)* is the normalized workload execution time. $WTIME(T) = \sum_{i = workloads} W(i) \times Time(i, T)$, the weighted arithmetic average of the individual workload execution times. The weights normalize execution times such that each workload runs for the same amount of time with an ideal TLB. $Weight(i) = \frac{\sum_{i = workloads} Tideal(j)}{Tideal(i)}$; $W(i) = \frac{Weight(i)}{\sum_{j = workloads} Weight(j)}$

Table 2-2 shows the calculation of *Tideal* for each workload. The second and third columns show *RunTime(i, SPARCstation)* and *TLBMisses(i, SuperSPARCTLB)* from Table 2-1. The fourth column shows the TLB miss penalty estimate—40 cycles or one microsec at 40MHz. The fifth column derives the time spent in TLB miss handling and the sixth column shows *Tideal* for each workload. Weights for calculating normalized speedup can be derived from *Tideal* using the above formulae—column seven shows the weights used in rest of the thesis. The last column shows the maximum speedup possible for each workload relative to a 64-block fully-associative single-page-size (4KB) TLB. Note that many tables in Appendix I list speedups that are relative to TLBs different from a 64-block fully-associative single-page-size (4KB) TLB and may differ significantly from the maximum speedup shown in Table 2-2.

The calculation of execution times, speedups, and weights are highly sensitive to the TLB miss penalty estimate. I illustrate this sensitivity in Table 2-3, where I compare normalized speedups relative to a 64-block fully-associative single-page-size (4KB) TLB for four fully-associative TLBs that occupy comparable chip area—128-block single-page-size (4KB) TLB, 123-block superpage TLB that supports 4KB and 32KB pages using the policy described in Section 2.7.1, 114-block partial-subblock TLB with subblock factor of 16, and a 72-block com-

Table 2-2: Parameters used to calculate normalized speedup

Workload	RunTime for SuperSPARC (seconds)	#TLB misses SuperSPARC TLB (thousands)	TLB miss penalty (cycles) 1cycle = 25ns	TLB miss handling time (seconds)	Tideal(i) (seconds)	Weight (W(i))	Maximum Speedup relative to SuperSPARC TLB
	(a)	(b)	(c)	(d) (b) * (c) * 25	(e) (a) - (d)	(f) function(e)	(g) (a)/(e)
coral	177	85974	40	85.97	91.03	0.088	1.94
nasa7	387	152357	40	152.36	234.64	0.034	1.65
compress	99	21347	40	21.35	77.65	0.103	1.27
fftpde	55	11280	40	11.28	43.72	0.183	1.26
wave5	110	14510	40	14.51	95.49	0.084	1.15
mp3d	37	4050	40	4.05	32.95	0.242	1.12
spice	620	41923	40	41.92	578.08	0.014	1.07
pthor	48	2580	40	2.80	45.42	0.176	1.06
ML	945	38423	40	38.42	906.58	0.009	1.04
gcc	118	2440	40	2.44	115.56	0.069	1.02

plete-subblock TLB with subblock factor 4. Tables I2-3a to I2-3c in Appendix I show individual benchmark speedups. I consider TLB miss penalties of 30, 40, and 50 cycles. A larger TLB miss penalty makes the speedup more significant for these workloads.

Table 2-3: Sensitivity to TLB miss penalty—execution time speedup for alternate fully-associative TLBs relative to 64-block fully-associative single-page-size (4KB) TLB

TLB miss penalty	128-block single-page-size (4KB) TLB	123-block superpage (4KB/32KB) TLB	114-block partial-subblock TLB (subblock factor 16)	72-block complete-subblock TLB (subblock factor 4)
30	1.045	1.132	1.161	1.072
40	1.061	1.185	1.227	1.098
50	1.078	1.242	1.301	1.125

I include enough data in the thesis for readers to recalculate execution time for any workload or TLB configuration with different TLB miss penalty assumptions using formulae described earlier in this section. Calculating the execution time for a workload requires *Tideal(i)* and *TLB-Misses(i, TLB)*. *Tideal(i)* can be calculated by redoing the calculations in Table 2-2. Tables in Appendix J show *TLBMisses(i, TLB)* for all the TLB configurations I describe in the thesis. Table 2-4 shows the effect of varying the TLB miss penalty from 30 to 50 cycles on Tideal and the weights.

I use a normalized speedup with weights that treat all ten workloads with equal importance. Readers can choose different weights for the workloads if the desired workload mix is different from my assumptions.

2.6 TLB Replacement Policy

All my TLB simulations use a pseudo-LRU replacement algorithm—Go-Down-Stack (GODS) policy [Devi92]—that some commercial processors also use, *e.g.*, UltraSPARC [Yung94]. Each TLB block includes one extra bit, the *used* bit that is set on TLB hits. The algo-

Table 2-4: Effect of TLB miss penalty on Tideal and Weights

Workload	Tideal (seconds)			Weight (W(i))		
	30	40	50	30	40	50
coral	112.52	91.03	69.53	0.075	0.088	0.107
nasa7	272.73	234.64	196.55	0.031	0.034	0.038
compress	82.99	77.65	72.32	0.101	0.103	0.103
fftpde	46.54	43.72	40.90	0.181	0.183	0.183
wave5	99.12	95.49	91.86	0.085	0.084	0.081
mp3d	33.96	32.95	31.94	0.248	0.242	0.234
spice	588.56	578.08	567.60	0.014	0.014	0.013
pthor	46.06	45.42	44.77	0.183	0.176	0.167
ML	916.18	906.58	896.97	0.009	0.009	0.008
gcc	116.17	115.56	114.95	0.073	0.069	0.065

rithm for choosing a victim block for replacement on a TLB miss is as follows—assume that TLB blocks in a TLB set are numbered starting from 0[2]:

1) If there are any invalid TLB blocks, choose the lowest numbered invalid TLB block. 2) If there are no invalid TLB blocks, choose the lowest numbered TLB block with the used bit clear. 3) If there are no unused TLB blocks, clear all the used bits and restart the algorithm—chooses TLB block 0.

A 2*n-input priority encoder easily implements this algorithm. However, setting of used bits on TLB hits may affect the TLB access critical path. Foxtrot manipulates hardware-maintained referenced bits in the page table to simulate used bits in an exact fashion.

In the rest of the thesis, I only include results using the above replacement policy. Simulation results show that using this replacement policy often results in fewer TLB misses than others. For illustration, I consider three alternate TLB replacement policies. Clock [East79] implements a second-chance replacement algorithm often used in operating system page replacement, with the additional optimization that invalid TLB blocks are replaced first. Random [Kane89] replaces an arbitrary TLB block that may or may not be valid. FIFO implements a straightforward first-in-first-out algorithm. Table 2-5 shows the sensitivity to replacement policy for a 64-block fully-associative single-page-size TLB. For these workloads, the Clock replacement policy performs comparable to the Go-down-stack policy, but is more complicated to implement. Random and FIFO are simpler to implement but have slightly worse performance as they do not account for reference history. However, the results are not uniform. Table I2-5 in Appendix I shows that **fftpde**, for example, incurs fewer TLB misses with a Random replacement policy than with pseudo-LRU replacement.

Table 2-5: Sensitivity to TLB replacement policy—execution time speedups relative to 64-block fully-associative single-page-size (4KB) TLB using Go-down-stack (GODS) replacement policy

GODS	Clock	Random	FIFO
1.00	1.00	0.98	0.98

2. A few TLB blocks are often reserved for special operating system code that needs to execute without incurring TLB misses. I assume that the replacement algorithm skips these special blocks.

2.7 OS Support for superpage and subblock TLBs

Superpage and partial-subblock TLBs are largely ineffective without proper operating system support. For the TLB simulations in chapters 3-5, I fix a single set of operating system mechanisms and policies—Chapter 6 discusses alternate mechanisms and policies. For superpage TLBs, Foxtrot implements a page-size assignment policy that decides when to use superpages and when to use base pages. For partial-subblock TLBs, Foxtrot includes mechanisms to properly place (Table 1-1) base pages in physical memory.

2.7.1 Description of superpage page-size assignment policy

Foxtrot implements a page-size assignment policy based on the working set threshold (Section 6.1)—it uses superpages when the number of pages referenced within a page block crosses a threshold. The thresholds I use are: 50% for disk file pages, 75% for network file pages, 100% for heap pages. I chose the thresholds based on the likelihood of finding unreferenced base pages within a page block already in physical memory. This depends on the prefetching policy in the file system and virtual memory system of Solaris. I have not studied the effect of varying the thresholds.

To implement this policy, Foxtrot includes some new operating system mechanisms. The page fault handler maintains counters for each virtual page block to keep track of the number of pages referenced within each page block. A page promotion mechanism changes the page size when the counters cross the predetermined thresholds.

A naive implementation of page promotion would require base pages to be copied into a physical page block before using superpages. Foxtrot, instead, implements page reservation (Section 6.2.5) to allocate pages at the "proper" place in the first place and avoid copies completely. The policy is as follows:

On the first page fault to a page block, Foxtrot reserves a physical page block for the virtual page block. After I/O for the faulting base page completes, Foxtrot stores a base page mapping in the page table. On subsequent page faults to other base pages within the page block, Foxtrot fetches the data from backing store into the prereserved physical pages and loads base page mappings in the page table. When the number of page faults to a page block crosses the page promotion threshold, Foxtrot promotes the page block to use a superpage mapping. Page promotion involves unloading the base page mappings from the page table, fetching from backing store into the prereserved pages any base pages not present in memory, and loading a superpage mapping into the page table.

Foxtrot reduces page promotion costs by prefetching into memory, in the background, unreferenced base pages within a page block. This reduces the wait time to fetch extra pages during page promotion. Solaris, by default, prefetches the next eight base pages to the faulting base page. Foxtrot modifies the prefetching policy to *read-around* the faulting base page, *i.e*, prefetches pages that both precede and follow the faulting base page but limits the prefetch to within a single page block. Foxtrot prefetches a full page block for **ufs** files, the following eight base pages that belong to the same page block for **nfs** files, and does not prefetch (or preinitialize) heap pages.

Page promotion, reservation, or prefetching do not occur for segments that do not span a full page block, *e.g.*, segment size smaller than the page block size, the first (or last) page block in a segment that starts (or ends) at an unaligned virtual address, or when base pages within a

page block do not have the same attributes.

2.7.2 *Physical Memory Allocation for Partial-subblock TLBs*

Partial-subblock TLBs do not require the operating system to implement a page-size assignment policy but do require physical memory allocation to properly place base pages in physical memory. Foxtrot uses page reservation, described in Section 6.2.5, to allocate physical memory such that base pages are often properly placed.

Chapter 3 Superpage TLBs

This chapter evaluates the use of *superpages* [Tall92, Mogu93, Tall94a] to increase TLB reach. *Superpages* use the same linear address space as conventional paging, have sizes that must be power-of-two multiples of the *base page size*, and must be aligned in both virtual and physical memory [Tall94a]. Superpages, however, are not universally useful and there is an inherent tradeoff in using superpages [Tall92]. Superpages decrease the number of TLB misses but increase memory demand[1] due to internal fragmentation. Thus, the key to using superpage TLBs is an operating system that uses superpages where appropriate and base pages elsewhere. Nearly every current microprocessor supports superpages, *e.g.*, MIPS [Kane92], UltraSPARC [Yung95], Alpha [Bann95], PowerPC [Silh93], HP-PA RISC [Hunt95]. The MIPS R4000 [Kane92], for example, supports a 4KB base page size and superpages of 16KB, 64KB, 256KB, 1MB, 4MB, and 16MB. However, I am not aware of any operating system that uses superpages in a general manner. I discuss the operating system issues in Chapter 6.

This chapter studies the issues involved in building both fully-associative and set-associative TLBs that support superpages, discusses how to handle TLB misses for superpage TLBs, and compares the TLB performance with alternate single-page-size TLBs.

A superpage TLB block's tag maps variable-sized page blocks and the data stores a single mapping. Mappings to base virtual pages within a page block can share a single superpage TLB block if *all* are mapped, *all* are present in physical memory, *all* are properly placed in physical memory, *all* have the same attributes, the operating system has recognized these base pages, and promoted them. The hardware complexity to add superpage support to fully-associative TLBs is small but requires significant operating system support to use superpages.

Figure 3-1: Superpage TLB block

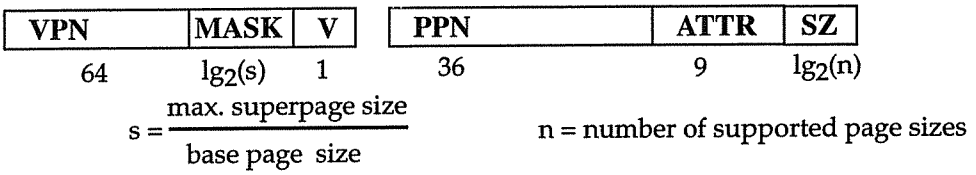


Figure 3-1 shows the format of a superpage TLB block, which adds a size field to both the tag (MASK) and data (SZ) portions of a single-page-size TLB block. The MASK field prevents certain tag bits from participating in tag comparison for superpage mappings and the SZ attribute controls a multiplexor during physical address generation. A fully-associative TLB can simultaneously support multiple superpage sizes, but set-associative TLBs do not efficiently support more than one page size as explained in Section 3.2.2.

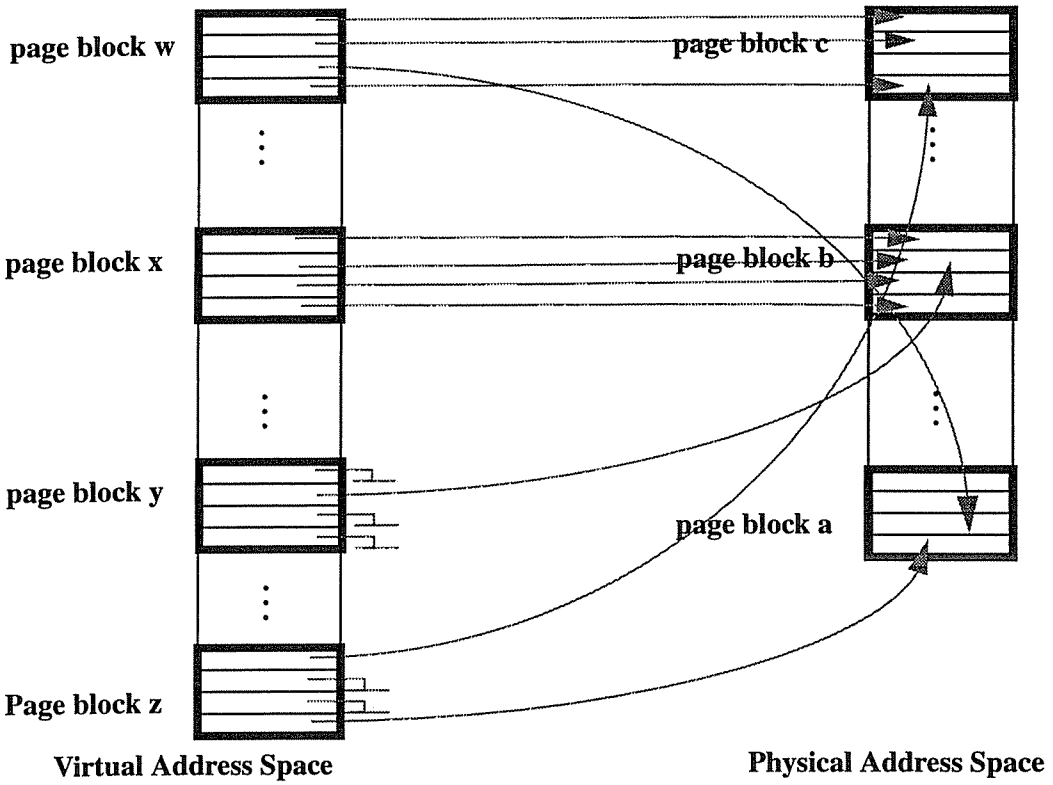
Figure 3-2 shows some mappings from a virtual address space to a physical address space and how they can be stored in an 8-block fully-associative superpage TLB that supports 4KB and 16KB pages. Only the mappings for page block **x** can use a superpage mapping—page block **w** has one base page not properly placed in memory with respect to other base pages, page block **y** has some unmapped base pages, page block **z** has both improperly placed and unmapped base pages.

1. On today's large physical memory machines. increased memory demand may not be a concern, but initialization overhead—zeroing or doing I/Os—for the extra memory used can increase execution time.

Large superpage sizes (≥256B) can improve TLB performance significantly and are clearly useful in certain applications, *e.g.*, kernel text/data, frame buffers, and database buffer caches. I assume, and strongly recommend, use of large superpages in such situations. If there are usually only a few large superpages in use, it may be possible to setup their mappings with limited changes to existing operating systems.

Medium-sized superpages (< 128KB) are more appropriate for general use, as most objects mapped in an address space are not large enough to use large superpage sizes and the cost of making a wrong decision in choosing a medium-sized superpage is small. The cost of a 64KB disk sequential I/O, for example, is not that different from that of a 4KB I/O. However, medium-sized superpages require more substantial operating system support to provide policies for choosing appropriate page sizes and the mechanisms to support them.

Figure 3-2: Virtual Address to Physical Address mappings in a superpage system



xXX	11	✔
w00	00	✔
w02	00	✔
w03	00	✔
w01	00	✔
y02	00	✔
z03	00	✔
z00	00	✔

VPN MASK V

bXX	Attr	1
c00	Attr	0
c02	Attr	0
c03	Attr	0
a01	Attr	0
b02	Attr	0
c01	Attr	0
a00	Attr	0

PPN ATTR SZ

Superpage TLB storing mappings for above

Throughout this thesis, I assume superpage TLBs that support two page sizes—a base page size of 4KB and a medium-sized superpage size that is a power of two multiple of 4KB. I also assume that the operating system supports the generic use of superpages with a dynamic page-size assignment policy (as explained in Section 2.7.1) that chooses between these two page sizes. Section 3.1 discusses other way of using superpage TLBs and the reasons why I restrict my thesis to this alternative.

Section 3.2.1 shows how fully-associative single-page-size TLBs include superpage support with little overhead. Set-associative superpage TLBs are harder to build as it is not clear which bits from the virtual address must index the TLB. Section 3.2.2 suggests three indexing schemes—base page index, superpage index, and exact index. The base page index does not reduce TLB misses, the superpage index has unacceptable performance if the operating system does not use superpages, and the exact index is costly to implement as the page size corresponding to a virtual address is unknown when starting the lookup. Further, neither is practical when simultaneously supporting more than two page sizes. Simulation results show that set-associative superpage TLBs can be effective at reducing the number of TLB misses. However, if the operating system does not use superpages, and uses base pages only, a set-associative superpage-index TLB incurs significantly higher TLB misses than a single-page-size TLB with the same number of blocks and associativity.

While the number of TLB misses reduce with use of superpages, other system overheads may increase and offset some gains. The system overheads include page fault service time, paging traffic, memory demand, TLB miss penalty, and execution of page-size assignment policy. With efficient implementation of the superpage mechanisms, policies (Section 6.1), and page tables (Section 7.4), it is possible to minimize these overheads such that the reduction in number of TLB misses also reduces overall execution time. Section 3.3, in particular, shows how to handle superpage TLB misses without increasing the TLB miss penalty over a single-page-size system.

Section 3.4 compares the performance of superpage TLBs with set-associative single-page-size TLBs of comparable chip area. For the workloads I consider, results show that the superpage TLBs, though they have fewer TLB blocks, incur fewer TLB misses if the operating system uses superpages. The superpage TLBs, however, incur more TLB misses if superpages are not used. Section 3.5 reiterates the conclusions.

3.1 Superpage TLB and Operating System taxonomy

Before discussing how to build superpage TLBs and evaluating their performance, I first discuss the different varieties of superpage TLBs used in different processors and how operating systems use them. TLBs that support multiple page sizes can typically support one or more of three different features:

a) per process/system configurable page size: The TLB supports a single page size, but the page size can be changed either during system initialization or process switch. Examples of this category include Motorola 68040, which has a mode bit to select between 4KB and 8KB page size [Eden90], Motorola 68020 using MC68851 controller can select a page size from 256 bytes to 2KB [Moto86], the SGI R8000 processor allows two page sizes—one for instructions and another for data—that are selectable per-process [MIPS93], AMD29000 [John87] supports a per-process page-size.

Many single-page-size operating systems use a configurable system-wide PAGESIZE constant. Solaris, for example, uses 8KB pages on V7 or V9 machines and 4KB pages for V8 machines; IRIX uses 4KB pages for R4X00 machines and 16KB pages for R6000 machines. Superpage TLBs allow such operating systems to choose different page sizes for different machines, *e.g.*, server machines could use 64KB pages while desktop workstations use 4KB pages. This is an important use for TLBs that support multiple page sizes. I am not aware of any operating system that supports varying per-process page sizes on a single system. I do not explore this option further, as the version of the Solaris operating system I use does not support changing the default PAGESIZE.

b) separate superpage TLB: The main TLB supports the base page size and a separate TLB supports the large superpages. This allows the operating system to use large superpages for special cases such as kernel text, database buffer caches, and frame buffers. Examples of this category include HP-PA RISC [Hewl93, Hunt95], Motorola 88x00 [Mile90], Intel i860XP [Inte91], and PowerPC [Silh93, May94]. Such a superpage TLB is also known as a Block TLB or a Block Address Translation Cache (BATC). Most implementations of separate superpage TLBs require special TLB miss handling for superpages as the default TLB miss handler does not handle superpage mappings. I do not explore this option further as a limited number of superpage TLB blocks largely restricts superpage usage to a few restricted situations and not applicable for generic user programs.

c) multiple-page-size TLB: The TLB can simultaneously store mappings of many different page sizes. These TLBs are usually fully-associative due to the difficulty of building set-associative TLBs that support multiple page sizes (Section 3.2.2). For example, MIPS R4x000 supports seven page sizes from 4KB to 16MB [Kane92], UltraSPARC [Yung95] and Alpha [Bann95] support four pages sizes of 8KB, 64KB, 512KB, and 4MB. Others include ETA-10 [ETA 86], ARM6 [Adva93], and SPARC Reference MMU [SPAR91]. Many also include a default TLB miss handler (in hardware or software) that can load superpage mappings in the TLB.

This is more useful than supporting a single configurable page size as programs have a mix of mappings that cannot all use superpages, *e.g.*, stack pages would rarely use 4MB pages that frame buffers could use. Multiple page size TLBs also can be used as configurable page size TLBs if the operating system uses mappings of only a single page size. The interesting case, is when the operating system uses a mix of page sizes making a dynamic choice of different page sizes for different page blocks—page-size assignment.

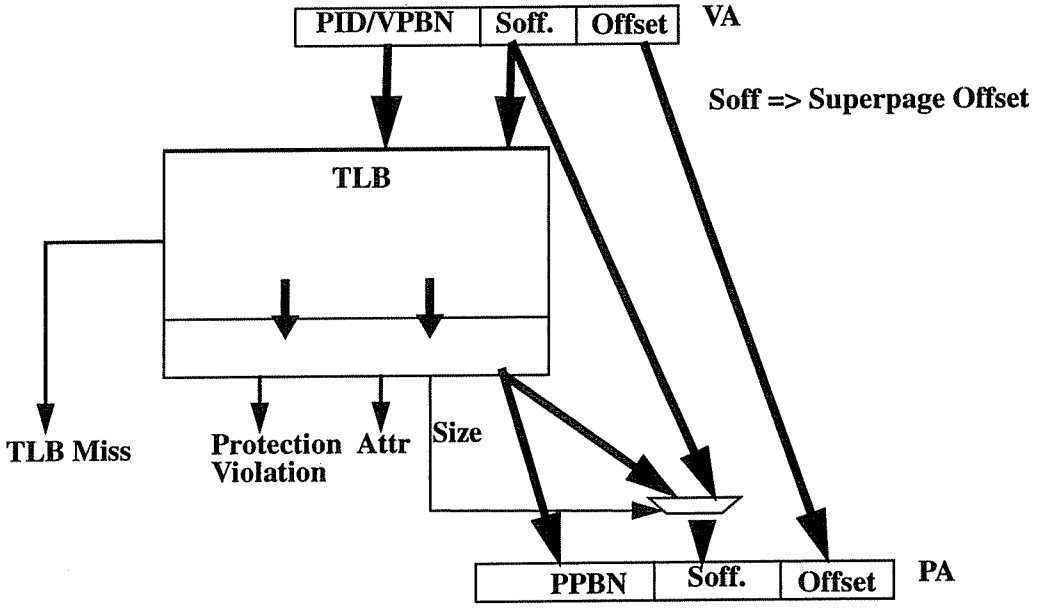
In this thesis, I assume superpage TLBs that support two page sizes—a base page size of 4KB and a medium-sized superpage size that is a power of two multiple of 4KB. I also assume that the operating system supports the generic use of superpages with a dynamic page-size assignment policy (as explained in Section 2.7.1) that chooses between these two page sizes. I do not explore the effect of using more than two page sizes simultaneously because I did not complete in time a working implementation of the page-size assignment policy and mechanisms needed to support more than two page sizes.

3.2 Mechanics of a superpage TLB

A superpage TLB is a small modification to a single-page-size TLB with a remarkable increase in TLB reach. This section identifies the basic hardware differences from a single-page-size TLB and discusses fully-associative and set-associative implementations.

Each TLB block in a superpage TLB can store mappings for different page sizes. The TLB uses for tag comparison the VPN bits assuming the smallest page size with the Offset bits passed through to physical address generation. A superpage TLB adds two things to a single-page-size TLB design (Section 1.5), as shown in Figure 3-1. First, each TLB block includes a page size mask (MASK) that identifies which bits of the VPN are part of the superpage VPN. The tag comparators include in the tag match only the superpage VPN bits, treating the rest of the tag bits as "don't care". Second, the physical address generation depends on the page size corresponding to the virtual address—for superpages the Soff field from the virtual address is used in the physical address, and for base pages the PPN bits output from the TLB are used. On a TLB hit, the page size attribute (SZ) read from the TLB controls a multiplexor to implement the physical address generation, as in Figure 3-3.

Figure 3-3: Structure of a superpage TLB

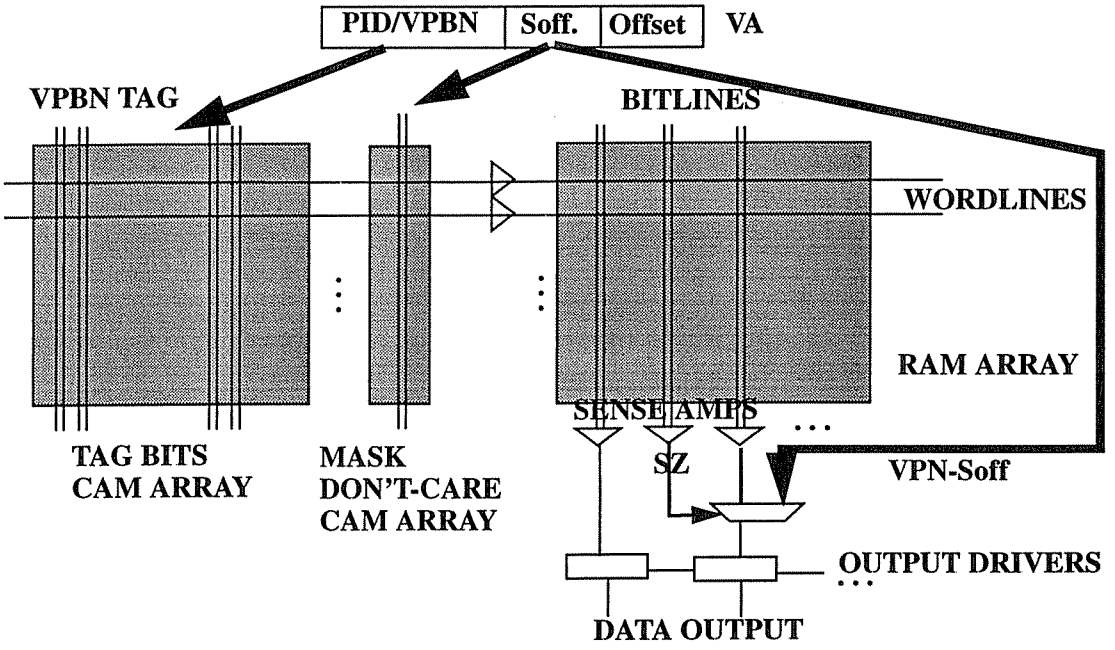


3.2.1 *Fully-associative superpage TLBs*

A fully-associative TLB includes an individual tag comparator for every TLB block that can be modified to support multiple superpage sizes. If the operating system uses superpages, the number of TLB misses can decrease significantly. Further, if the operating system does not use superpages, a fully-associative superpage TLB behaves exactly as a single-page-size TLB with the same number of blocks.

A fully-associative superpage TLB (shown in Figure 3-4) uses regular CAM cells for the VPBN bits (assuming the largest superpage size) and don't-care cells for the superpage offset bits in the tag (Appendix A). Implementation of the rest of the tag remains unchanged from a single page-size TLB—except for a longer match line to traverse the wider don't-care cells. The MASK bits store the page size in a predecoded form in the don't-care cells, *e.g.*, a superpage mapping for four base pages stores a mask of 0011 and prevents the two low-order bits of the VPN from participating in the tag match. When loading a mapping into the TLB, the PTE format can include the MASK field, the TLB miss handler can read the MASK from a special register, *e.g.*, MIPS R4000 [Kane92], or hardware can decode the size attribute in the PTE, *e.g.*, UltraSPARC [Yung95].

Figure 3-4: A fully-associative superpage TLB



The size field (SZ), read from the data array along with the attributes, controls physical address generation. This increases the TLB access time as it serializes the RAM access and the multiplexor control. This also requires the size field to be stored in the data RAM. It is possible to use the mask field in the tag to setup the physical address multiplexor in parallel to the data RAM access. However, it is difficult to read a CAM during the tag match and extra wires between the two arrays also make it difficult to implement.

Adding superpage support to a fully-associative TLB has very small area and access time overheads (bottom of Table 3-1) but can significantly reduce the number of TLB misses. The number of TLB misses declines for two reasons. First, the increased TLB reach allows the TLB to hold a larger fraction of the working set and misses less often, *e.g.*, 64KB superpages increase the TLB reach of a 4KB single-page-size TLB by a factor of 16. Second, a superpage mapping loads mappings to multiple base pages on a single TLB miss that would have taken multiple TLB misses in a single page size TLB. Table 3-1 shows the normalized[2] speedup when using superpage TLBs supporting two page sizes—a superpage and a base page size of 4KB. The simulation assumes that the operating system implements a page-size assignment policy that uses superpages—the policy described in Section 2.7.1. The speedups shown in Table 3-1 are significant, 1.05 to 1.21. The workloads I use spend significant time in TLB miss handling, smaller workloads may have less speedup.

3.2.2 *Set-associative superpage TLBs*

A set-associative TLB reads out a selected set of tags from the tag array and the corresponding data from the data array. Tag comparators compare the few tags and output the corresponding physical address if a TLB hit. To support superpages, the tag comparators and the physical address generation can be modified to use don't-care bits and a multiplexor respectively as in fully-associative TLBs. However, indexing into the tag and data arrays to select the

2. The normalized execution time speedup is shown here, as explained in Section 2.5. Appendix I shows execution time speedups for individual workloads.

Table 3-1: Execution time speedups for fully-associative superpage TLBs relative to single-page-size (4KB) TLBs with same number of blocks

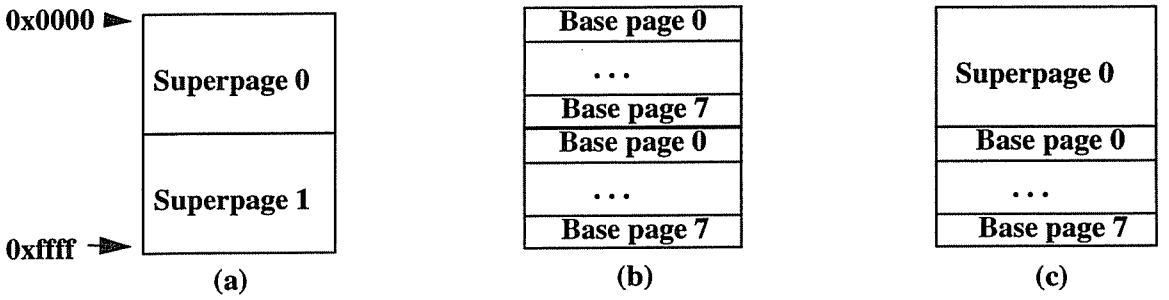
	#blocks	Superpage TLB with superpage size			
		8KB	16KB	32KB	64KB
Average Speedup	64	1.09	1.15	1.18	1.21
	128	1.07	1.11	1.13	1.17
	256	1.05	1.06	1.07	1.08
Relative Chip area	64-256	1.02	1.03	1.04	1.05
Relative Access time	64-256	1.00	1.00	1.00	1.00

set of tags to compare is nontrivial in a superpage TLB. A set-associative TLB usually uses the low-order bits of the VPN as index bits. Some TLBs use a hash function of all or some bits of the VPN, but for performance reasons the hash function always includes the low-order x bits of the VPN, where x is $\lg_2$(number of TLB sets).

When using superpages the page size for a virtual address is unknown when starting the TLB access to identify the index bits uniquely—the page size is known when inserting a mapping into the TLB. In this section, I discuss three ways to index a set-associative TLB that supports *two* page sizes only—a 4KB base page size and superpage size (*e.g.*, 32KB). None of the solutions I discuss here are, however, practical to support multiple superpage sizes. The exact index method, described later, is practical to implement if the page size corresponding to the virtual address is known when starting the TLB access, *e.g.*, the virtual address includes the page size [Cart94].

Which bits from the virtual address should the TLB use to index the tag array? There are at least three options to consider: the VPN of the base page; the VPN of the superpage; or the exact VPN with apriori knowledge of the page size. I illustrate these next with a direct-mapped 8-block TLB storing mappings for three different 64KB regions of address space shown in Figure 3-5.

Figure 3-5: Mapping a 64KB address space with 4KB and 32KB superpage mappings



Indexing the TLB by the VPN of the base page. The TLB uses as index the virtual address bits <14..12>, the least significant bits of the base page VPN. This would be same as the index used in a set-associative single-page-size TLB and would work fine for the case shown in Figure 3-5(b), which has only base pages. However, for the superpage mapping in Figure 3-5(a) or (c), all eight TLB blocks are candidates to store a superpage mapping, depending on the value of virtual address bits <14..12> that are part of the superpage page offset. This negates the very reason to support superpages and the TLB has the same performance as a similar set-associative TLB (the top row of Table 3-2). Therefore, there is no advantage to supporting superpages in a TLB indexed by the VPN of the base page.

Indexing the TLB by the VPN of the superpage. The TLB uses as index the virtual address bits <17..15>, the least significant bits of the superpage VPN. This would be same as the index used in a set-associative single-page-size TLB with a 32KB page size and would work fine for the case shown in Figure 3-5(a), which has only 32KB pages. However, using small pages, as in Figure 3-5(b) and (c), eight consecutive base pages compete for the same TLB block. For example, in Figure 3-5(b), base pages 0-7 all index into TLB block 0, causing many conflict misses. The collision cost is very high if no superpages are used—similar to using virtual address bits <17..15> in a single-page-size TLB with 4KB pages. If the operating system uses superpages, however, the collision cost may not be large, because:

- References to multiple base pages within the same page block cause collisions. The operating system should use superpage mappings for such page blocks (Figure 3-5(c)).

- If the program exhibits a non-looping sequential access pattern, *e.g.*, scanning an array, then the TLB uses only a single TLB block instead of overwriting the rest of the TLB. This helps applications such as **nasa7** and **fftpde** even when using only base pages.

- Increasing the set-associativity reduces the impact of collisions. Increasing the associativity to eight, for example, allows the base pages 0-7 to reside in separate TLB blocks though they map to the same set.

Superpage-index TLBs result in a simple hardware implementation for supporting medium-size superpages, but may perform much worse than equivalent single-page-size TLBs if the operating system does not use superpages. If the operating system uses superpages, the TLB incurs fewer misses, for reasons explained above, and results in execution time speedup relative to a single-page-size TLB that uses the base page index (Table 3-2). However, the workloads suffer a slowdown if the operating system does not use superpages, *e.g.*, if the operating system support is lacking or the application has small segments or there is shortage of physical memory. Thus, superpage-index TLBs are sensitive to the available operating system support and workloads that can use superpages.

Table 3-2: Execution time speedups for 256-block 4-way set-associative superpage TLBs relative to single-page-size (4KB) TLBs

	Indexing scheme	superpage size (using superpages)				superpage size (using base pages only)			
		8KB	16KB	32KB	64KB	8KB	16KB	32KB	64KB
Average Speedup	base-page-index	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	superpage-index	1.06	1.09	1.10	1.07	1.01	1.02	0.87	0.81
	exact-index	1.06	1.10	1.11	1.14	1.00	1.00	1.00	1.00
Relative Chip area		1.02	1.03	1.04	1.05				
Relative Access time		1.00	1.01	1.01	1.01				

Indexing the TLB by the exact VPN. If the TLB knew the page size before starting the TLB access or can magically guess the correct page size, *e.g.*, the virtual address may include the page size [Cart94], the TLB can be indexed by the superpage VPN (bits <17..15>) for superpages and by the base page VPN (bits <14..12>) for base pages. This solution would use a single TLB block for superpages without increasing the collision costs for using base pages. The third row of Table 3-2 shows that set-associative TLBs using the exact index are very effective if using superpages. Using the exact index is more effective because base pages within a page block map to different sets in an exact-index TLB. These base pages would have mapped to the same set in a superpage-index TLB and cause additional conflict misses.

However, an exact index set-associative TLB is not easy to implement if the page size corresponding to the virtual address is unknown prior to starting the access. There are at least three strategies and each has significant costs.

Parallel access: A dual-ported or replicated TLB can use different index bits on the two ports to translate the same virtual address. This solution has two opportunity costs that make it unattractive. First, a dual-ported TLB can improve the CPI of a processor by allowing two memory access translations per cycle that may increase overall performance better than TLB performance benefits of superpages, *e.g.*, multiple load/store pipes in processors already saturate the bandwidth of multi-ported single-page-size TLBs and supporting superpages would use half the TLB bandwidth. Second, a dual-ported TLB occupies a larger chip area and is slower to access. Other single-page-size or fully-associative TLB designs that occupy a similar area become attractive alternatives. For example, a 256-block fully-associative TLB requires less chip area than a dual-ported 256-block direct-mapped TLB, incurs fewer TLB misses, and can support multiple superpage sizes.

Rehash schemes: The TLB can first index assuming a base page and on a miss can repeat the access, next cycle, using the superpage index. Similar schemes have been used to improve the performance of set-associative CPU caches [Agar88, Kess89, Agarwal93] and page tables [Thak86, May94]. The TLB access takes a variable number of cycles and can complicate pipeline design. If TLB access is in the critical path, increasing the TLB hit time for superpage mappings decreases their usefulness. If TLB access is not in the critical path, *e.g.*, when using virtual-tagged caches, a rehash scheme may be practical. In particular, rehash schemes are attractive for operating systems to traverse hashed page tables that store superpage mappings (Section 7.4.2).

Split TLBs: A processor can include separate TLBs accessed in parallel for the two page sizes, similar to split instruction and data caches [Smit82]. The two TLBs can be either both set-associative, both fully-associative or one set-associative and one fully-associative. This has the disadvantage of unused hardware if pages are not appropriately distributed between the two page sizes.

Supporting more than two page sizes in set-associative TLBs makes them further unattractive. Indexing with the VPN of the largest supported superpage sizes increases the conflict misses for all smaller, more frequently used, page sizes. Indexing with the exact VPN requires either a) too many ports, b) many reprobes, or c) many separate TLBs. A compromise solution uses a set-associative single-page-size TLB for base pages and a separate fully-associative TLB for multiple superpage sizes, *e.g.*, HP PA-RISC [Lee89b], PowerPC [May94]. This allows a much larger single-page-size TLB to be built but restricts the number of superpage mappings and requires separate TLB miss handlers.

In summary, set-associative superpage TLBs can use either the base page index, superpage index, or exact index. The base page index does not reduce TLB misses, the superpage index has unacceptable performance if the operating system does not use superpages, and the exact index is costly to implement as the page size corresponding to a virtual address is unknown when starting the lookup. Further, neither is practical when simultaneously supporting more than two page sizes.

3.3 TLB miss handling in a Superpage TLB

TLB miss handling for a superpage TLB may be more complicated than for a single-page-size TLB. It is important that superpage TLB miss handling does not increase the TLB miss penalty and offset the gains from the fewer TLB misses incurred. I first show a naive way to handle TLB misses for a superpage TLB using a single-page-size page table. I then show that by modifying the operating system and page table to store superpage mappings explicitly, the TLB miss penalty can be no worse than in a single-page-size TLB.

Most operating systems use page tables that store base page mappings only. A naive TLB miss handler for a superpage TLB scans the PTEs corresponding to all base pages that belong to the faulting page block and, if they are all superpage compatible, loads a superpage TLB block. Otherwise, it loads a base page mapping for the faulting virtual address. The naive TLB miss handler is as follows:

```
Mapping = Find_mapping(VPN);
for i = 0 to (s-1), except Block Offset(VPN)    /* s is number of base pages per superpage */
        if (!compatible(Find_mapping(i + VPBN), Mapping)) break;
if (all bases pages compatible) {
        Mapping.SZ = 1;
        TLB_demap(VPBN, s * PAGESIZE);    /* remove existing base page mappings */
}
load_TLB(Mapping, blocki);                          /* set the MASK field based on SZ */
```

This is clearly inefficient, does not extend easily to multiple superpage sizes, and would increase the TLB miss penalty by more than an order of magnitude neutralizing any benefit due to reduction in the number of TLB misses. An alternate solution, used by many current microprocessors, is to store explicitly superpage mappings in the page table (*e.g.*, SPARC reference MMU [SPAR91]) that the TLB miss handler can load into the TLB without further checking. Storing superpage mappings in the page table has two advantages. First, it is more efficient as the operating system does compatibility checks only during page faults, which are less frequent than TLB misses. Second, when loading a superpage mapping into the TLB, preexisting base page mappings for the same virtual address range as the superpage must be invalidated from the TLB. The operating system can guarantee this by invalidating the base page mappings from the TLB during page promotion, instead of requiring the TLB miss handler to include a TLB_demap operation (as shown above). This simplifies the TLB miss handler for the superpage TLB to be same as a single-page-size TLB miss handler:

```
Mapping = Find_mapping (VPN);              /* base page or superpage */
load_TLB(Mapping, blocki);                 /* blocki is TLB replacement victim block*/
```

TLB miss penalty depends on the time to traverse a page table that stores superpage mappings. Replicating a superpage PTE at each base page PTE site extends any single-page-size page table to store superpage mappings without increasing the TLB miss penalty over that for a single-page-size TLB. Chapter 7 discusses other adaptations to popular page tables to store superpage mappings. *Clustered page tables*, for example, reduce page table size using medium-size superpage mappings, without affecting the TLB miss penalty.

In some microprocessors, the default TLB miss handler services only TLB misses to base page mappings. On base page table misses, the operating system page fault handler traverses other data structures to find and load superpage mappings into the TLB, *e.g.*, block TLB miss handling in PowerPC [May94].

3.4 Sample design given area constraint

A fully-associative single-page-size TLB can support superpages with little area and access time overhead (Section 3.2.1). Further, the TLB performance of a fully-associative superpage TLB is comparable to that of a single-page-size TLB in the absence of operating system support and vastly superior in the presence of operating system support. Thus, if one were building a fully-associative single-page-size TLB, based on other single-page-size design studies, adding superpage support is a small additional cost with huge potential benefits. Many microprocessors support superpages with fully-associative TLBs and in this section I try to answer the question: Does a superpage TLB always outperform a single-page-size TLB of comparable implementation complexity?

I answer this question by comparing superpage TLBs with different set-associative single-page-size TLB implementations that require comparable chip area[3]. I use the area model described in Section 2.2 to find the size of fully-associative single-page-size TLBs and superpage TLBs supporting superpage sizes of 32KB or 64KB that have comparable chip area to 4-way set-associative single-page-size (4KB) TLBs. Note that fully-associative TLBs can have non-power-of-two number of TLB blocks. Set-associative TLBs typically require a power-of-two number of sets. Table 3-3 shows the normalized execution time speedup of the different TLBs relative to the set-associative single-page-size TLB. The table differentiates between superpage TLB performance depending on whether the operating system uses superpages. The columns titled "Speedup using superpages" assume the operating system uses superpages (using the policy in Section 2.7.1) and the columns titled "Speedup using base pages" assume the operating system uses only base pages. Set-associative superpage TLBs use the superpage index. There are two observations to make:

Table 3-3: Execution time speedups for superpage TLBs relative to set-associative single-page-size (4KB) TLBs of comparable chip area

Area (rbe)	4KB Single-page-size TLB		4KB/32KB Superpage TLB			4KB/64KB Superpage TLB		
	#blocks	Speedup	#blocks	Speedup using superpages	Speedup using base pages	#blocks	Speedup using superpages	Speedup using base pages
19160	162 fully-associative	1.02	156 fully-associative	1.12	1.02	154 fully-associative	1.14	1.02
	256 set-associative	1.00	256 set-associative	1.10	0.87	256 set-associative	1.07	0.81
35412	304 fully-associative	1.00	293 fully-associative	1.07	1.00	290 fully-associative	1.07	1.00
	512 set-associative	1.00	512 set-associative	1.06	0.84	512 set-associative	1.01	0.77

3. TLB access time is also an important criterion in choosing a TLB design. However, the assumptions in my timing model prevent access time comparison of fully-associative TLBs with set-associative TLBs (Section 2.3).

First, when the operating system uses superpages, superpage TLBs result in a significant speedup over the set-associative single-page-size TLBs, for these workloads. Second, in the absence of superpage operating system support, *i.e.*, when using only base pages, the superpage TLBs have worse performance than the single-page-size TLBs. Though the fully-associative TLBs appear to have comparable performance to the set-associative TLBs, excluding **fftpde** the fully-associative TLBs have an execution time slowdown (Tables I3-3a and I3-3b in Appendix I). They incur more TLB misses as they have fewer TLB blocks and a smaller TLB reach. The set-associative superpage TLBs use suboptimal index bits when using only base pages and incur a significant slowdown in execution time.

Thus, my results show that for these workloads, superpage TLBs are clearly more effective than equivalent single-page-size TLBs but only if the operating system uses superpages. Hardware designers are building superpage TLBs, assuming that operating systems will use superpages and exploit the potential for execution time speedup. Commercial operating systems, however, do not support medium-sized superpages and use base pages only. Hardware designers could have built better set-associative single-page-size TLBs if the intent was to use only base pages.

3.5 Conclusion

A superpage TLB allows different TLB blocks to map different sized virtual address regions. With simple hardware extensions to a single-page-size TLB, superpages increase the TLB reach by one or two orders of magnitude. For superpage TLBs to be effective, however, the operating system must use superpages and exploit the increased TLB reach. While some commercial operating systems use superpages for special cases only, *e.g.*, databases, there is some evidence that upcoming operating system releases will support superpages for more user programs.

This chapter shows that with proper operating system support, fully-associative superpage TLBs using medium-sized superpages can result in a significant speedup in execution time, *e.g.*, 10%, relative to using equivalent single-page-size TLBs. As explained in Section 2.4, the workloads I chose have potential for and show more execution time speedups than many small and short-lived programs.

This chapter also shows that set-associative implementations of superpage TLBs that support more than two page sizes are impractical. Set-associative implementations of superpage TLBs that support two page sizes can use the superpage index, but perform much worse than single-page-size TLBs if superpages are not used by the operating system.

Superpage support is easy to implement in hardware due to the strict definition of a superpage. However, the restrictions make superpages ineligible to be used in many situations—segments smaller than the superpage size (*e.g.*, a 60KB file with a superpage size of 64KB), segments with unaligned boundaries, segments with holes, pages with different attributes. Thus, there is an opportunity cost where a less restrictive definition of "superpage" would have resulted in more frequent usage. The next two chapters describe subblock TLBs that offer a better alternative to medium-sized superpage TLBs. Subblock TLBs incur fewer TLB misses and require simpler operating system support. The last two chapters describe operating system and page table support that minimize overhead and make it more attractive to use superpages.

Chapter 4 Complete-subblock TLBs

This chapter explores the use of subblocking[1], a feature commonly used in cache design, to increase TLB reach as an alternative to use of medium-sized superpages[2]. The key idea in subblocking is to allow a single TLB block to map multiple base pages but with each base page having it's own mapping. I call this *complete subblocking* and a TLB built with this feature a *complete-subblock TLB*. Complete-subblock TLBs have the same TLB reach advantages of medium-sized superpages and exploit spatial locality to improve TLB performance but do not require any operating system support. This chapter discusses the basic operation of complete-subblock TLBs, some implementation issues, and compares the advantages and disadvantages of a complete-subblock TLB with single-page-size and superpage TLBs. While commercial processors, such as MIPS R4000, implement complete-subblock TLBs with subblock factor of two, I show complete-subblock TLBs with larger subblock factors to be effective as well. My results show complete-subblock TLBs to be a superior choice to conventional single-page-size TLBs that occupy comparable chip area. The primary contribution of this chapter is that I show how hardware designers can use the larger chip area available in today's VLSI implementations to build complete-subblock TLBs that are more effective and *faster* to access than equivalent single-page-size TLBs.

A complete-subblock TLB block's tag maps a fixed page block size but the data stores separate mappings for the base pages. Mappings to base virtual pages share a single TLB block if they are part of the same virtual page block. Base pages within the page block need not be properly placed, page block aligned, or have the same attributes, and only a subset need be mapped or present in memory. The hardware complexity to build a complete-subblock TLB is small, but the chip area cost is high. However, complete-subblock TLBs require no additional operating system support.

A TLB block for a complete-subblock TLB of subblock factor s, has a tag that contains the virtual page block number (VPBN)—the VPN without the $\log_2(s)$ low order bits—and a data part that has space for s mappings. A complete-subblock TLB block also has a block valid bit (BV) in the tag to identify valid TLB blocks besides the individual subblock valid bits (V_0-V_3) in the data. Figure 4-1 compares a single-page-size TLB block and a complete-subblock TLB block with subblock factor of 4.

For a typical 64-bit system, the size of the tag (~64 bits) is comparable to the size of the data (~64 bits) in a single-page-size TLB. A complete-subblock TLB uses fewer tags by associating a single tag with multiple mappings—saving chip area and reducing access time. The saving does not come for free—only mappings to pages that belong to the same page block can be stored in the data fields sharing a single tag. If a workload has good spatial locality—*i.e.*, references many pages within a page block—the TLB has comparable performance to a TLB that has independent tags but at a lower cost. If the program has bad locality—*i.e.*, references only a small fraction of the pages within a page block—the TLB performance is worse. This is the tradeoff in using subblock TLBs or caches.

1. Subblocking [Hill84] has also been called sectoring [Lipt68] and address/transfer blocks [Good83].

2. This chapter concentrates on subblock TLBs as an alternative to medium-sized superpages. Appendix E illustrates how subblock TLBs support large superpages.

Figure 4-1: Structure of a complete-subblock TLB block

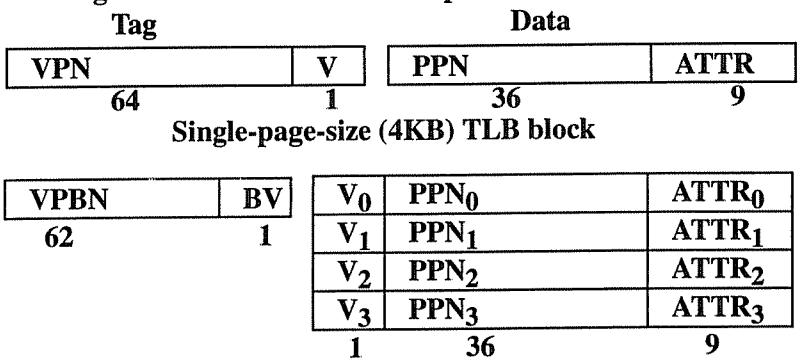
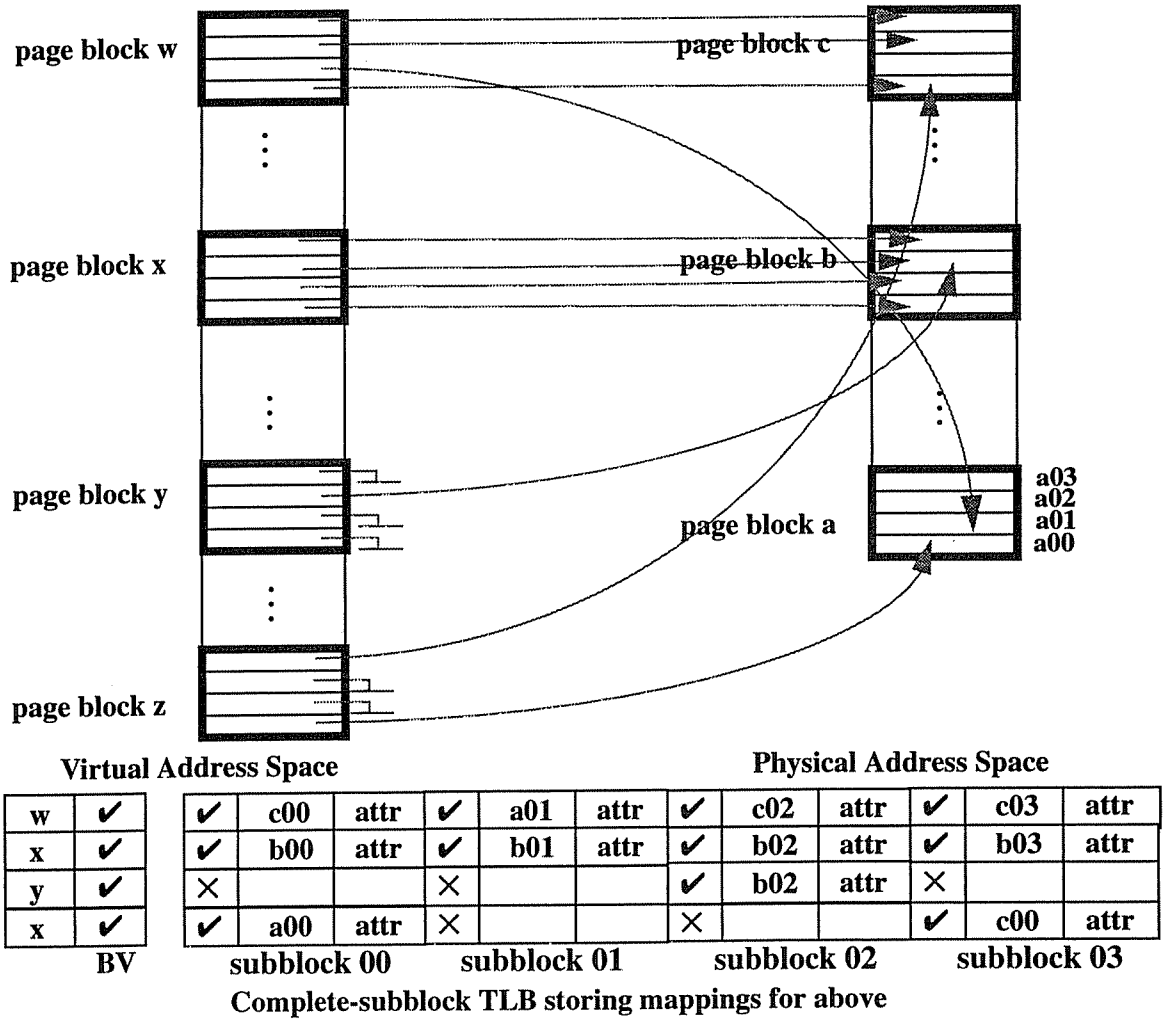


Figure 4-2 shows how a fully-associative complete-subblock TLB stores some mappings from a virtual address space to a physical address space. A complete-subblock TLB's fields are as shown in the TLB block format in Figure 4-1. Note that each page block in the virtual address space requires a single TLB block—a superpage TLB uses multiple TLB blocks for page blocks that cannot be mapped by a superpage mapping (Figure 3-2).

Figure 4-2: Virtual Address to Physical Address Mappings in a Complete-subblock System

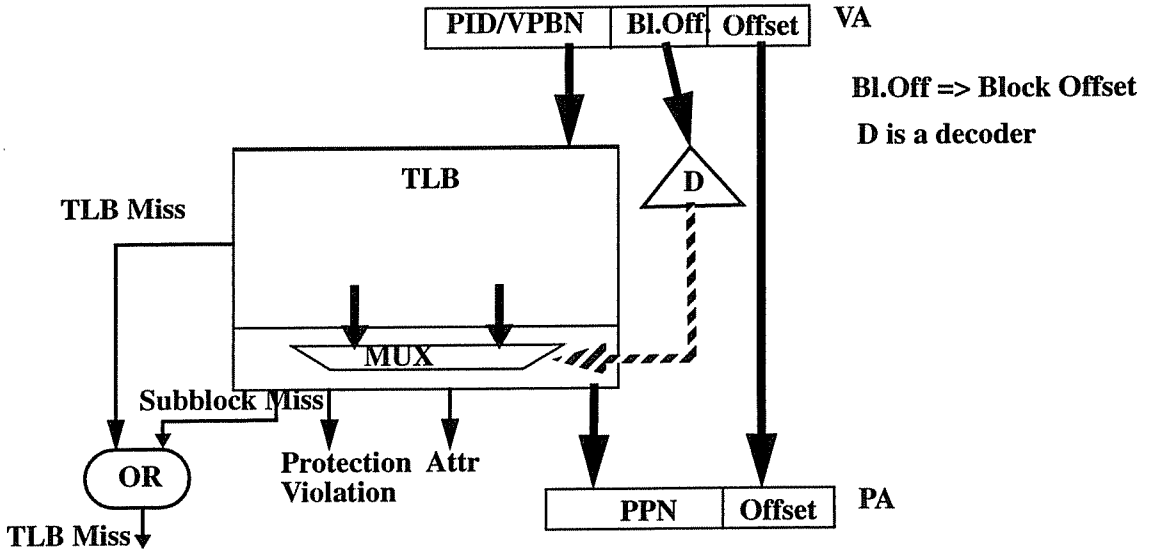


The rest of this chapter describes the structure and operation of a complete-subblock TLB, describes TLB miss handling in a complete-subblock TLB, introduces *preloading* or prefetching into subblock TLBs, compares alternate TLBs that occupy comparable chip area, and concludes with a comparison of complete-subblock TLBs with other single-page-size and superpage TLBs.

4.1 Mechanics of a Complete-subblock TLB

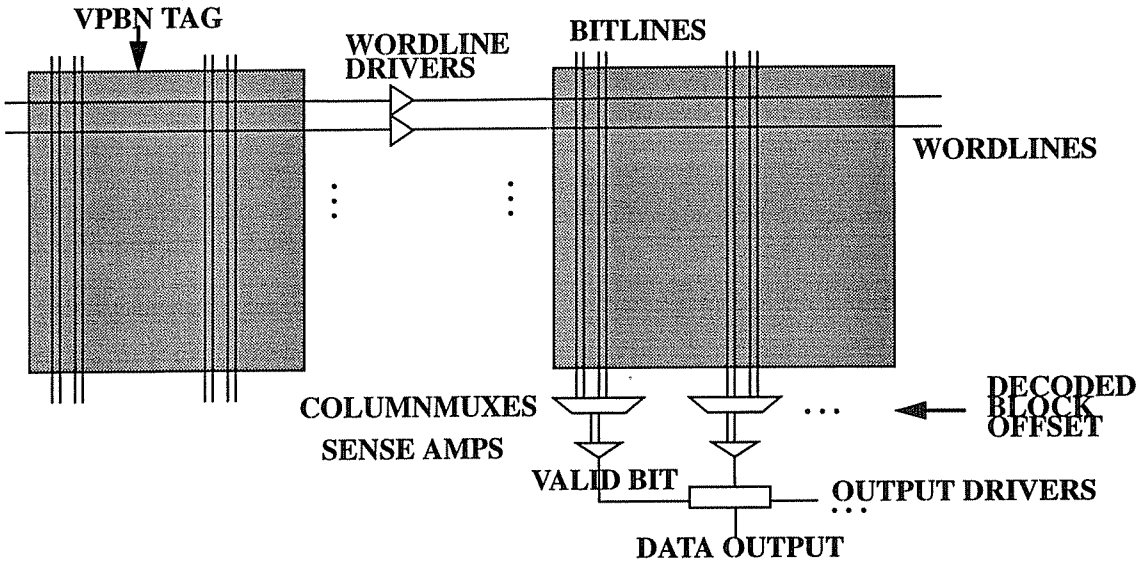
The structure and operation of a complete-subblock TLB (Figure 4-3) differs in three ways from that of a single-page-size TLB (Figure 1-3) with the same number of blocks and associativity. First, since each TLB block maps s pages, the tag stored in a complete-subblock TLB block is $\log_2(s)$ bits smaller. Second, the data RAM is s times wider and the low-order $\log_2(s)$ bits of the VPN, the Block-Offset bits, control a *subblock multiplexor* to select the appropriate subblock from the data RAM. Third, each TLB block has multiple (s) subblock valid bits. TLB miss signal generation includes the status of the appropriate subblock valid bit read from the TLB and the conventional result of the tag compares. Note that the Block Offset bits of the virtual address do not pass through to form the physical address as the Page Offset bits do and the TLB data stores the full PPN.

Figure 4-3: Structure of a complete-subblock TLB



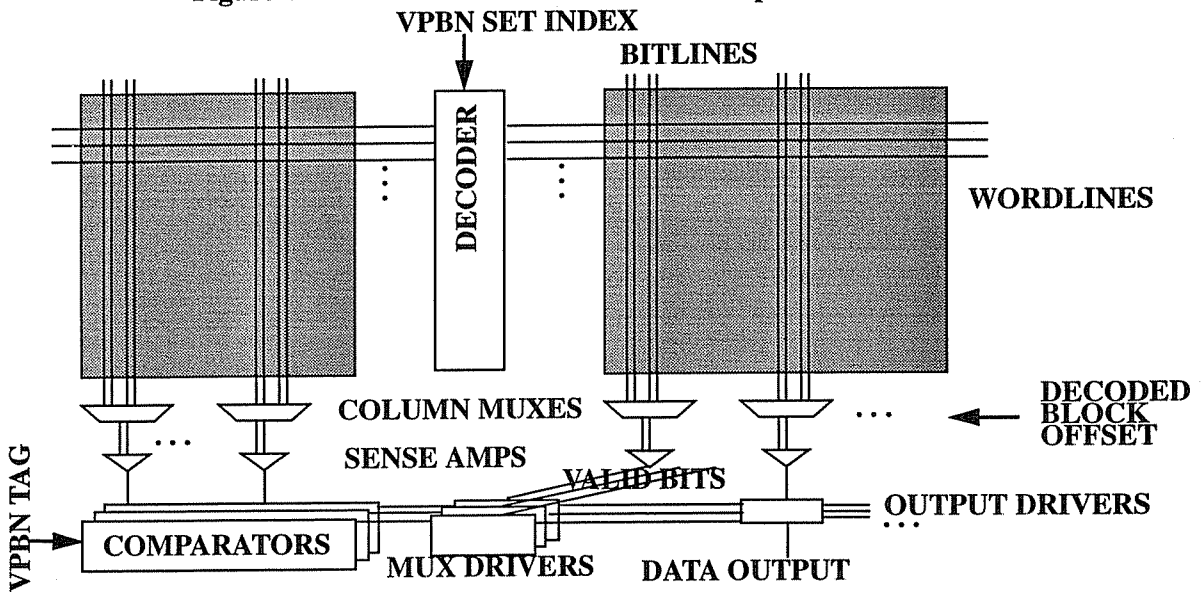
The basic structure of a fully-associative complete-subblock TLB is very similar to that of a single-page-size fully-associative TLB (Section 1.5.1) as shown in Figure 4-4. Each TLB block stores a tag consisting of a virtual page block number (VPBN) and a block valid bit—implemented using the variations described in Appendix A. Each TLB block stores s mappings in a single row of the data RAM. The tag memory, implemented as a content-addressable memory, compares the VPBN of the faulting address. The selected row of RAM cells reads out s mappings onto the bitlines. The column multiplexors, controlled by the decoded block offset bits, select one bit from each set of s bits to output one mapping. The subblock valid bit read out of the data RAM enables the output drivers. In a single-page-size TLB the TLB hit signal, the logical OR of the wordlines, enables the output drivers.

Figure 4-4: Structure of a fully-associative complete-subblock TLB



A set-associative complete-subblock TLB has a similar construction. The tag and data RAM decoders use as index bits the low order bits of the VPBN instead of the VPN. The Block Offset bits control column multiplexors to select only *a* out of *s*a* mappings, where *a* is the associativity. The subblock valid bits read from the data RAM (there are *a* bits read, one for each degree of associativity) ANDed with the tag compare result enable the data output through the MUX drivers. The tags for a set-associative complete-subblock TLB need not store a block-valid bit and can use the subblock valid bits only, as in Figure 4-5.

Figure 4-5: Structure of a set-associative complete-subblock TLB



4.1.1 Implementation Issues for complete-subblock TLBs

In implementing a complete-subblock TLB, three issues must be addressed. First, for fixed number of blocks, the size of the data RAM increases and may increase access time. Second, there are at least two alternatives for positioning the subblock multiplexor. Third, the presence

of multiple subblock valid bits provides at least three different implementation alternatives. These issues are similar to those faced in implementing subblock caches.

Complete-subblock TLBs with large subblock factors will result in a large, wide data RAM array. The access time for a monolithic array will be slow. It also may require unusually large wordline drivers to drive the wide data word. RAM designers commonly divide a large RAM array into smaller blocks to improve cycle time and layout aspect ratio. Similar techniques are applicable here but this thesis does not discuss details of these or other optimizations [Wada92, Wilt93]. One simple optimization for fully-associative subblock TLBs is to split the RAM array into two halves and place the two halves on either side of the tag array. While this requires additional drivers, it reduces access time. In my model, I assume a monolithic RAM array.

The position of the subblock multiplexor also affects access time. Appendix C discusses two alternatives. The first alternative is to use a column multiplexor in the data RAM to act as the subblock multiplexor. The second alternative is to combine the subblock multiplexor with the output multiplexor associated with the output drivers—this is analogous to implementing a multiplexor in caches that read out a word smaller than the cache line size. Using the column multiplexors has several advantages and results in a faster access time, as explained in Appendix C. I assume the use of column multiplexors when estimating the access time for complete-subblock TLBs.

The presence of multiple subblock valid bits results in a more complicated design than a non-subblock TLB. A simple solution that I describe in Appendix B uses a block valid bit in the tag and stores subblock valid bits in the data RAM. The block valid bit stores the logical OR of all the subblock valid bits. This allows tag comparison to work as in a single-page-size TLB and only requires examining the single subblock valid bit that is output from the subblock multiplexor. My area model, access time model and TLB simulations assume the use of block valid bits for complete-subblock TLBs, as shown in Figure 4-4 for fully-associative TLBs. Though set-associative complete-subblock TLBs need not use block valid bits (Figure 4-5), I assume the use of block valid bit for uniformity.

Using the block-valid bit, however, requires the operating system or hardware to guarantee that there can never be two TLB blocks with the same VPBN tag co-residing in the TLB. A complete-subblock TLB can have two TLB blocks with the same tag if the TLB miss handler uses different TLB blocks for base page mappings that belong to the same TLB blocks. Section 4.2 and Appendix D illustrate two examples where a complete-subblock TLB can have two TLB blocks with the same tag. If the operating system cannot guarantee against this error condition, the complete-subblock TLB implementation must include the appropriate subblock-valid bit in the tag comparison. Selecting the appropriate subblock valid bit requires decoding the block-offset field of the virtual address and serializing tag comparison. In Appendix B, I discuss two alternatives for implementing subblock valid bits in the tag. The first alternative stores subblock valid bits in tag memory and extends the tag compare logic. The CAM array in fully-associative TLBs or the tag comparator in set-associative TLBs is extended to compare the decoded block offset field of the virtual address with the subblock valid bits. The second alternative recognizes that the decoded block-offset has a one-hot encoding and uses a valid bit RAM that can be read in parallel with tag comparison and may be faster.

4.1.2 Effect of complete subblocking

Modern microprocessors have an increasing number of transistors and chip area available. A complete-subblock TLB is one way to use the extra area. This section explores the effect of

increasing the TLB reach of a single-page-size TLB by keeping the number of blocks constant and storing multiple mappings in each TLB block using complete subblocking. The complete-subblock TLB occupies a larger chip area and has a larger access time but incurs fewer TLB misses.

The complete-subblock TLB has a larger TLB reach (subblock factor, *s*, times larger) and incurs fewer TLB misses than a single-page-size TLB with the same number of blocks and associativity. As shown in Table 4-1, the reduction in the number of TLB misses translates to normalized execution time speedups of 1.04 to 1.17 for my workloads. The performance improves further when using prefetching as described in Section 4.2.2.

Table 4-1: Execution time speedups for complete-subblock TLBs relative to single-page-size (4KB) TLBs with same number of blocks

TLB Type	#blocks	Subblock factor			
		2	4	8	16
Fully-Associative	64	1.04	1.09	1.16	1.17
	128	1.05	1.10	1.11	1.15
4-way Set-Associative	256	1.05	1.08	1.10	1.14

The complete-subblock TLBs, however, occupy a larger area and require additional control logic. The left half of Table 4-2 compares the area for a complete-subblock TLB with the area for a single-page-size TLB using the area model described in Section 2.2. The key observation is that a complete-subblock TLB of subblock factor *s* does not require *s* times the area of the single-page-size TLB. Since a complete-subblock TLB only duplicates the data portion but saves on tag memory, the savings are more significant in fully-associative TLB. The set-associative complete-subblock TLBs have a higher overhead as the tags account for a smaller portion of the area.

Table 4-2: Chip Area and Access Time for complete-subblock relative to single-page-size TLBs with same number of blocks

TLB	Chip Area				Access Time			
	2	4	8	16	2	4	8	16
(64-256) block Fully-Associative	1.24	1.71	2.67	4.59	1.02	1.06	1.11	1.20
256-block 4-way Set-Associative	1.38	2.14	3.66	6.71	1.00	1.01	1.04	1.12

Unfortunately, the extra area usually translates to an increase in TLB access time also. If TLB access time is on the critical path for processor cycle time, this is an important consideration. The right half of Table 4-2 shows the access time, calculated using the timing model described in Section 2.3. Doubling the subblock factor impacts access time in three ways: the data RAM lookup time increases as the wordline has to drive a wider array, the number of inputs to the column multiplexors double and increases the data RAM access time, but the shorter tags reduce the tag lookup and compare time.

For fully-associative TLBs, both the tag and data lookup are on the critical path. The data RAM slowdown dominates and results in a slower TLB. For set-associative TLBs, only the tag side is on the critical path and that decreases! Increases in the data RAM access time do not affect the access time---except for very large subblock factors where the data RAM access

becomes the critical path. However, the tag compare multiplexor output must be driven across a much wider data RAM and adds to the critical path. It is possible to optimize this path to make the set-associative subblock TLBs faster than the single-page-size TLBs! This is important since in many modern microprocessor designs, access time is more important than small increases in chip area.

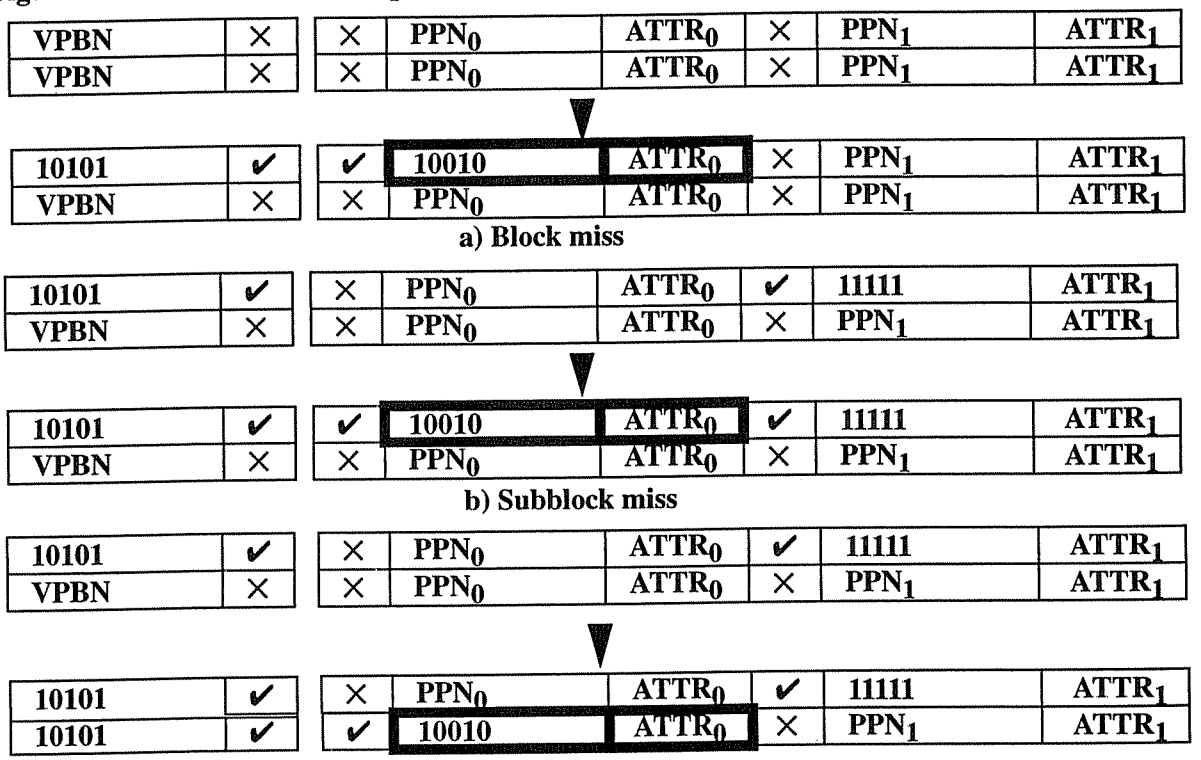
In summary, in designs where additional area is available for the TLB, extending the TLB reach through complete subblocking with a small subblock factor is an attractive option. In fully-associative TLBs the access time will increase but in set-associative TLBs the access time can decrease!

4.2 TLB miss handling for complete-subblock TLBs

TLB miss handling for a complete-subblock TLB is understandably more complicated than for a single-page-size TLB. This section explores the three steps described in Section 1.5.3 and discusses the changes needed to support complete-subblock TLBs—subblock miss checking and loading a new mapping into the correct subblock in the TLB data.

First, the process of locating the mapping for the faulting address is the same as in a single-page-size system. The page table structure, page table traversal hardware and/or software need not change and can use the same page size and algorithms, including reference/modified bit updates.

Figure 4-6: Subblock miss example for VPN 101010 (Complete-subblock TLB subblock factor 2)



Second, deciding where to place the mapping in the TLB is different. A TLB miss in a non-subblock TLB always results in allocating a new TLB block to store the new mapping, possibly

requiring TLB replacement. TLB misses in a subblock-TLB can be either a *block miss* or a *subblock miss*. In a subblock miss, an existing TLB block may be able to hold the new mapping in an unused subblock with the same tag. Figure 4-6 illustrates three possible scenarios when loading a new mapping for VPN 101010: a) a block miss, b) a subblock miss, and c) a subblock miss that uses a new TLB block. Sharing subblocks is the key to the performance advantages of complete-subblock TLBs and Section 4.2.1. explains in detail how subblock-miss checking can be implemented to avoid condition c)[3].

Third, loading a new mapping into the TLB differs on whether the miss resulted from a subblock miss or not. On a regular miss, TLB replacement clears the victim TLB block and writes a new tag and one of the fields at the correct offset in the data part of the TLB block. On a subblock miss, only one mapping should be written while the tag without affecting the rest of the TLB block. This is similar to writing a word to a cache with block size larger than a word. Most implementations will read the existing TLB block, update the subblock in question and rewrite the full TLB block into the same location but may take two cycles.

4.2.1 Implementing subblock miss checking

As explained above, before loading a new mapping into the TLB, hardware or software should check if an existing empty subblock can be used, *subblock miss checking*. A pure software approach is inefficient, requiring scanning all the TLB blocks' tags, and hardware support is essential for efficient TLB miss handling. This section suggests two mechanisms.

First, TLB lookup may already identify the TLB block where the new mapping can be loaded on a subblock miss. The MIPS processors, for example, cause a special trap on subblock misses. Hardware could store this information in a register accessible to the TLB miss handler. The miss handler can directly specify the destination TLB block (the virtual address implicitly identifies the subblock). This solution is straightforward. The main disadvantage is that nested TLB misses (TLB misses during execution of the TLB miss handler) will require special handling as the old information will be lost.

The second way to implement subblock miss checking is to perform a TLB lookup just before loading the new mapping to determine if a subblock miss occurred. The hardware can check for subblock misses as part of the **load_TLB** operation, making it a multi-cycle operation. In a system where the software miss handler explicitly identifies where the mapping is to be placed, the hardware can provide a special **load_TLB_subblock** operation that checks for subblock misses and updates the TLB. On a block miss the software can continue as before. This adds to the TLB miss handler as illustrated below:

```
PTE = Find_mapping(VPN)
load_TLB_subblock(PTE)
if (fail) load_TLB(PTE, blocki)
```

In the next section, I discuss preloading, which attempts to eliminate subblock miss checking.

4.2.2 Preloading

This section discusses a software approach to TLB miss handling in complete-subblock

3. Having two blocks with the same tag will cause electrical problems in most implementations and is undesirable.

TLBs as an alternative to subblock miss checking hardware support. This approach handles TLB misses by always loading all the mappings for a page block instead of just a single mapping for the faulting page—*preloading*. If the subblock factor is two, the TLB miss handler will fetch and load into the TLB the mappings to both the pages in the page block that the faulting address belongs to, *e.g.*, MIPS R4000 TLB miss handling.

A preloading complete-subblock TLB miss handler would traverse the page table, possibly multiple times to locate all the base page mappings for the page block that the faulting virtual address belongs to. It can then load all the mappings using either a single operation or a series of operations. The TLB miss handler might be as follows:

```
for i = 0 to (s-1)
        PTE_array[i] = Find_mapping(VPBN+i)
load_TLB(PTE_array, blocki)
```

If the page table stores mappings for a page block adjacent in memory, then the miss handler can be more efficient. First, it needs to traverse the page table only once—to the mapping for the first base page of the page block. It can then read consecutive mappings using a simple pointer increment. Second, the cache performance improves as the mappings would fit in fewer cache blocks. Third, longer width memory loads, *e.g.*, 128-bit loads, can reduce the number of instructions executed in a TLB miss handler. However, the TLB miss penalty increases if using preloading. Since the costs of a TLB miss handler are dominated by the cost of traps and traversing the page table, the increase in TLB miss penalty is small if base page mappings for a virtual page block are adjacent in memory.

Preloading has two advantages. First, it requires no hardware support for subblock miss checking as it loads all subblocks of the TLB block. Second, preloading results in significantly fewer TLB misses. By prefetching neighboring mappings on a single TLB miss, programs that exhibit spatial locality benefit by encountering one TLB miss per page block instead of one per base page. Table 4-3 shows the normalized speedup for a complete-subblock TLB with preloading over a complete-subblock TLB without preloading. Note that a complete-subblock TLB already has a significant speedup relative to a single-page-size TLB. I assume the same TLB miss penalty of 40 cycles for both the preloading and non-preloading versions of the TLB miss handler in Table 4-3. In practice, the TLB miss penalty for a preloading TLB miss handler can be higher. To quantify the tradeoff between the increase in the TLB miss penalty vs. decrease in the number of TLB misses, the table includes a *critical TLB miss penalty*. A critical miss penalty of two implies that a TLB with preloading and less than twice the TLB miss penalty[4] (than a TLB without preloading) delivers better TLB performance. Table 4-3A to Table 4-3D show the individual benchmark speedups and critical miss penalties. Table 4-3 shows the normalized speedup and the critical miss penalty that would result in a normalized speedup of 1.00.

Both fully-associative and set-associative TLBs benefit from preloading. With large TLBs the working sets of some workloads fit in the TLB incurring only compulsory misses and do not show much benefit from preloading. The critical miss penalties also show that the preloading TLB miss handler can be only slightly more complicated. The critical miss penalty for subblock factor of 2 is 1.33, *i.e.*, for preloading to be worthwhile, a TLB miss handler must spend less than 33% extra time as in a single-page-size TLB to fetch and load two base page PTEs. For workloads that spend less time in TLB miss handling than the workloads I use, the critical

4. TLB miss penalty should include the effects of cache and TLB misses within the miss handler.

Table 4-3: Effect of preloading in complete-subblock TLBs

TLB Type	#blocks	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
		Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty
Fully-Associative	64	1.05	1.33	1.06	1.65	1.02	1.41	1.04	2.05
	128	1.02	1.26	1.01	1.20	1.02	1.36	1.02	2.85
4-way Set-Associative	256	1.01	1.08	1.01	1.21	1.01	1.39	1.00	4.03
	512	1.00	1.09	1.01	1.16	1.00	2.32	1.00	1.45

miss penalty may be much smaller.

There are at least three disadvantages to preloading. First, applications with little spatial locality do not benefit from preloading but still pay higher TLB miss costs. Second, preloading is much more expensive in some page table organizations where mappings of neighboring pages in a page block are not adjacent, *e.g.*, hashed page tables. This increases the TLB miss penalty significantly and may be larger than the critical miss penalty. In Chapter 7, I describe how different page tables are suited for preloading and propose one that has advantages for preloading. Third, sometimes a base page becomes valid after the page block's mappings were preloaded into the TLB—at the time of preloading all base pages within a page block need not have valid mappings. Blindly preloading can result in multiple copies of base page mappings in a TLB. Appendix D describes how to handle this.

4.3 Sample design given area constraint

A computer architect, given a fixed TLB chip area, can choose between different single-page-size, superpage, and complete-subblock TLB configurations. In this section, I consider alternate fully-associative TLBs that occupy comparable chip area and try to find the best TLB architecture for various chip area budgets, as predicted by the area model described in Section 2.2. I compare the different TLBs using as metrics TLB access time, with the timing model described in Section 2.3, and normalized execution time speedup.

I consider four area budgets—the area required to implement fully-associative single-page-size TLBs of 64, 128, 256, and 512 blocks—and calculate the number of TLB blocks of complete-subblock TLBs with different subblock factors and superpage TLBs that will fit in the same area[5] (Table 4-4). The superpage TLBs support only two page sizes—the base page size (4KB) and a superpage size eight times larger (32KB). An interesting observation from Table 4-4 is that the superpage and complete-subblock TLBs have *fewer* TLB blocks than the single-page-size TLBs but have a significantly *larger* TLB reach. A 92-block complete-subblock TLB with subblock factor 8, for example, requires comparable area to a 256-block single-page-size TLB but has nearly three times larger TLB reach of 736 base pages.

I calculate the TLB access time for the different TLBs (Table 4-4) compared to the access time for a single-page-size TLB of comparable chip area. The superpage TLBs have nearly equal access times. The complete-subblock TLBs may have a shorter access time because they have fewer blocks and shorter tags. They may have a larger access time due to a larger data RAM and a subblock multiplexor. The important observation here is that it is possible to build

5. Since a TLB cannot have fractional number of blocks or fractional subblock factor, the TLB size chosen has an area closest to the budget.

a subblock TLB that has a faster access time than an equivalent single-page-size TLB. This is important since microprocessor designers have increasingly larger area available to them but cannot build large single-page-size TLBs due to slow access times. Further, complete-subblock TLB access time can be improved over the monolithic memory design I assume.

Table 4-4: Access time for fully-associative TLBs relative to single-page-size (4KB) TLB of equal area

Area (rbe)	Single-page-size TLB		Superpage TLB (32KB)		Complete-subblock TLB (subblock factor)							
					2		4		8		16	
	#blocks	Access Time	#blocks	Access Time	#blocks	Access Time	#blocks	Access Time	#blocks	Access Time	#blocks	Access Time
7984	64	1.00	62	1.00	51	1.01	35	1.03	20	1.06	9	1.13
15298	128	1.00	123	1.00	102	1.00	72	1.01	44	1.04	23	1.10
29928	256	1.00	247	1.00	206	0.99	147	0.99	92	1.00	51	1.05
59186	512	1.00	494	0.99	413	0.96	297	0.92	188	0.90	107	0.93

TLB performance is next metric to consider as access time alone does not dictate better performance. Table 4-5 shows the normalized execution time speedup for the TLBs shown in Table 4-4 relative to the respective single-page-size TLB. Note that I assume superpage TLBs and complete-subblock TLBs with preloading have the same TLB miss penalty as a single-page-size or a complete-subblock TLB without preloading.

Table 4-5: Execution time speedups relative to single-page-size (4KB) TLBs of equal area

Area (rbe)	Superpage TLB (4KB/32KB)		Complete-subblock TLB (NO preloading)				Complete-subblock TLB (with preloading)			
	Using superpages	Using base pages	2	4	8	16	2	4	8	16
7984	1.18	1.00	1.03	1.04	1.01	0.81	1.08	1.12	1.13	0.98
15298	1.13	1.00	1.02	1.04	1.05	1.03	1.06	1.09	1.10	1.12
29928	1.07	1.00	1.02	1.03	1.01	1.00	1.04	1.05	1.03	1.04
59186	1.03	1.00	1.00	1.00	1.00	0.99	1.01	1.01	1.01	1.01

Six important observations should be made from the above performance comparison using a fixed chip area budget. First, superpage TLBs have clearly the best performance as they have the largest TLB reach—almost eight times that of the single-page-size TLBs—and can load mappings to eight base pages on a single TLB miss. However, to make effective use of the TLB reach, operating system support is essential. If the operating system does not use superpages, the superpage TLB performance degenerates to that of a fully-associative single-page-size TLB with *fewer* TLB blocks.

Second, a complete-subblock TLB of small subblock factor (two or four) is preferable to building a single-page-size fully-associative TLB. The subblock TLBs have comparable or better access time but better TLB performance. This is an important observation because few TLB designs use subblocking today. The performance is better in spite of having fewer TLB blocks in the complete-subblock TLBs as workloads have sufficient spatial locality to exploit their larger TLB reach.

Third, for a large area budget, complete-subblock TLBs, of any subblock factor, are clearly a

better choice than building large single-page-size TLBs (*e.g.*, 512 blocks) that may have an unacceptably large access time. Complete-subblock TLBs fit in very well here by providing a design that not only has a shorter access time but also a larger TLB reach and better TLB performance! A 188-block complete-subblock TLB with subblock factor 8, for example, is 10% faster to access and has three times the TLB reach of a 512-block single-page-size TLB.

Fourth, use of very large subblock factors (16) may result in worse TLB performance and slower TLBs if the limited chip area allows a TLB with very few blocks. Though a 9-block subblock TLB with subblock factor 16 has a TLB reach of 138 pages, it performs worse than a 64-block single-page-size TLB with TLB reach of 64 pages. Nine blocks are often not enough to span the working set of many programs. With a large area budget, however, even the larger subblock factors are effective. Thus, it is important that the subblock factor chosen does not severely limit the number of blocks.

Fifth, complete-subblock TLBs are attractive compared to single-page-size TLBs, and preloading in the TLB miss handler only makes them more attractive. Preloading requires simpler hardware support for TLB miss handling and results in fewer TLB misses. Preloading, however, may increase the TLB miss penalty but the effect is small if the page table stores mappings for a page block contiguously in memory. The last four columns of Table 4-5 assume preloading in the TLB miss handler.

Sixth, complete-subblock TLBs with preloading significantly reduce the number of TLB misses, and have speedups that are close to those with superpage TLBs. This is important because superpage TLBs require substantial operating system support and introduce other overheads. Complete-subblock TLBs offer a competitive hardware solution. The additional gains from using medium-sized superpages may not justify the need to modify operating systems. Large superpage mappings can still be supported in complete-subblock TLBs as explained in Appendix E.

In this example, assuming no operating system support for medium size superpages, I would choose a complete-subblock TLB design with subblock factor of 4. Preloading should be used in the TLB miss handler if the page table stores mappings for a page block contiguously in memory—four base page mappings would typically fit in a single cache line. In the presence of operating system support, superpage TLBs are more effective.

4.4 Comparison with other TLB architectures of same TLB reach

This section compares the chip area, access time, and TLB performance of complete-subblock TLBs to single-page-size and superpage TLBs of equal TLB reach.

4.4.1 Complete-subblock vs. single-page-size TLBs

This section compares two brute force ways to increase TLB reach—more TLB blocks or complete-subblocking, *e.g.*, a 64-block complete-subblock TLB with subblock factor 4 and a 256-block single-page-size TLB have equal TLB reach. The complete-subblock TLB incurs more TLB misses but is attractive as it occupies less chip area and has a smaller access time.

- Worse TLB performance: Though the TLBs have identical TLB reach, the complete-subblock TLB cannot always fully use the TLB reach due to less than optimal spatial locality. An n-block single-page-size TLB can map any n independent pages whereas an n-block subblock TLB can map n pages from only n/s page blocks. Table 4-6 compares the performance of com-

plete-subblock and single-page-size TLBs with equivalent TLB reach. There are three observations to make: First, halving the number of TLB blocks does not double the number of TLB misses as the programs exhibit spatial locality and use the larger area mapped by individual TLB blocks (Appendix J). Second, for small subblock factors the performance degradation is reasonable. Large subblock factors, however, result in a TLB with very few tags. A 16-block TLB with subblock factor 16 has the same TLB reach as a 256-block TLB, for example, but has much worse performance. A minimum number of tags are necessary to capture the working set and 16 tags are not sufficient. Third, preloading in the TLB miss handler helps the subblock TLBs with fewer tags to perform comparable to the much larger single-page-size TLBs.

Table 4-6: Execution time speedups for complete-subblock TLBs relative to single-page-size (4KB) TLBs with same TLB reach

TLB type	Single Page Size (4KB)		Complete-Subblock (subblock factor)				Complete-Subblock with preloading (subblock factor)			
			2	4	8	16	2	4	8	16
	#blocks	N	N/2	N/4	N/8	N/16	N/2	N/4	N/8	N/16
Fully-Associative	256	1.00	0.96	0.94	0.92	0.88	0.98	0.99	1.00	1.00
	512	1.00	0.99	0.96	0.95	0.92	0.99	0.97	0.97	0.98
4-way Set-Associative	256	1.00	0.98	0.96	0.93	0.87	1.00	1.01	1.00	0.97
	512	1.00	0.99	0.98	0.96	0.94	0.99	0.99	0.99	1.00

- Smaller chip area: The complete-subblock and single-page-size TLBs with same TLB reach store nearly identical number of data bits[6] and have similar data RAM sizes. A complete-subblock TLB's data array, being a wider and thinner rectangle, has higher driver/sense amp overhead. The complete-subblock TLB however has s times fewer tags and, further, the tags are lg(s) bits shorter. In fully-associative TLBs, the savings in tag memory CAM cells is significant. In set-associative implementations, the effect of reduction in tag memory is smaller—the savings from RAM cells is smaller and the tags were a smaller fraction of the overall area. There is a small increase in area for large subblock factors and small number of tags due to the fixed overheads that become significant as the data RAM becomes wider and thinner.

Table 4-7: Chip Area and Access Time for complete-subblock TLBs relative to single-page-size TLBs with same TLB reach

TLB type	Single Page Size		Chip Area				Access Time			
			2	4	8	16	2	4	8	16
	#blocks	N	N/2	N/4	N/8	N/16	N/2	N/4	N/8	N/16
Fully-Associative	256	1.00	0.63	0.46	0.39	0.39	0.94	0.91	0.92	0.96
	512	1.00	0.63	0.44	0.36	0.34	0.86	0.81	0.80	0.83
4-way Set-Associative	256	1.00	0.79	0.73	0.81	1.05	0.99	0.99	1.00	1.08
	512	1.00	0.76	0.66	0.67	0.79	0.94	0.93	0.96	1.01

- Faster access time: The smaller complete-subblock TLBs are also faster to access. The access time reduces as the smaller, shorter, tag array results in a faster tag access time, which is always on the critical path. In fully-associative subblock TLBs, the larger data RAM array is slower and negates some benefits of the faster tag access. In set-associative TLBs the access time does not improve—though the critical tag access and compare times reduce—due to an

6. An n-block complete-subblock TLB of subblock factor s stores $n*s$ data bits (the subblock valid bits) more than a $n*s$-block single-page-size TLB.

increased delay in driving the multiplexor output across a much wider data RAM (as noted in Section 4.1.2, this can be optimized further).

To summarize, a complete-subblock TLB with subblock factor *s* is more effective than building a single-page-size TLB with *s* times as many TLB blocks. A complete-subblock TLB has a faster access time, occupies less area, and offers competitive TLB performance. Further, complete-subblock TLBs with preloading have comparable TLB performance to the much larger, slower, single-page-size TLBs—a win-win situation.

4.4.2 Complete-subblock vs. Superpage TLBs

In Chapter 3, I proposed medium-sized superpages as one way to increase TLB reach if the operating system can do proper page-size assignment to use superpage mappings. Complete subblocking is a brute force way of increasing TLB reach without depending on any operating system support. For the same number of TLB blocks, a complete-subblock TLB, with subblock factor *s*, and a superpage TLB, supporting a single superpage size of *s* * base page size, have the same TLB reach. The complete-subblock TLB requires significantly larger area and slower access time but has better TLB performance. The area and access time arguments are similar to those in Section 4.1.2, because a superpage TLB occupies nearly equal area and has similar access time to a single-page-size TLB with the same number of blocks (Table 3-1). Table 4-8 summarizes the area and access time comparisons for superpage and complete-subblock TLBs with the same TLB reach. The set-associative superpage TLBs use the superpage index (Section 3.2.2).

Table 4-8: Chip Area and Access Time for complete-subblock TLBs relative to superpage TLBs

Superpage TLB Type	Chip Area subblock factor:superpage size				Access time subblock factor:superpage size			
	2:8KB	4:16KB	8:32KB	16:64KB	2:8KB	4:16KB	8:32KB	16:64KB
(64-256_ block Fully-associative	1.22	1.67	2.58	4.39	1.02	1.06	1.11	1.20
256-block 4-way Set-associative	1.38	2.14	3.67	6.71	1.00	1.00	1.03	1.10

The advantages of complete-subblock TLBs over superpage TLBs are:

Better TLB performance: Complete-subblock TLBs can deliver better performance than the already significantly improved TLB performance of superpage TLBs. Table 4-9 compares the complete-subblock and superpage TLB performance. Complete-subblock TLBs without preloading perform worse than superpage TLBs. They often have more TLB misses than for a superpage TLB. A superpage TLB loads all the mappings for a page block in a single TLB miss, and a complete-subblock TLB takes multiple TLB misses to do so. Preloading addresses this shortcoming. A complete-subblock TLB with preloading always incurs fewer TLB misses than a superpage TLB with the same TLB reach, number of blocks, and associativity (Appendix J). However, since the superpage TLBs reduce TLB miss handling time significantly already, the incremental benefit of complete-subblocking is small for these workloads. In workloads where superpages cannot be used for all of the address space, *e.g.*, due to difference in attributes or length of segments, complete-subblock TLBs can still share a single TLB block for multiple base pages within a page block.

No operating system support: Except in the TLB miss handler, a complete-subblock TLB

Table 4-9: Execution time speedups for complete-subblock TLBs relative to superpage TLBs

TLB Type	#blocks	subblock factor: superpage size				With preloading subblock factor: superpage size			
		2:8KB	4:16KB	8:32KB	16:64KB	2:8KB	4:16KB	8:32KB	16:64KB
Fully-Associative	64	0.96	0.95	0.98	0.97	1.00	1.00	1.00	1.00
	128	0.98	0.99	0.98	0.98	1.00	1.00	1.00	1.00
4-way Set-associative	256	1.00	0.99	1.00	1.07	1.00	1.00	1.01	1.07

does not require additional operating system support. The superpage TLBs require substantial support from the operating system. This is important since a processor may have to run old software that may not support superpages—complete-subblock TLBs can better use their TLB reach advantages.

Less I/O: Workloads run on complete-subblock TLB systems do the same I/O operations as in a single-page-size TLB system, as the operating system is the same. Systems with superpage TLBs will transfer more data to backing store as the operating system accepts some fragmentation in return for better TLB performance. However, even with more data is transferred in a superpage system, there might be fewer I/O operations as I/O operations for multiple base pages can be clustered.

Reduced Page Fault Service Time: Again, systems with complete-subblock TLBs have the same page fault latency as in a single-page-size system. Superpages can take longer to initialize and/or transfer from backing store, increasing the page fault service time. Subblock TLBs can result in a better overall execution time.

Subblock caches allow a portion of a cache line to be accessed before completely fetching the full cache line from memory [Hill86, Hill87]. Instruction caches use this feature to reduce hit time, often combined with a fetch policy that brings the referenced word first from memory. Subblock TLBs can similarly exploit this feature to reduce the page fault latency (not TLB miss penalty) for superpages by using the operating system to implement the following policy for servicing superpage page faults: The operating system initiates I/O for the superpage with I/O for the referenced page first from backing store. The process resumes after I/O to the first base page is complete while the rest of the superpage loads into memory in the background (similar to the cache example above). A subsequent page promotion finally results in storing a superpage mapping in the page table. A superpage TLB will use multiple base page TLB blocks for a partially-filled superpage while the I/O is in progress, whereas a subblock TLB will continue to share a single TLB block for the all mappings within a partially filled superpage.

Reference and Modified information granularity: Complete-subblock TLBs store a full mapping for every base page and store reference or modified information at the granularity of a base page size. Superpage TLBs can only store such information at the granularity of superpage size for superpage mappings. The finer granularity results in better page replacement decisions and reduces the number of dirty pages written to backing store in complete-subblock systems.

In summary, complete-subblock TLBs offer better TLB and overall system performance by exploiting spatial locality more effectively and providing for more efficient operating system implementations than medium-size superpages. However, complete-subblock TLBs occupy

larger chip area and have a slower access time. In the next chapter, I address these disadvantages by proposing partial-subblock TLBs.

4.5 Conclusion

Subblocking has long been used for caches and this thesis shows that subblocking improves the performance of TLBs also. A complete-subblock TLB associates with a page block a single tag but allows for storage of separate mappings for base pages within the page block. Spatial locality in programs helps subblock TLBs incur fewer misses than a purely random access pattern would predict.

A complete-subblock TLB is more complicated to build than a single-page-size TLB. However, it does not require new implementation technologies—cache and RAM designers have long used the techniques required for subblock TLBs. Superpage TLBs provide a cheap way for the hardware to increase TLB reach but shift the burden of exploiting it to software—complete-subblock TLBs are more hardware-centric, requiring no additional operating system support.

A complete-subblock TLB presents yet another win-win situation as superpages do—a complete-subblock TLB has a larger reach and better performance and yet has a shorter access time than a single-page-size TLB. Processor designers have an increasing amount of chip area and transistors available but are unable to build larger TLBs due to cycle time constraints. The key contribution of this chapter is that it shows that complete-subblock TLBs can use the extra transistors to increase TLB reach without increasing the access time—especially in set-associative designs.

Chapter 5 Partial-subblock TLBs

This chapter proposes and evaluates a new TLB architecture, *partial subblocking*, that combines the low implementation cost of medium-sized superpage[1] TLBs and simpler operating system support by borrowing subblock valid bits from complete-subblocking. A partial-subblock TLB has an implementation complexity comparable to that of a superpage TLB, requires less operating system support than medium-sized superpages but still incurs fewer misses than a superpage TLB. The main contribution of this chapter is that it shows that partial-subblock TLBs have the best TLB performance compared to alternate single-page-size, superpage, and complete-subblock TLBs that occupy similar chip area. All three new architectures allow mappings for multiple base virtual pages within a virtual page block to share a single TLB block—but conditions under which mappings are considered compatible for sharing differ (Table 1-2 in Chapter 1).

A partial-subblock TLB block's tag maps a fixed page block size, like a complete-subblock TLB block. Multiple base page mappings share a single PPN and attribute field in the data but have individual subblock valid bits[2] for the base pages. Two or more base virtual pages share a single partial-subblock TLB block if they belong to the same virtual page block, are properly placed (Table 1-1) in physical memory, and have the same attributes. A partial-subblock TLB allows incompatible or unaligned mappings in the TLB but they use different TLB blocks. A partial-subblock TLB (with preloading) incurs fewer TLB misses while requiring simpler operating system support than superpage TLBs. Partial-subblock TLBs occupy much smaller area than complete-subblock TLBs but incur comparable number of TLB misses (Section 5.6.3).

Two base page mappings can share the same partial-subblock TLB block if they have the same attributes and are properly placed. With subblock factor *s*, base pages x and y are properly placed if they are placed in the same virtual and physical page blocks (`PPN(x) div` *s* = `PPN (y) div` *s* and `VPN(x) div` *s* = `VPN(y) div` *s*, where `div` is integer division) and are both page block aligned (`VPN(x) mod` *s* = `PPN(x) mod` *s* and `VPN(y) mod` *s* = `PPN(y) mod` *s*, where `mod` is integer modulus operation). Mappings that are not properly placed are allowed, but in separate TLB blocks that can reside in the TLB simultaneously. Partial-subblock TLB blocks store unaligned mappings (`VPN(x) mod` *s* ≠ `PPN(x) mod` *s*) by setting the SB attribute to 0 to disable subblocking.

Figure 5-1: Format of a partial-subblock TLB block

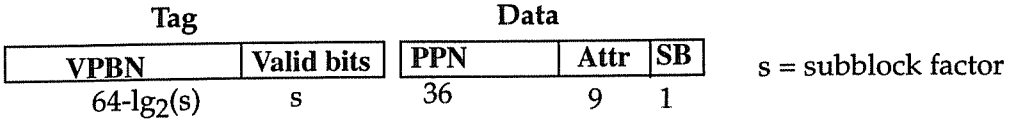


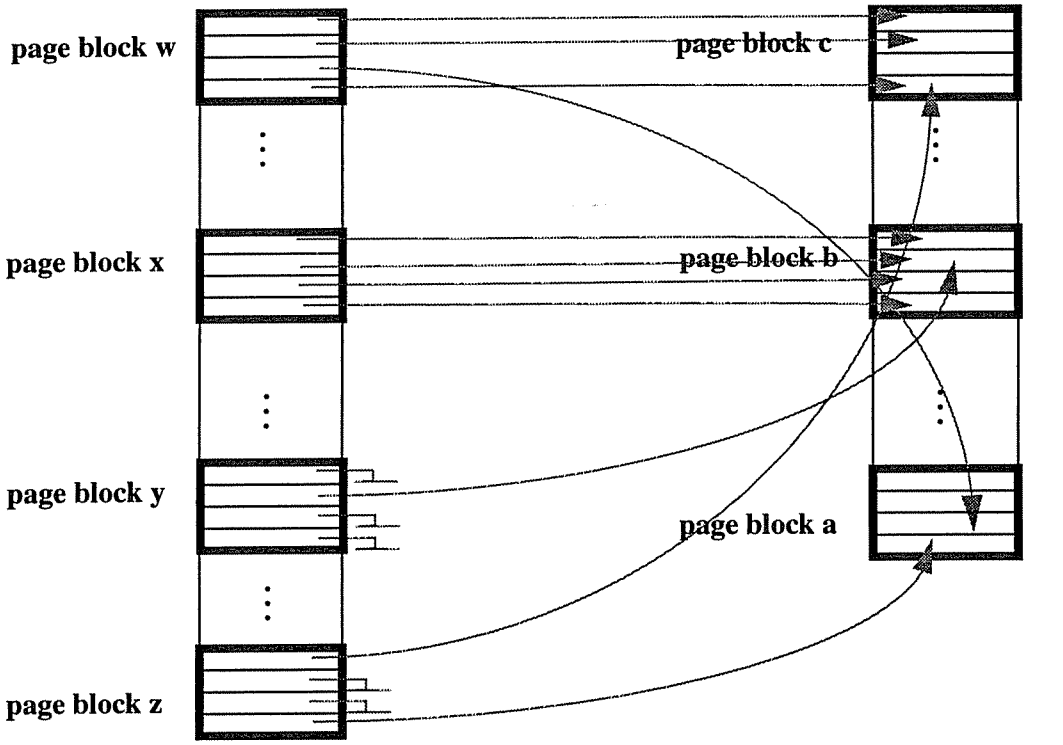
Figure 5-1 shows the format of a partial-subblock TLB block. The tag stores a virtual page block number (VPBN) and *s* valid bits for *s* individual base pages within the page block specified by VPBN. The *i*th valid bit set (✔) or clear (✕) shows whether the TLB block has a valid mapping for the corresponding base virtual page, (VPBN + *i*). The data portion stores a single physical page number (PPN) and an attribute field. A subblock attribute bit (SB) is set to enable subblocking. When SB is clear, a TLB block stores a single base page mapping that may or

1. This chapter concentrates on partial-subblock TLBs as an alternative to medium-sized superpages. Appendix H adds support for large superpages.
2. Variations of partial-subblock TLBs that replicate other fields of the mapping are discussed in Section 5.7.

may not be page block aligned. This feature permits partial-subblock TLBs to be used with an operating system unaware of partial subblocking.

Figure 5-2 illustrates how a partial-subblock TLB stores base page mappings and brings out the important properties of partial-subblock TLBs. First, the mappings from virtual-page block x to physical page block b could use a superpage mapping, also use a single partial-subblock TLB block. A partial-subblock TLB will use a single TLB block for mappings that could have used superpage mappings—thus removing the need for supporting medium-size superpages. A partial-subblock TLB always uses less or same number of blocks than a superpage TLB to map an address space.

Figure 5-2: Virtual Address to Physical Address Mappings in a Partial-Subblock System



VPBN	Valid
x	✓✓✓✓
w	✓✓X✓
w	XX✓X
y	X✓XX
z	XXX✓
z	✓XXX

PPN	Attr	SB
b00	Attr	1
c00	Attr	1
a00	Attr	1
b00	Attr	1
a00	Attr	1
c01	Attr	0

Partial-subblock TLB storing mappings for above

Second, a partial-subblock TLB can have multiple TLB blocks cached in the TLB that have identical VPBNs but disjoint valid bits, *e.g.*, the mappings from page block w to physical page blocks a and c. The mappings for base pages 0,2,3 share a single TLB block as they are properly placed (assuming identical attributes). The mapping for page 1 cannot share the same TLB

block as it is placed in a different physical page block. Such a situation frequently occurs in operating systems that use the copy-on-write optimization [Rash88].

Third, a partial-subblock TLB block can be incrementally populated, *e.g.*, the mapping from page block **y** uses a TLB block that also could store another properly placed mapping (if established) from page block **y** to page block **b**. Thus, if the operating system properly places pages in partially-populated page blocks, mappings established later can share a single TLB block. Another option is to perform gather operations when adding new mappings, as some superpage systems do during page promotion (Section 6.2.2).

Fourth, a mapping that is not page block aligned has the SB bit clear and cannot share the TLB block with any other page, *e.g.*, the mapping from page **3** of page block **z**. Unaligned mappings differ from aligned mappings in the way the TLB generates physical addresses for them, as discussed in Section 5.1.1.

Fifth, partial-subblock TLBs treat differences in attributes similar to improperly placed pages—different TLB blocks store the mappings. If pages **0** and **1** of page block **x**, for example, had a different attribute from the rest of the page block, then they would share one TLB block, while pages **2** and **3** would share another.

The most important feature of partial-subblock TLBs is the presence of subblock-valid bits. This allows compatible base pages within a page block to share a single TLB block while other base pages in the page block may be improperly placed or unmapped, *e.g.*, small objects or objects that do not start or end at a page block boundary. Subblock-valid bits relieve the operating system of the need to implement page promotion and a page-size assignment policy. This feature helps partial-subblock TLBs deliver comparable or better speedups to superpage TLBs but only require the operating system to make a *best-effort* at page placement without providing *guarantees* required for superpages.

To support the above feature a partial-subblock TLB must allow for multiple TLB blocks with the same VPBN, but disjoint subblock-valid bits, to be present in the TLB simultaneously. Disallowing multiple TLB blocks with the same tag can result in a significantly worse TLB performance and may livelock[3]. Single-page-size, superpage, or complete-subblock TLBs do not require support for multiple blocks with same VPNs[4].

In the following sections, I describe how a partial-subblock TLB works, discuss implementation alternatives to simplify the hardware, discuss techniques to handle TLB misses without increasing TLB miss penalty, compare alternate TLBs given a fixed chip area, compare partial-subblock TLBs with alternate single-page-size, superpage, and complete-subblock TLBs of equal TLB reach, and list some possible variations of partial-subblock TLBs.

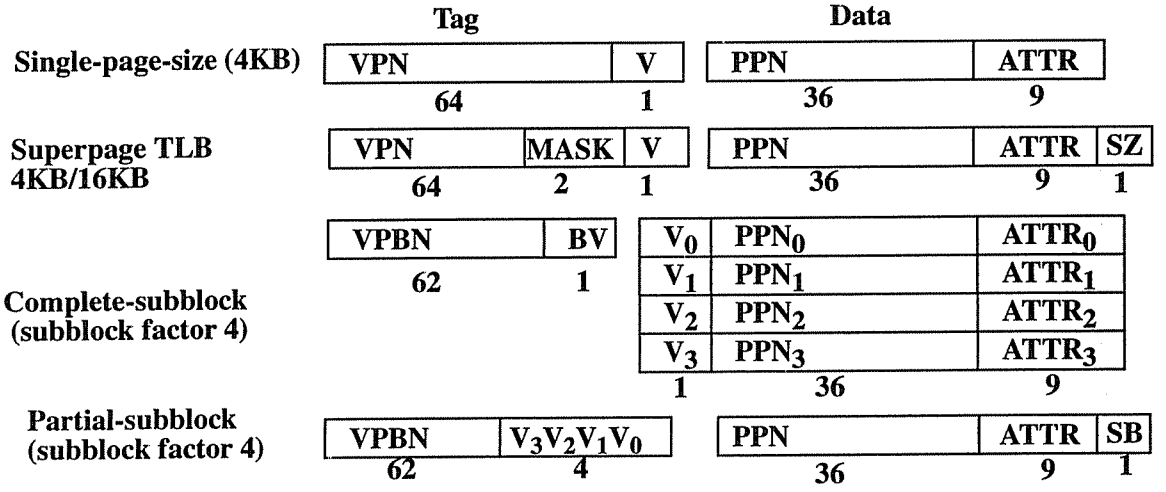
5.1 Mechanics of a Partial-subblock TLB

A complete-subblock TLB uses spatial locality in programs to deliver TLB performance competitive to a non-subblock TLB with independent tags—the benefit is a significantly smaller and faster tag memory. A partial-subblock TLB extends this using the operating system to

3. If a program attempts to execute an instruction on page **0** to read data from page **1**, processor implementations may require that the TLB hold mappings to both pages. If the mappings are incompatible, they use different partial-subblock TLB blocks with the same tag. If the TLB disallows this, the program will livelock. All workloads in my simulations did livelock. Separate instruction and data TLBs would avoid livelock in this example.
4. Operating system software or TLB hardware must guarantee that two TLB blocks cannot have the same VPN.

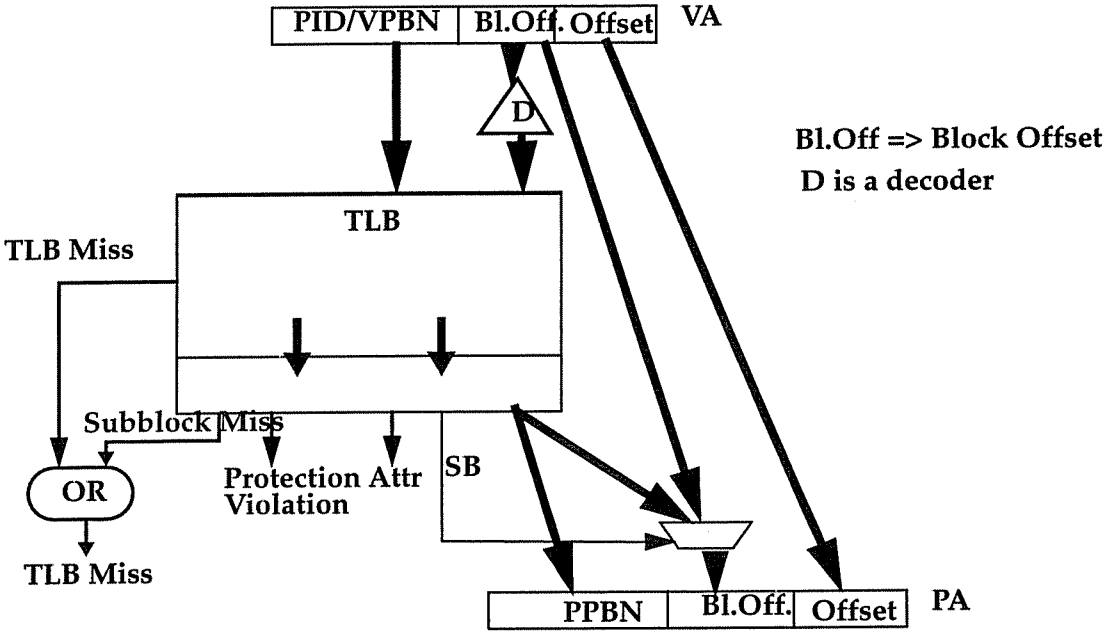
properly place pages in physical memory and delivers TLB performance competitive to a complete-subblock TLB but with a significantly smaller and faster data memory.

Figure 5-3: TLB blocks for different TLB architectures



A partial-subblock TLB differs from the basic structure of a complete-subblock TLB (Section 4.1) in three ways. First, the TLB block stores only a single PPN and Attr field, as shown in Figure 5-3, and the data RAM is comparable in size to that in a single-page-size TLB. Second, supporting multiple blocks with identical VPBNs requires the valid bits to be part of the tag and there is no block-valid bit. Third, support for unaligned mappings requires a subblock attribute bit (SB).

Figure 5-4: Structure of a partial-subblock TLB



Implementing a partial-subblock TLB requires two changes to a single page-size TLB (Figure 1-3). First, physical address generation requires a multiplexor to support unaligned mappings, discussed in Section 5.1.1. Second, similar to a complete-subblock TLB, the decoded block offset field of the virtual address selects the appropriate subblock valid bit, but as dis-

cussed in Section 5.1.2, the complete-subblock TLB solution of using a block-valid bit does not work. Figure 5-4 shows the basic structure of a partial-subblock TLB.

5.1.1 Physical Address Generation in a partial-subblock TLB

Calculating the physical address from the selected mapping is trivial in single-page-size or complete-subblock TLBs—the page offset from the virtual address appends to the PPN. In superpage TLBs, a multiplexor selects bits from either the PPN or the virtual address based on the page size of the selected mapping. In partial-subblock TLBs, physical address generation depends on whether the selected mapping is page block aligned.

For page block aligned mappings (SB=1), both the block offset and page offset fields of the virtual address append to the physical page block number (PPBN) from the mapping to produce the physical address. If all mappings are page block aligned, a partial-subblock TLB block need store only the PPBN and neglect the low-order bits of the PPN.

However, a partial-subblock TLB block must store the complete PPN to allow for unaligned mappings (SB=0). In an unaligned mapping the block offset fields of the virtual and physical page numbers are not equal, *e.g.*, VPN = 0x5f0 and PPN = 0x891 with subblock factor 16. Using the block offset field from the virtual address will generate an incorrect PPN (0x890), instead the TLB stores the complete PPN. Unaligned mappings occur if operating systems do not (or cannot) allocate aligned physical pages.

A naive solution requires the operating system guarantee that physical memory allocation always result in page block aligned mappings. The TLB then uses the trivial physical address generation technique described above for aligned pages, there is no subblock attribute bit, and only the PPBN is stored in the TLB block. This solution, however, is impractical. It turns physical memory into a set-associative cache of pages and has a higher page fault rate than in the default fully-associative mode that can have unaligned mappings. While operating systems may create page block aligned mappings in the common case, it is inefficient to guarantee page block aligned mappings. Further, some UNIX APIs allow users to establish unaligned mappings that the naive solution does not support.

Another solution uses the subblock attribute bit (SB) to control a multiplexor. The multiplexor selects the block offset bits from either the PPN read from the TLB block or from the block offset field of the virtual address (Figure 5-4). This requires the TLB block to store a full PPN—a small cost of $\log_2(s)$ extra RAM bits. Also, the multiplexor adds to TLB access time in fully-associative TLBs. In set-associative implementations where the data RAM access is not on the critical path, the multiplexor may not affect TLB access time.

5.1.2 Subblock-valid bits in a partial-subblock TLB

A partial-subblock TLB block has multiple subblock-valid bits and the appropriate one must be selected to determine a TLB hit—just as in a complete-subblock TLB. Complete-subblock TLBs could use a block valid bit in the tag and store subblock valid bits in the data RAM. A partial-subblock TLB cannot use the block-valid bit technique as it does not allow multiple TLB blocks with identical tags to reside in the TLB (Appendix B).

As discussed in Appendix B there are two ways to implement subblock-valid bits in partial-subblock TLBs. The TLB simulation results do not differ between the two subblock valid bit implementations, but differ in area and access time characteristics.

The first alternative includes subblock valid bits in the tag memory and extends the tag compare logic—the CAM array in fully-associative TLBs or the tag comparator in set-associative TLBs—to compare subblock-valid bits with the decoded block offset field of the virtual address. The second alternative uses a separate valid bit RAM that is read in parallel with tag comparison. The selected valid bit combines with the tag match signal as part of the wordline or multiplexor drivers. In set-associative TLBs, the subblock valid bits can be stored in the data array itself.

In this thesis, the area and access time models assume the valid bit tag comparator implementation. The valid bit RAM implementation is faster and occupies a smaller area (as explained in Appendix B). This makes my results pessimistic for partial-subblock TLBs. However, I still show that partial-subblock TLBs are faster and more effective than other TLB architectures and a faster implementation only makes them more attractive.

Figures 5-5 and 5-6 show fully-associative and set-associative implementations of partial-subblock TLBs respectively. They differ from the complete-subblock implementations in Appendix B in two ways. First, they do not require column multiplexors as the data stores a single mapping. Second, a multiplexor selects the physical block offset bits based on the subblock attribute bit read out of the RAM.

Figure 5-5: Fully-associative Partial-subblock TLB

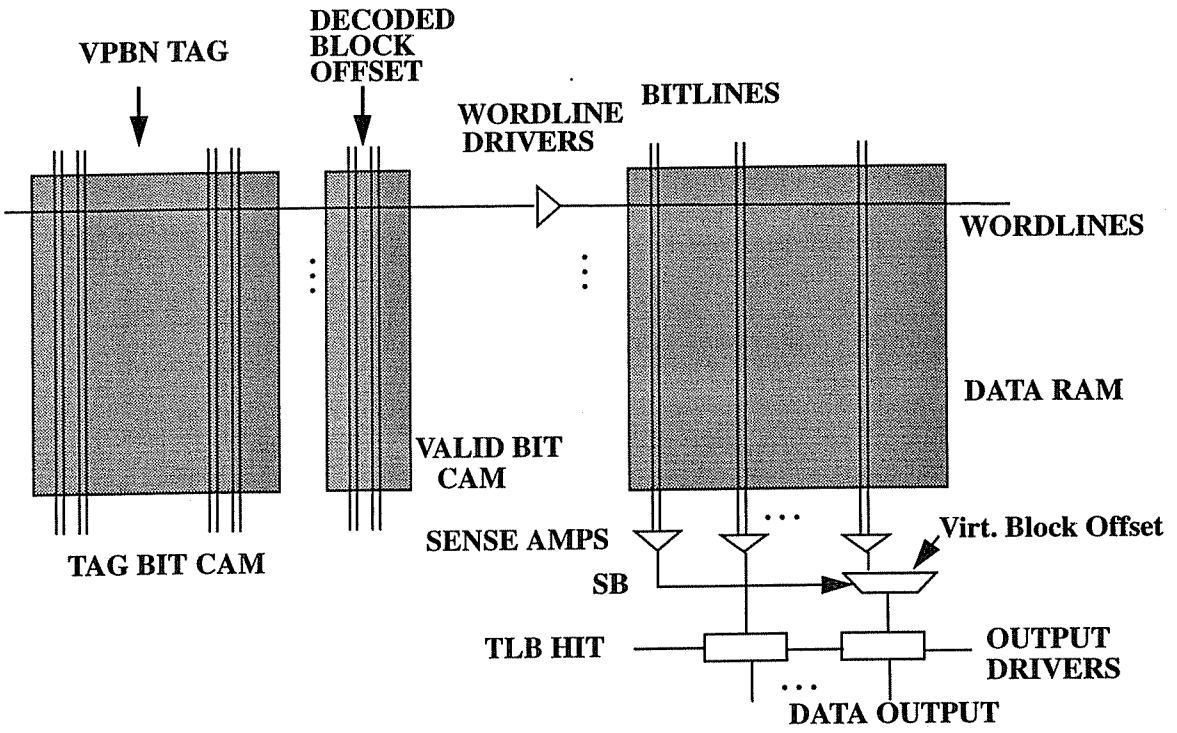
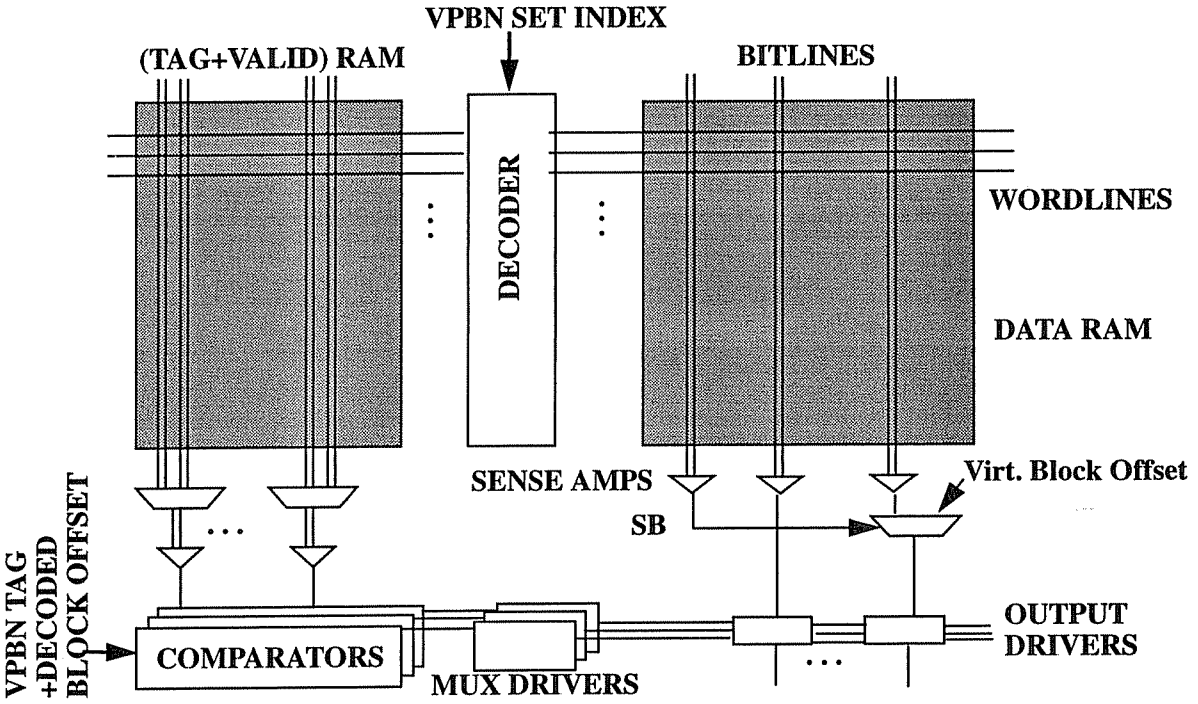


Figure 5-6: Set-associative Partial-Subblock TLB



5.1.3 *Modified Bits Update*

Modified bits mark dirty pages that the operating system has to update in backing store. The TLB miss handler sets modified bits in the page table on **mod_bit_faults**. A modified bit is one of the attribute bits and, in a partial-subblock TLB, changing the modified bit for a base page affects the compatibility of the mappings sharing a TLB block. There are at least three ways to handle modified bit update[5] in a partial-subblock TLB.

First, a partial-subblock TLB block could store subblock-modified bits for each base page and mappings can continue to share a TLB block even after the TLB miss handler updates one of the modified bits. In some systems, the write-permission bit emulates the modified bit and providing subblock-modified bits can reduce the number of TLB misses by making both read-only and read-write mappings compatible to share a single partial-subblock TLB block. This, however, increases the chip area and may require wider page table entries.

Second, a partial-subblock TLB block could store a single modified bit in the attributes—as the TLB block format in Figure 5-1 assumes—with clean and dirty base pages using separate partial-subblock TLB blocks. This has the advantage that only a single modified bit need be stored in the TLB but has the disadvantage that the number of TLB misses increases as more TLB blocks are needed to store the same mappings. However, TLB simulations show that the increase in the number of TLB misses is negligible[6].

Third, a partial-subblock TLB block stores a single *block-modified bit* that is set if the pro-

5. The modified bit is special because it is an attribute that is updated by the TLB miss handler. Other attributes are set by the operating system but not updated by the TLB miss handler. The referenced bit is also updated by the TLB miss handler but can be emulated using the valid bits.

6. My simulations run with sufficient memory and run to completion without paging. There will a greater difference in TLB performance when short of memory.

gram writes to any base page mapped by the TLB block, *i.e.*, marks *all* base pages sharing the TLB block as dirty. This has the disadvantage of a coarser granularity than a base page size and could result in an increase in the number of dirty pages written to backing store. The advantage is that some programs exhibit spatial locality in writes—pages close to a recently written page are likely to be written soon—and setting the modified bit on the first mod-bit fault avoids later mod-bit faults for other pages in the page block.

The simulations in this chapter assume the block-modified bit option. If there are unused bits in the PTE format, subblock modified bits may be preferred. However, further study is needed to evaluate the tradeoff between the decrease in mod-bit faults and the increase in backing store I/O when using the block-modified bit.

5.2 *Effect of Partial subblocking*

An n-block partial-subblock TLB significantly increases the TLB reach of an n-block single-page-size TLB but only occupies a slightly larger area and has comparable access time. Partial-subblock TLBs, similar to superpage TLBs, depend on operating system support to exploit the increased TLB reach and achieve good TLB performance. A partial-subblock TLB adds to a single-page-size TLB block multiple valid bits (s bits), an extra attribute bit (SB) and a multiplexor for physical address generation but stores $\log_2(s)$ fewer VPN bits.

Partial-subblock TLBs significantly reduce the number of TLB misses through effective use of a larger TLB reach. With proper placement of physical pages by Foxtrot, Table 5-1 shows the normalized execution time speedups when using partial-subblock TLBs relative to using single-page-size TLBs with the same number of TLB blocks and associativity.

Table 5-1: Execution time speedup with partial-subblock TLBs relative to single-page-size (4KB) TLBs with same number of blocks

TLB Type	#blocks	subblock factor			
		2	4	8	16
fully-associative	64	1.04	1.09	1.16	1.17
	128	1.05	1.10	1.11	1.15
4-way set-associative	256	1.05	1.08	1.10	1.12

Both fully-associative and set-associative partial-subblock TLBs show a speedup for my workloads and larger subblock factors result in better performance. However, set-associative TLBs with subblock factors greater than the set-associativity sometimes show a slowdown (Table I5-1c in Appendix I) due to an increase in conflict misses. When pages within a page block have incompatible mappings that cannot share a single partial-subblock TLB block, the base pages all map to the same TLB set in a partial-subblock TLB. Associativity helps accommodate the multiple mappings in the same set but if the subblock factor is greater than the associativity, it can cause an excessive number of conflict misses.

Table 5-2 shows the chip area and access time overhead for adding partial-subblock support to single-page-size TLBs. The chip area overhead is small for the large increase in TLB reach. Further, adding partial-subblocking does not affect the access time. As mentioned in Section 5.1.2, the timing model assumes a combined comparator for the VPN and valid bits. Using the valid bit RAM approach (Appendix B) can reduce the chip area and access time further.

Table 5-2: Chip Area and Access Time for partial-subblock TLBs relative to single-page-size TLBs with same number of blocks

TLB	Fully-associative (subblock factor)				4-way set-associative (subblock factor)			
	2	4	8	16	2	4	8	16
Relative Chip Area	1.01	1.02	1.05	1.12	1.01	1.02	1.05	1.11
Relative Access Time	1.00	0.99	1.00	1.01	1.00	1.00	1.00	1.03

5.3 TLB miss handling for partial-subblock TLBs

TLB miss handling for a partial-subblock TLB is more complicated than for a single-page-size or complete-subblock TLB. It is important that TLB miss handling does not increase the TLB miss penalty and offset gains from reductions in the number of TLB misses. I first discuss a naive way to handle TLB misses that requires hardware support for subblock miss checking. I then show how *subblock preloading*[7], introduced in Section 4.2.2 for complete-subblock TLBs, improves TLB performance while requiring simpler hardware. In Chapter 7, I show how conventional page tables can be extended to support preloading without increasing the TLB miss penalty.

5.3.1 Naive TLB miss handling

A naive way to handle partial-subblock TLB misses uses a single-page-size page table to store mappings and *subblock miss checking* to determine if the new mapping can share any of the valid TLB blocks before loading the mapping into the TLB. This corresponds to the same three steps in a single-page-size system as described in Section 1.5.3. The naive TLB miss handler is as follows:

```
mapping = Find_mapping(VPN)
load_TLB_subblock(mapping)
if (fail) load_TLB(mapping, blocki)        /* blocki is TLB replacement victim */
```

The process of locating the mapping for the faulting address is the same as in a single-page-size system. The page table structure, page table traversal hardware and/or software need not change and use the same page size and algorithms, including reference/modified bit updates.

A subblock TLB can incur either a block miss or a subblock miss. In a complete-subblock TLB, the virtual address of the new mapping and the TLB tags are sufficient to determine a subblock miss (Section 4.2.1). In a partial-subblock TLB, the data field of the TLB blocks also needs to be compared with the new mapping to determine if the new mapping results in a subblock miss. Further, multiple TLB blocks may have the same tag and are candidates for storing the new mappings if the data fields match. Appendix G explores different alternatives for subblock miss checking in a partial-subblock TLB. One solution, first-tag-hit hardware, results in a simple hardware solution. I do not discuss this issue further as I introduce in the next section a TLB miss handling technique, preloading, that eliminates the need for subblock miss checking and also delivers better TLB performance.

Loading a new mapping into a partial-subblock TLB on a subblock miss only requires setting one valid bit in the TLB block, while the rest of the tag and data fields do not change. This

7. Robert Yung, Sun Microsystems Laboratories, first suggested preloading in partial-subblock TLBs to me.

can be implemented by either reading out the valid bits, setting the bit and writing them back or by providing control to write individual bits in the valid bit RAM or CAM. On a block miss, TLB replacement occurs and the new mapping overwrites the victim TLB block.

5.3.2 TLB miss handling using preloading

Preloading involves prefetching into the TLB all the mappings within a page block that will occupy the same TLB block as the mapping for the faulting virtual address. Preloading has two advantages over naive TLB miss handling—reduces the number of TLB misses significantly and does not require any hardware support for subblock miss checking. Preloading can increase TLB miss penalty, but Chapter 7 describes how page tables can support preloading in partial-subblock TLBs without increasing the TLB miss penalty.

Preloading exploits spatial locality by prefetching mappings to base pages within the same page block as the faulting virtual address. If the program references the neighboring pages before the TLB block is replaced, it will hit in the TLB. Since subblock TLBs, both complete- and partial-, use fewer tags and depend on spatial locality to expand TLB reach, it is only natural that preloading helps reduce the number of TLB misses. A partial-subblock TLB prefetches only the mappings that are properly placed with respect to the mapping of the faulting virtual address. This guarantees that a single TLB block is replaced as all the properly placed mappings share a single TLB block.

Table 5-3 shows the normalized execution time speedup due to preloading relative to partial-subblock TLBs without preloading. Preloading is very effective at reducing the number of TLB misses for the smaller TLBs. The larger TLBs incur fewer TLB misses due to preloading (Appendix J) but do not see any execution time speedup as the base partial-subblock TLB was able to map most of the working set.

Table 5-3: Effect of preloading in partial-subblock TLBs

TLB Type	#blocks	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
		Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty
Fully-associative	64	1.05	1.33	1.06	1.65	1.02	1.41	1.04	2.05
	128	1.02	1.25	1.01	1.20	1.02	1.36	1.02	2.79
	256	1.01	1.16	1.01	1.38	1.01	2.00	1.00	3.76
4-way Set-Associative	256	1.01	1.08	1.01	1.21	1.01	1.38	1.00	1.19
	512	1.00	1.09	1.01	1.17	1.00	1.81	1.00	1.06

I assume that the TLB miss penalty with preloading in partial-subblock TLBs is the same as in a single-page-size TLB. The *critical miss penalty* in Table 5-3 shows the break-even point for the TLB miss penalty where the reduction in the number of TLB misses is offset by the increase in TLB miss penalty. Therefore, for preloading to be useful, the TLB miss penalty must be less than (critical miss penalty * original TLB miss penalty). As the critical miss penalty is small—less than 1.4 for subblock factor 2 or 4 for subblock factor 16—the TLB miss handler must implement preloading efficiently, which I discuss next.

5.3.3 TLB miss handler for preloading in a partial-subblock TLB

A naive TLB miss handler preloads by fetching all the base page mappings for the page

block of the faulting virtual address, checking if any are properly placed with respect to the faulting mapping, constructs a partial-subblock TLB block and loads it into the TLB:

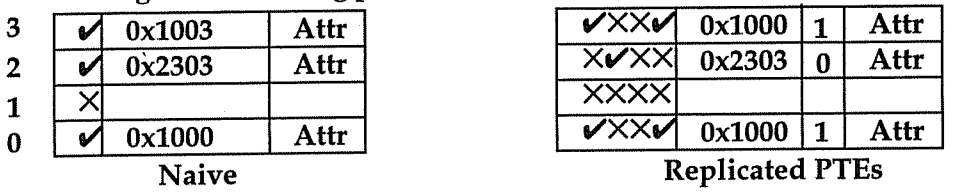
```
Mapping.Valid = 0;              /* bit vector */
PTE = Find_mapping(VPN);
Mapping.data = PTE.data;
Mapping.Valid[Block Offset(VPN)] = 1;
if (Mapping.SB = aligned(Mapping, VPN))
        for i = 0 to (s-1), except Block Offset(VPN)
                if (compatible(Find_mapping(i + VPBN), Mapping))
                        Mapping.Valid[i] = 1;
load_TLB(Mapping, blocki)[8]
```

The TLB miss handler can be simplified slightly if the hardware supports two varieties of the **load_TLB_subblock** operation—*strong* and *weak*. The strong version loads a base page mapping into the TLB, first checking to see if it can be placed in any partial-subblock TLB block, then forcing a replacement if it is not a subblock miss. The weak version loads a base page mapping only if it is compatible with the specified partial-subblock TLB block.

```
Mapping = Find_mapping(VPN);
blocki = load_TLB_subblock(Mapping, strong)
if (aligned(Mapping, VPN))
        for i = 0 to (s-1), except Block Offset(VPN)
                load_TLB_subblock(Find_mapping(i + VPBN),blocki, weak)
```

The naive TLB miss handlers are terribly inefficient and will easily exceed the critical miss penalty shown in Table 5-3. There are two inefficiencies in the above code. First, it is constructing a partial-subblock TLB block in the TLB miss handler when it would be more efficient to have the operating system preconstruct partial-subblock PTEs—as it does for superpages (Section 3.3). Second, it is fetching multiple mappings from the page table.

Figure 5-7: Storing partial-subblock PTEs in a linear page table



A simple solution for preloading in the TLB miss handler is to modify the page table to store partial-subblock PTEs computed by the operating system on page faults or when attributes change. Figure 5-7 illustrates how a linear page table can store partial subblock PTEs. Chapter 7 discusses adaptations to other page tables. The TLB miss handler is as follows:

```
Mapping = Find_mapping(VPN)
load_TLB(Mapping, blocki)
```

8. When preloading mappings, it is important to ensure that the TLB does not end up with multiple identical mappings. Blindly preloading on every TLB miss can result in such duplicates if some pages were mapped after a previous preload. Appendix D explores solutions.

This TLB miss handler is similar to that in a single-page-size TLB system. If a partial-subblock PTE fits in the same number of words as a base page PTE, the TLB miss penalty for a partial-subblock TLB with preloading will be same as the original TLB miss penalty. Thus, all the gains from preloading translate directly into reduction in time spent in TLB miss handling. Further, in some page tables, the use of partial-subblock mappings reduces the size of the page tables and may result in a faster page table lookup and a partial-subblock TLB miss penalty may be *less* than the single-page-size TLB miss penalty (Chapter 7).

In summary, if the page tables can be reorganized to store partial-subblock PTEs, TLB miss handling for partial-subblock TLBs is very efficient. In the rest of this chapter, I assume preloading in partial-subblock TLBs as it results in a) fewer TLB misses, b) simpler hardware, and c) potential for smaller page table size.

5.4 Impact of operating system support

Partial-subblock TLBs, like superpage TLBs, have a much larger TLB reach than a single-page-size TLB with the same number of blocks. However, they require operating system support to use the TLB reach effectively. Table 5-4 shows normalized execution time speedups when using n-block partial-subblock TLBs relative to using an n-block single-page-size TLB. The left half shows the speedup when the operating system properly places pages in physical memory. For my workloads, the partial-subblock TLBs results in significant speedups. The right half of Table 5-4 shows the speedup when running either old software or in a small memory system where aligned memory allocation may not be practical.

Table 5-4: Execution time speedups for partial-subblock TLBs (with preloading) relative to similar single-page-size (4KB) TLBs

TLB Type	with OS support (subblock factor)				without OS support (subblock factor)			
	2	4	8	16	2	4	8	16
64-block Fully-Associative	1.09	1.15	1.18	1.21	1.00	1.00	1.00	1.00
256-block 4-way Set-Assoc	1.06	1.09	1.11	1.12	1.01	1.02	0.87	0.80

The behavior of fully-associative partial-subblock TLBs degenerates to that of single-page-size TLBs with the same number of blocks in the absence of operating system support. This is because random physical memory allocation (the default) results in mostly unaligned mappings and no properly placed mappings for the partial-subblock TLBs.

For set-associative TLBs, the absence of operating system support is disastrous. The partial subblock TLB is more complicated to build than a single-page-size TLB but results in a slowdown! The performance gets worse as the subblock factor increases. This behavior is similar to that of a set-associative superpage TLB using the superpage index (Section 3.2.2) as a partial-subblock TLB also uses the same index bits. While associativity (four in this example) limits some losses for small subblock factors (upto four), the performance degradation is still significant.

It is important that a TLB provide reasonable performance in systems that either run legacy operating systems or have small amounts of memory where the operating system may not be successful at aligned physical memory allocation. Thus, set-associative implementations of partial-subblock TLBs are not attractive, if there is doubt in the availability of operating system support.

5.5 Sample design given area constraint

In this thesis, I propose three alternate TLB architectures—superpage, complete-subblock and partial-subblock TLBs—which have advantages and disadvantages as discussed in Sections 3.4, 4.4, and 5.6. In this section, I consider alternate fully-associative TLBs of the different TLB architectures that occupy comparable chip area, as estimated by the area model described in Section 2.2. Section 4.3 includes a similar design study considering single-page-size, complete-subblock, and superpage TLBs where I concluded that for a fixed chip area budget, in the absence of operating system support for superpages, a complete-subblock TLB is the best alternative. Section 3.4 also includes a similar design study considering set-associative single-page-size and fully-associative superpage TLBs where I concluded that in the absence of operating system support, a set-associative single-page-size TLB is a better choice. In both those sections, with the presence of operating system support, superpage TLBs result in the best execution time.

In this study I again consider the same four area budgets as in Section 4.3—the area required to implement fully-associative single-page-size TLBs of 64, 128, 256 and 512 blocks—and calculate the number of TLB blocks of the partial-subblock, superpage and complete-subblock TLBs that will fit in the same area[9]. Table 5-5 shows the number of TLB blocks for the different TLB architectures considered in this study. Note that the superpage TLB supports only the base page size and a superpage size of 32KB and not all possible superpage sizes.

Table 5-5: Number of blocks in alternate fully-associative TLBs of equal area

Area (rbe)	Single-page-size	Superpage (32KB)	Partial-subblock				Complete-subblock			
			2	4	8	16	2	4	8	16
7984	64	62	64	63	61	57	51	35	20	9
15298	128	123	127	126	122	114	102	72	44	23
29928	256	247	255	252	244	228	206	147	92	51
59186	512	494	509	504	489	456	413	297	188	107

I then estimate the access time for all the TLB configurations using the model described in Section 2.3. Table 5-6 illustrates the access time normalized with respect to the single page size TLB of comparable area. The partial-subblock TLBs are faster than the single-page-size and superpage TLBs as there are fewer tags and the tags are shorter. For large area budgets, the complete-subblock TLBs are faster due to the fewer tags. The difference in access time, however, is small and since they occupy comparable chip area, the implementation costs for the different TLBs are comparable. Therefore, TLB performance is more important criteria for selecting between these TLBs.

Table 5-6: Access time for alternate fully-associative TLBs of equal area

Area (rbe)	Single-page-size	Superpage (32KB)	Partial-subblock				Complete-subblock			
			2	4	8	16	2	4	8	16
7984	1.00	1.00	1.00	0.99	0.99	1.00	1.01	1.03	1.06	1.13
15298	1.00	1.00	1.00	0.99	0.99	1.00	1.00	1.01	1.04	1.10
29928	1.00	1.00	1.00	0.99	0.99	0.99	0.99	0.99	1.00	1.05
59186	1.00	0.99	1.00	0.99	0.98	0.97	0.96	0.92	0.90	0.93

9. Since a TLB cannot have fractional number of blocks or fractional subblock factor, the TLB size has an area closest to the budget.

Table 5-7 shows the normalized execution time speedup for the TLBs shown in Table 5-5 relative to the respective single-page-size TLB—assuming no subblock preloading in the TLB miss handler. Table 5-8 shows the normalized execution time speedup assuming subblock preloading. The performance of partial-subblock TLBs is better than that of the complete-subblock TLBs. The partial-subblock TLBs incur fewer TLB misses because they have more TLB blocks and a larger TLB reach than the complete-subblock TLBs.

Both superpage and partial-subblock TLBs show substantial speedups over the single-page-size TLBs. Partial-subblock TLBs are more attractive with preloading. The partial-subblock TLBs with subblock factor of 8 or 16 have a better speedup than superpage TLBs.

Table 5-7: Execution time speedups relative to single-page-size (4KB) TLBs of equal area

Area (rbe)	Single-page-size	Superpage (32KB)	Partial-subblock				Complete-subblock			
			2	4	8	16	2	4	8	16
7984	1.00	1.18	1.04	1.09	1.15	1.17	1.03	1.04	1.01	0.81
15298	1.00	1.13	1.05	1.10	1.11	1.15	1.02	1.04	1.05	1.03
29928	1.00	1.07	1.04	1.05	1.06	1.08	1.02	1.03	1.01	1.00
59186	1.00	1.03	1.01	1.01	1.03	1.03	1.00	1.00	1.00	0.99

Table 5-8: Execution time speedups using preloading in subblock TLBs relative to single-page-size (4KB) TLB of equal area

Area (rbe)	Single-page-size	Superpage (32KB)	Partial-subblock				Complete-subblock			
			2	4	8	16	2	4	8	16
7984	1.00	1.18	1.09	1.15	1.18	1.21	1.08	1.12	1.13	0.98
15298	1.00	1.13	1.07	1.11	1.13	1.17	1.06	1.09	1.10	1.12
29928	1.00	1.07	1.05	1.06	1.07	1.08	1.04	1.05	1.03	1.04
59186	1.00	1.03	1.01	1.02	1.03	1.03	1.01	1.01	1.01	1.01

The subblock factor of a partial-subblock TLB can be increased with very little overhead in hardware and software. Supporting a larger superpage size has little overhead in hardware but has other costs—increasing the superpage size reduces the number of segments that can use superpages and increases the amount of internal fragmentation and memory usage. Increasing the subblock factor in a partial-subblock TLB does not have these overheads. Thus, it is more likely that an operating system will support a partial-subblock TLB with subblock factor 16 than a superpage TLB with superpage size of 64KB.

If operating system support for superpage or partial-subblock TLBs is not available, their performance degenerates to that of a fully-associative single-page-size TLB with *fewer* TLB blocks! Table 5-9 compares the execution time speedups for the same TLBs assuming the operating system does not use superpages or do proper physical memory allocation. Complete-subblock TLBs are a better alternative as explained in Section 4.3.

In summary, for a fixed chip area budget, partial-subblock TLBs with preloading offer the best TLB performance. The largest subblock factor that would allow the PTE to fit in a single word should be chosen—8 or 16. In the absence of operating system support, complete-subblock TLBs are a better choice.

Table 5-9: Execution time speedups using base pages relative to single-page-size (4KB) TLBs of equal area (without preloading)

Area (rbe)	Single-page-size	Superpage (32KB)	Partial-subblock TLB				Complete-subblock TLB			
			2	4	8	16	2	4	8	16
7984	1.00	1.00	1.00	1.00	0.99	0.98	1.02	1.04	1.01	0.81
15298	1.00	1.00	1.00	1.00	1.00	0.99	1.02	1.04	1.05	1.03
29928	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.03	1.01	1.00
59186	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	1.00	0.99

5.6 Comparison with other TLB architectures

In this section I compare the costs and benefits of increasing TLB reach using partial-subblock TLBs relative to using single-page-size, superpage, and complete-subblock TLBs with the same TLB reach.

5.6.1 Partial-subblock vs. single-page-size TLBs with same TLB reach

This section compares two different approaches to increasing TLB reach—a larger single-page-size TLB or a smaller, smarter partial-subblock TLB. The partial-subblock TLB has worse TLB performance and requires operating system support but occupies a much smaller chip area and has a significantly smaller access time.

A partial-subblock TLB increases its TLB reach by large factors using few extra tag bits and some control logic. Increasing the TLB reach in a single-page-size TLB by a similar amount results in much larger (and slower) TLBs. A 64-block partial-subblock TLB with subblock factor 8, for example, has the TLB reach of a 512-block single-page-size TLB, which occupies seven times larger area and is 40% slower to access.

The left half of Table 5-10 summarizes the area overhead for a partial-subblock TLB compared to a single-page-size TLB with the same TLB reach, *i.e.*, the partial-subblock TLB with subblock factor s has $1/s$ times the number of blocks in the corresponding single-page-size TLB.

Table 5-10: Chip Area and Access Time for partial-subblock and single-page-size TLBs with same TLB reach

TLB type	Single Page Size		Relative Chip Area				Relative Access Time			
			2	4	8	16	2	4	8	16
	#blocks	N	N/2	N/4	N/8	N/16	N/2	N/4	N/8	N/16
Fully-associative	256	1.00	0.51	0.27	0.15	0.09	0.91	0.85	0.82	0.80
	512	1.00	0.51	0.26	0.14	0.08	0.83	0.76	0.72	0.70
4-way set-associative	256	1.00	0.58	0.36	0.26	0.22	0.98	0.98	0.97	0.99
	512	1.00	0.55	0.32	0.20	0.15	0.94	0.92	0.92	0.93

A partial-subblock TLB, being much smaller, is also faster to access than the larger single-page-size TLB—as shown in the right half of Table 5-10. The fully-associative TLBs with larger subblock factors show increasingly smaller access times as the partial-subblock TLBs have much smaller tag and data arrays and smaller arrays have faster access times. The set-associative TLBs show only a marginally better access time as the tag side of the TLB is on the critical path and reduction in the data RAM size does not help the overall access time. The tag access

time does not improve much since the multiple valid bits in the tags of partial-subblock TLBs increase the tag compare and multiplexor driver delays—this can be optimized further through sizing the drivers.

Table 5-11 shows the normalized execution time speedups for partial-subblock TLBs (with preloading) relative to single-page-size TLBs with the same TLB reach. Each row of Table 5-11 uses a 256- or 512-block single-page-size TLB as the base TLB and each column uses successively fewer partial-subblock TLB blocks with larger subblock factors. Though the partial-subblock TLBs have fewer tags, they have comparable performance to the larger, slower single-page-size TLBs. Subblocking, however, is only effective when there are enough tags to exploit spatial locality. With very few blocks, subblocking is not useful as TLB blocks get replaced before the program uses all the subblocks.

Table 5-11: Execution time speedups for partial-subblock TLBs relative to single-page-size (4KB) TLBs with same TLB reach

TLB type	Single Page Size (4KB)	Partial-subblock with preloading (subblock factor)			
		2	4	8	16
	#blocks (N)	N/2	N/4	N/ 8	N/16
Fully-Associative	256	0.98	0.99	0.99	0.97
	512	0.99	0.97	0.97	0.98
4-way Set-Associative	256	1.00	1.00	0.98	0.88
	512	0.99	0.99	0.98	0.95

Comparing set-associative TLBs in the last two lines of Table 5-11, shows that associativity is important in a partial-subblock TLB. A partial-subblock TLB depends on the ability of the TLB to store multiple TLB blocks with the same VPBN (but different valid bits) to accommodate pages that have incompatible mappings. An associativity greater than or equal to the subblock factor (4 in this example) results in good performance for the partial-subblock TLB though it has fewer tags. Larger subblock factors cause thrashing that the associativity cannot handle for unaligned and incompatible page blocks.

In summary, partial-subblock TLBs offer a large TLB reach for a low cost but depend on spatial locality and proper physical memory allocation. For these workloads, given enough TLB blocks (64 or larger), associativity (larger than or equal to the subblock factor), and proper operating system physical memory allocation, partial-subblock TLBs offer competitive performance to a much larger and slower single-page-size TLB.

5.6.2 *Partial-subblock vs. Superpage TLBs with same TLB reach*

Of the TLBs considered in this thesis, partial-subblock and superpage TLBs with the same TLB reach are closest in implementation costs. A partial-subblock TLB with subblock factor s has the same TLB reach as a superpage TLB that supports a single medium-sized superpage (superpage size equal to the page block size) and has the same number of TLB blocks. Partial-subblock TLBs adds subblock valid bits to superpage TLBs and are fractionally more complicated to build. However, a partial-subblock TLB block share TLB blocks more often than superpages due to simpler restrictions and incurs fewer TLB misses. Further, partial-subblock TLBs require simpler, more efficient operating system support.

Comparing the chip area required, the left half of Table 5-12 shows that the two TLB types occupy comparable area. The data fields of each TLB block are comparable for both superpage and partial-subblock TLBs. In the tags, a partial-subblock TLB has $\log_2(s)$ fewer VPN bits but has (s-1) more valid bits. A superpage TLB block has an additional $\log_2(s)$ MASK bits. For small subblock factors, two or four, a partial-subblock TLB occupies a smaller area. For larger subblock factors, a partial-subblock TLB occupies more area.

Table 5-12: Chip Area and Access Time for partial-subblock TLBs relative to superpage-TLBs

Superpage TLB Type	Chip Area subblock factor:superpage size				Access Time subblock factor:superpage size			
	2:8KB	4:16KB	8:32KB	16:64KB	2:8KB	4:16KB	8:32KB	16:64KB
Fully-associative	0.99	0.99	1.01	1.07	0.99	0.99	0.99	1.01
4-way Set-associative	0.99	0.99	1.01	1.06	0.99	0.99	1.00	1.02

Comparing the access time in the right half of Table 5-12, the partial-subblock TLBs are marginally faster except for a subblock factor of 16 where they are marginally slower. The don't-care bits in a fully-associative superpage TLB block degrade the time taken for the tag compare. In a partial-subblock TLB, the fewer tag bits improve the tag compare time while the extra valid bits degrade it.

Table 5-13 shows the TLB performance of partial-subblock TLBs relative to superpage TLBs—a partial-subblock TLB with subblock factor s is compared with a superpage TLB that supports a single medium-sized superpage size equal to the page block size. The set-associative superpage TLBs use the superpage index. A partial-subblock TLB without preloading incurs more misses than a superpage TLB because it takes multiple TLB misses to load the mappings for the base pages within a page block while a superpage TLB can load multiple mappings in a single TLB miss. Preloading in partial-subblock TLBs eliminates this advantage of superpages and the right half of the table shows that partial subblocking is comparable to using medium-size superpages. Tables in Appendix J further show that the partial-subblock TLBs incur fewer misses than superpage TLBs—though the reduction does not result in a noticeable execution time speedup for these workloads.

Table 5-13: Execution time speedups for partial-subblock TLBs relative to superpage TLBs

TLB Type	#blocks	Without subblock preloading subblock factor:superpage size				With subblock preloading subblock factor:superpage size			
		2:8KB	4:16KB	8:32KB	16:64KB	2:8KB	4:16KB	8:32KB	16:64KB
Fully-associative	64	0.96	0.95	0.98	0.96	1.00	1.00	1.00	1.00
	128	0.98	0.99	0.98	0.98	1.00	1.00	1.00	1.00
4-way Set-associative	256	0.99	0.99	1.00	1.04	1.00	1.00	1.01	1.05

The partial-subblock TLBs improve system performance in other ways that make them more attractive than medium-size superpages.

First, superpage systems incur some penalty due to internal fragmentation—increased paging I/O, page initialization overhead—which the partial-subblock system does not. A partial-subblock system uses the same amount of memory as in a single-page-size system. This is an important benefit that I do not quantify here.

Second, use of partial-subblock PTEs reduces page table size by 20% to 50% more than superpage PTEs do (Table 7-6). This effect can be more significant than just the memory savings as it reduces cache pollution and page table search time, which translates into smaller TLB miss penalty.

Third, partial-subblock TLBs also support superpage mappings by appropriately setting the subblock valid bits (Appendix H). If the operating system chooses to use superpages, the page fault latency can be reduced by using the subblock feature to read only the base page (subblock) needed first and complete the rest in the background—a feature that CPU subblock caches also exploit to reduce cache hit time.

Fourth, reference information is available at the granularity of a base page size. This can result in better page replacement decisions or can be used for more efficient page promotion and demotion decisions in a superpage system—where reference information is only available at a coarser granularity. Similarly, modified information will be available at the granularity of a base page size in a partial-subblock TLB with replicated write-permission bit or if separate TLB blocks are used for dirty and clean subblocks.

Fifth, in systems that maintain attribute information at the granularity of a base page size, *e.g.*, page-based distributed shared memory systems, superpages pages are less likely to be used due to the difference in attributes or holes. Partial-subblock TLBs can still share TLB blocks for multiple base pages within a page block though there may be holes or differences in attributes.

Lastly, partial-subblock systems use only a subset of the operating system mechanisms required for medium-sized superpage systems—requires only variable size freelist management and careful physical memory allocation. Consequently, a partial-subblock system is more efficient.

In summary, a partial-subblock TLB includes all the performance benefits of using a medium-sized superpage and adds some more. A partial-subblock TLB incurs fewer TLB misses than a superpage TLB with the same TLB reach and has fewer operating system costs. The partial-subblock systems are at their best with a TLB miss handler and page table that stores partial-subblock PTEs.

5.6.3 Partial- vs. complete subblock TLBs with same TLB reach

Partial subblocking is a low-cost alternative to complete-subblocking but delivers comparable TLB performance. A complete-subblock TLB increases TLB reach by providing space for separate mappings for every base page within a page block—a high cost in both chip area and access time but deliver superior TLB performance without operating system support. A partial-subblock TLB, on the other hand, provides only a single copy of the mapping for pages within a page block with individual valid bits allowing multiple TLB blocks to map disjoints sets of base pages of a page block—a low overhead in both chip area and access time but requires operating system support.

The left half of Table 5-14 compares the area required to build partial- and complete-subblock TLBs with equal TLB reach. Since a partial-subblock TLB shares most of the TLB block only requiring additional subblock valid bits and an extra attribute bit, a partial-subblock TLB block is much smaller than a complete-subblock TLB block.

Table 5-14: Chip Area and Access Time for partial-subblock TLBs relative to complete-subblock TLBs

TLB Type	Chip Area				Access Time			
	2	4	8	16	2	4	8	16
Fully-associative	0.81	0.59	0.39	0.24	0.97	0.94	0.90	0.84
4-way set-associative	0.73	0.48	0.29	0.17	1.00	0.99	0.97	0.92

The smaller TLB blocks also translate to faster access time for the partial-subblock TLBs (the right half of Table 5-14). Both TLBs have the same number of VPN bits in the tag but partial-subblock TLBs have a longer tag compare time due to the subblock valid bits included in tag comparison—complete-subblock TLBs use a block-valid bit. However, the smaller data RAM in fully-associative partial-subblock TLBs has a significantly faster lookup time, resulting in an overall faster access time. In set-associative partial-subblock TLBs, the data RAM access time is not on the critical path but the multiplexor drivers are faster as they drive the signals across a thinner RAM than in complete-subblock TLBs.

A partial-subblock TLB delivers comparable TLB performance to that of a much larger and slower complete-subblock TLB (Table 5-15). Tables J-10 to J-37 in Appendix J show that the number of TLB misses incurred in a partial-subblock TLB is only a few percent more than in a complete-subblock TLB. This is important since a faster partial-subblock TLB access time may improve processor cycle time, which affects program execution time more than a few percent change in the number of TLB misses (Table 7-5).

Table 5-15: Execution time speedups for partial-subblock TLBs relative to complete-subblock TLBs with same subblock factor, number of blocks and associativity

TLB type	#blocks	with preloading subblock factor			
		2	4	8	16
fully-associative	64	1.00	1.00	1.00	1.00
	128	1.00	1.00	1.00	1.00
4-way set-associative	256	1.00	1.00	1.00	0.98

A partial-subblock TLB has more misses than an equivalent complete-subblock TLB for three reasons. First, pages within a page block that have different attributes share the same complete-subblock TLB block but require separate partial-subblock TLB blocks. This results in a smaller effective TLB reach and more TLB misses in a partial-subblock TLB. The effect, is noticeable for the set-associative TLB with subblock factor 16 where the associativity of 4 limits the number of partial-subblock TLB blocks for the same page block that can coreside in the TLB. Second, virtual address allocation for a complete-subblock TLB is denser than in a partial-subblock TLB system[10] that results in more sharing in a complete-subblock TLB. Third, when two objects are mapped to addresses within the same virtual page block, a single complete-subblock TLB block is used. Foxtrot allocates different chunks of physical memory[11] for the two objects resulting in the use of multiple partial-subblock TLB blocks.

The TLB miss penalty for preloading in partial-subblock TLBs is smaller than in complete-subblock TLBs. With a page table that supports partial-subblock PTEs (Section 7.4.3), a partial-subblock TLB miss handler fetches a single word from memory whereas a complete-subblock

10. This is an artifact of Foxtrot's virtual address allocation strategy for partial-subblock systems (Section 6.3.1)

11. Foxtrot allocates contiguous aligned physical pages for *each* object but the two chunks of memory are not aligned to each other.

TLB miss handler fetches multiple words. Thus, though a partial-subblock TLB incurs more TLB misses than a complete-subblock TLB, partial-subblock TLBs may spend less time in TLB miss handling.

In summary, partial-subblock TLBs offer comparable TLB performance to complete-subblock TLBs, but occupy a significantly smaller chip area and have a faster access time. However, partial-subblock TLB performance depends on operating system support in physical memory allocation.

5.7 Variations of partial-subblock TLBs

Section 5.1 introduced partial-subblock TLBs with only subblock valid bits. Other variations are possible where the data stores other fields per-subblock—modified bits, other attributes, PPN. A complete-subblock TLB is a trivial variation, where all the fields in the mapping are stored per subblock. There is a tradeoff in choosing between providing storage for subblock attributes or shared attributes for a TLB block. A shared attribute has the advantage of a smaller TLB block and the TLB either occupies a smaller area or can fit more blocks in the same area. Shared attribute fields have the disadvantage of requiring separate TLB blocks if base pages within a page block have different attributes.

In practice, for small subblock factors the extra chip area required to provide per-subblock single bit attribute fields is negligible. The important consideration is whether the subblock-PTE format will continue to fit in a single word after replicating more bits. Using a single word PTE is preferable as the TLB miss handler is more efficient and is easier to do atomic update of page tables—multi-word atomic updates require use of other synchronization methods.

- Valid bits: Many advantages of subblock TLBs are due to subblock valid bits. Reducing I/O bandwidth, reduced memory usage and wider applicability than superpages are the main advantages. I expect most subblock-TLBs to support subblock valid bits.

Configurations that do not provide subblock valid bits are possible. They are simpler to implement, *e.g.*, will not require decoding of low-order VPN bits to determine a TLB hit, and have a faster access time. The ARM6x0 and the RS/6000 processors, for example, allow for different attributes for subpages but require all subpages to be valid. This is useful when the operating system or application requires fine-grain protections, *e.g.*, database locking, garbage collection. This also can be viewed as allowing subpage attributes in a superpage or single-page-size TLB.

Another option is to store an encoded version of the valid bit vector[12]. This has the advantage of allowing large subblock factors (32, 64) with small hardware cost but has the disadvantage that it allows only a few valid bit patterns. Two examples illustrate this. First, a superpage TLB is a trivial example, where a single bit encodes the valid bit vector. The TLB block is shared only if all base pages are compatible. Second, a subblock factor of 64 can be encoded as two 6-bit fields (instead of a 64-bit vector)—the position of the first and last valid bits with all bits in between set (a run). This allows any contiguous set of base pages that are all properly placed to share a single TLB block, *i.e.*, they need not start or end at a page block boundary. Other exotic encodings are possible but complex hardware (or software) may be needed to decode the encoding during tag comparison—a full valid bit vector results in a simple implementation. Further, encoding places tighter constraints on when base pages can

12. Encoding was first suggested to me by Russell Kao, DEC WRL.

share a TLB block and shares TLB blocks less often, *e.g.*, superpage TLBs are less effective than partial-subblock TLBs (Section 5.6.2).

- Physical Page Number (PPN): This field is typically the largest in a TLB block's data. Most of the area overhead of complete subblock TLBs is due to having subblock PPN fields. Having a single shared copy of this field for the subblocks in a page block allows a much smaller TLB to be built and is the primary advantage of the partial-subblock TLBs. However, the performance improvement is conditional to the operating system doing the appropriate physical memory allocation.

It is also possible to consider an implementation that shares only a *few* of the PPN bits, *e.g.*, the most significant bits, while maintaining separate copies for the low order bits. This will place less stringent constraints on the operating system and gives the operating system flexibility in physical memory allocation. My experience with operating systems suggests that the complexity of allocating for the less stringent system of constraints is not much easier that for the more stringent system, however, this is an option to explore.

On the other hand, storing the low order bits of the PPN per subblock, while storing a shared copy of the high order bits, gives the operating system flexibility in page coloring. Sharing the full PPN will make the operating system allocate physical pages with colors same as the virtual addresses (modulo the page block size). This interferes with physical page coloring for large physically addressed caches. Any skew in the virtual addresses used by the application will show up in the physical coloring too. Virtual color 0, for example, is used frequently as many segments start at a properly aligned address. Consequently, physical color 0 will be heavily used if the operating system attempts to optimize for partial-subblock TLB performance. Storing some low-order bits of the PPN per subblock allows the operating system to implement a different page coloring algorithm. This also eliminates the need for the SB attribute and the multiplexor in physical address generation, if the TLB stores block offset bits per subblock[13].

- Attributes: Attributes include fields such as protection, cacheability, page size, referenced and modified bits. A decision to share or replicate must be made for each field individually. Base pages in a page block that have different values for a replicated attribute can still share a single TLB block and improve TLB performance. Fields which can be expected to be different for consecutive virtual pages should be replicated, *e.g.*, write-permission bits. Distributed shared memory operating systems that use page-level protections to implement coherent memory, for example, would set write-permission bits at the granularity of base pages more frequently than in single-node computers. Some attributes, such as the privileged bit, tend to be identical for pages within a page block as they usually belong to the same virtual object, and can be shared among subblocks. Section 5.1.3 discussed the tradeoff in replicating modified bits.

In summary, while there are many possible partial-subblock TLB configurations possible by choosing different bits or (portions of) fields to share across subblocks, two important constraints limit the choices—the available chip area or access time and PTE format. The physical page number field is the largest and most obvious one to share but requires operating system physical memory allocation to be effective. Subblock valid bits give the most important properties of subblock TLBs and are essential. Subblock modified bits can be effective at reducing

13. This is important for a subblock factor of 2, where storing one additional low order bit of the PPN, eliminates the need for the SB attribute and the multiplexor, but adds a column multiplexor to select the subblock.

I/O and should be considered if the PTE format allows for it.

5.8 Conclusion

The partial-subblock TLB architecture, is a key contribution of my thesis. I have studied the issues involved in implementing a partial-subblock TLB and explored alternate ways to handle TLB misses. I have shown that a partial-subblock TLB is better than the state-of-the art superpage and complete-subblock TLB implementations.

A partial-subblock TLB allows base page mappings that are properly placed in physical memory to share a single TLB block. Using spatial locality in programs, a partial-subblock TLB uses fewer tags than a single-page-size TLB with the same TLB reach. Further, using intelligent physical memory allocation by the operating system, a partial-subblock TLB uses less datapath chip area than either single-page-size or complete-subblock TLBs with the same TLB reach. By providing subblock valid bits, partial-subblock TLBs incur fewer TLB misses than an equivalent superpage TLB but require only best-effort physical memory allocation from the operating system.

Set-associative partial-subblock TLBs have similar characteristics to superpage-index set-associative superpage TLBs. They are effective with operating system support for superpages but use sub-optimal index bits when not using superpages. Fully-associative implementations degrade gracefully to perform comparably to a single-page-size TLB with the same number of TLB blocks.

In the next two chapters, I discuss the operating system mechanisms needed to support partial-subblock TLBs (and superpage TLBs) and page tables to store partial-subblock (and superpage) mappings.

Chapter 6 Operating System Support

Virtual memory [Denn70] computer systems require a close interaction between the hardware architecture and the operating system. Operating system support for virtual memory with a single fixed page size is substantial but well-understood (*e.g.*, UNIX [Thom74, Bach86, Leff90, Ging87b], VMS [Levy82], NT [Cust93], MACH [Acce86, Rash88], OS/2 [Koga88]). It includes a virtual memory manager that allocates virtual addresses, enforces protection, initiates I/O and loads/unloads mappings from a page table; one or more file systems that manage and maintain structure/coherence of objects on disk/network; a physical memory manager that manages/allocates physical pages; and a page table manager that isolates page table and TLB details in a machine-dependent module, *e.g.*, SYSV UNIX **hat** layer [Mora88, Bala92] and Mach **pmap** layer [Rash88].

To be effective, however, superpage and partial-subblock TLBs require operating system support in areas other than TLB and page table management (Chapter 7 discusses page tables). The primary contribution of this chapter is that I identify the operating system support required and discuss alternate solutions. One new policy and upto six new mechanisms may be required to convert a single-page-size operating system to one that supports superpage or partial-subblock TLBs. Table 6-1 shows alternate TLB types with the mechanisms that are required, optional or not applicable (N/A) for each.

Table 6-1: Operating system mechanisms for superpage and partial-subblock TLBs

TLB Type	Page-size Assignment Policy	OS Mechanisms					
		Variable Size Freelist	Gather	Page Promotion/ Demotion	Monitor Reference Patterns	Multiple page-size framework	Careful Physical Memory Allocation
Partial-subblock	N/A	required	optional	N/A	N/A	N/A	required
Superpage	Static	required	N/A	N/A	N/A	optional[a]	optional[a]
	Dynamic	required	required	required	required	optional	required

a. Static page-size assignment requires at least one of the two mechanisms—multiple page-size framework or careful physical memory allocation.

Supporting partial-subblock TLBs requires two mechanisms—variable size freelist management to allocate contiguous regions of physical memory and careful physical memory allocation to *properly place* virtual pages in physical memory (Table 1-1). Optionally, a gather operation can correct mistakes in physical memory allocation by copying base pages to their proper places.

Superpage operating system support includes a new policy—*page-size assignment* policy—that decides when to use superpages, what size superpages, and for which address space regions. Page-size assignment can be *static*—the decision is made once and does not change over the lifetime of the process—or *dynamic*—the page size changes over time. Supporting superpages with a static page-size assignment policy requires variable size freelist management. In addition, the operating system data structures and interfaces should support a true multiple-page-size framework. With a static page-size assignment policy, it also suffices to use careful physical memory allocation in a single-page-size framework and a page table that coalesces compatible base page mappings into superpages. Static page-size assignment is prac-

tical in only a few situations as it does not provide a way to recover from wrong decisions or use smaller page sizes when memory is scarce.

A dynamic page-size assignment policy allows the page size for a virtual address region to change in response to changes in reference patterns or available physical memory. This, however, requires additional operating system mechanisms—page promotion and page demotion mechanisms to change the page size, a gather mechanism to collect base pages to their proper place, and a mechanism to collect program reference patterns to help make page-size assignment policy decisions.

A *gather* operation moves base pages to "proper" physical pages so that a superpage mapping or a partial-subblock mapping can be used. This requires locking pages in memory, modifying page tables and performing TLB shootdowns that adds significant overhead to the actual copy costs. Instead, I propose a physical memory allocation algorithm, *page reservation*, that carefully allocates "proper" physical pages in the first place and avoids the copying. Page reservation works by reserving a physical page block for specific base virtual pages and holding the base physical pages at the end of the freelist. When the program references these virtual pages, the operating system will allocate previously reserved properly placed pages. However, if reserved pages reach the head of the freelist without the program referencing them, they are reallocated. Page reservation makes a *best-effort* to properly place physical pages with low overhead. This is sufficient for a partial-subblock system and reduces (eliminates) gather costs when deciding between base pages and a single superpage size. Gather operations can be used to correct any improper placement or to augment page reservation for more page sizes.

The rest of this chapter discusses page-size assignment policies and the different mechanisms in detail. Section 6.1 discusses alternate page-size assignment policies. Section 6.2 discusses implementation of the different mechanisms—freelist management, gather, page promotion/demotion, page reservation, monitoring, and changes required to move an operating system to a multiple-page-size framework. Section 6.3 discusses interactions between the new policies and mechanisms with existing operating system policies and mechanisms. Chapter 7 discusses page tables that can store and service TLB misses for superpage and partial-subblock mappings.

6.1 Page-size assignment for superpage TLBs

A *page-size assignment policy* makes a tradeoff between the costs and benefits of using superpages in deciding the page size to use for each virtual address. The primary benefit of using superpages is a reduction in the number of TLB misses (Chapter 3). The costs of using superpages include a) overhead in monitoring the reference pattern of the workload, b) increased internal fragmentation, *i.e.*, larger working set size and increased page initialization costs [Tall92], c) page promotion costs (Section 6.2.3), and d) increase in TLB miss penalty (Chapter 7). Page-size assignment can be either static or dynamic and this section describes two classes of dynamic policies—working set threshold [Tall92, Tall94a] and competitive [Rome95].

A *static page-size assignment policy* makes the decision once and fixes the page size over the life of the mapping. Device pages and non-pageable memory can use a static policy of using the largest superpage size that maps the object (*e.g.*, kernel text, frame buffers, database buffers). Operating systems could also use simple heuristics for static page-size assignment poli-

cies based on the type of object and available free memory. For example, a static policy could assign base pages for stack pages and medium-sized superpages for heap pages.

A *dynamic page-size assignment policy* is more flexible allowing the page size for a virtual address region to change and is useful in two situations. First, when the operating system does not know enough about the costs and characteristics of accesses to an object to make a static decision, a dynamic policy allows it to guess a page size and modify the page size after monitoring the process for a while. Second, using superpages increases internal and external fragmentation and will increase paging traffic if the system is short of physical memory—a dynamic policy allows the operating system to adapt page-size assignment to changes in such system parameters.

A *working set threshold policy* promotes page blocks when the working set contains more base pages of a page block than a predetermined threshold, the *promotion threshold* (*e.g.*, eight 4KB base pages within a 64KB page block). It demotes superpages to base pages or smaller superpages when the working set contains fewer base pages from the page block than another predetermined threshold, the *demotion threshold* (*e.g.*, five 4KB base pages within a 64KB page block). Romer *et al.* characterize such policies as ASAP (as-soon-as-possible) policies [Rome95]. Foxtrot implements a working-set threshold policy and makes policy decisions between two page sizes [Tall94a].

Superpage TLBs and page tables do not gather reference information at base page granularity for superpage mappings—there is only a single referenced attribute bit per superpage PTE. Thus, all base pages of a superpage are in the working set or none are, *i.e.*, if the corresponding superpage is in the working set we cannot determine that a particular base page is not in the working set. A page replacement policy can choose to either replace superpages that are not in the working set or demote them. Replacing the superpage is more attractive than demoting it, as it is not in use anyway.

The promotion threshold is an important parameter in working set threshold policies. A high threshold uses superpages less often, reducing the number of page promotions and memory usage, incurs less internal fragmentation but incurs more TLB misses than with a lower threshold. A threshold of 0% always uses superpages and a threshold of 100% uses superpages only for fully populated page blocks. Four factors determine the threshold: page promotion cost, expected program reference pattern, amount of free physical memory, and page fault latency. If the program is expected to reference most or all of the page block, the threshold should be 0%—*i.e.*, use superpages always—as it is more efficient to allocate a superpage statically than allocating base pages and later promoting them. Foxtrot uses a threshold of 50% for **ufs** files, 75% for **nfs** files and 100% for heaps. The advantage of a working-set threshold policy is that it is cheap to monitor the working set, *e.g.*, maintaining counters on page faults, but can unnecessarily promote page blocks that do not incur many TLB misses.

A competitive algorithm makes decisions that result in performance within a constant factor of an optimal policy and competitive algorithms have been used in other contexts, *e.g.*, [Karl88, Karl91, Slea85, Cao94]. Romer *et al.* recently proposed a *competitive page-size assignment policy* that accounts for the cost of TLB misses and captures reference patterns by updating counters for every base page and superpage on TLB misses [Rome95]. The policy promotes pages when TLB miss costs exceed a threshold based on page promotion costs. The advantage is that, by accounting for the workload's TLB miss patterns, competitive policies can make a better page-size assignment, often have less internal fragmentation, and use less physical memory than working-set threshold policies. The disadvantage of competitive poli-

cies is the increase in TLB miss penalty, *e.g.*, from 30 to 130 cycles, and memory overhead and cache pollution due to the extra counters, *e.g.*, 3.125 counters per base page [Rome95]. By using superpages, these costs are offset by the decrease in the number of TLB misses and programs show a net decrease in execution time. Page demotions occur, if at all, when the superpage is selected for page replacement.

Any page-size assignment policy must support page demotion for user applications that change attributes for base pages within a superpage. Distributed shared memory machines that use page-level protections or garbage collection systems, for example, frequently change attributes at base page granularity. It is unlikely that operating systems will choose superpages for such applications, unless the applications are aware of the use of superpages by the operating system and adjust their attribute change requests. Partial-subblock systems are less affected by such changes as the rest of the unaffected base pages within the page block can continue to share a single TLB block.

The operating system also changes attributes for base pages that are part of superpages, *e.g.*, to implement copy-on-write or maintain modified bits. A page-size assignment policy can choose either to extend the attribute change for the full superpage or demote it. Section 6.3.3 illustrates this by explaining different ways an operating system could handle copy-on-writes to a superpage. Other attribute changes involve similar tradeoffs.

Besides implementing a default page-size assignment policy, an operating system also could export some mechanisms to user programs[1], compilers or run-time libraries, *e.g.*, through the UNIX **madvise** system call. This allows implementation of custom user-defined page-size assignment policies that exploit programs' knowledge of their access pattern. Some operating systems have similarly exported mechanisms such as page replacement [Youn89, Hart92], scheduling [Ande92], and cache coherence [Rein94].

In summary, operating systems have a choice of a variety of page-assignment policies and different workloads may prefer different policies. The key to operating system design is to identify and implement the mechanisms that can support many alternate policies. The next section identifies the basic mechanisms needed to support superpages.

6.2 New Operating System Mechanisms

Superpage and partial-subblock TLB support requires one or more of six new operating system mechanisms, besides page table support (Table 6-1)—variable sized free physical memory management (Section 6.2.1), a gather mechanism (Section 6.2.2), page promotion/demotion mechanisms (Section 6.2.3), a mechanism to monitor reference patterns (Section 6.2.4), careful physical memory allocation (Section 6.2.5) and data structure and interface changes to support a multiple-page-size framework (Section 6.2.6).

6.2.1 Freelist management

The most important mechanism required to support superpages and partial-subblocking is variable-sized physical memory allocation. Most operating systems treat all physical memory as interchangeable, equal-sized chunks (pages or frames) and use an unordered list of free pages as the freelist. A superpage requires allocating a physical page block (aligned and

1. The operating system would most likely treat user page-size assignment decisions as advisory and make a best-effort as it is often unacceptable in a multi-user system to allow user programs to control memory allocation.

contiguous region of memory) equal to the superpage size. There are two issues that arise. First, finding a free page block from the freelist is inefficient and needs better data structures. Second, superpage use can be limited by external fragmentation, *i.e.*, the amount of free memory is greater than the page block size but the free pages are scattered such that a contiguous chunk is not available.

Variable-sized freelist management has been studied in the context of segments and memory allocators [Know65, Hirs73, Barr93]. The problem is simpler here than general memory allocators as systems support only a few superpage sizes that are powers of two.

A buddy-block allocator [Knut68a, Pete77, Tayl81, Purd70, Bark89, Lee89c] organizes free pages into multiple freelists, one per supported allocation size and has a policy and a mechanism to coalesce free pages into a free superpage and vice versa. Buddy systems have been extensively studied, are easy to implement, and can efficiently handle multiple sizes. Foxtrot uses a buddy block allocator. Other alternatives include first-fit, best-fit or worst-fit algorithms that scan the physical page descriptors to find a free page block. A common problem with all variable-sized memory management is external fragmentation. The different algorithms differ in the rate at which memory gets fragmented but all eventually require some form of address space compaction. Another solution is to permanently partition physical memory into pools for each page size [Kagi91]. This is a feasible option in systems that use static page-size assignment but may increase page fault rate if the system does not use the different page sizes in anticipated proportions.

Partial-subblock systems can sometimes allocate sizes that are not powers-of-two but smaller than the page block size, *e.g.*, a 60KB object, and require more general freelist management algorithms. One solution is to allocate a larger power-of-two page block and free any extra base pages, however, this increases external fragmentation.

6.2.2 *Gather Mechanism*

A gather operation copies the contents of base pages corresponding to a virtual page block into a contiguous physical page block and frees the original base physical pages. Systems that use page copying during page promotion to support dynamic page-size assignment policies require a gather mechanism. Using page reservation to properly place physical pages when first allocated may render a gather mechanism unnecessary or would reduce the frequency of required gather operations (described in Section 6.2.5).

A gather operation typically involves the following operations: a) find and lock all the base physical pages, b) remove any mappings[2] to these physical pages from the TLB and page table to prevent other threads/processes from accessing the pages during the copy, c) allocate a physical page block of the required size from the freelist, d) copy the contents of the base pages, and e) unlock and free the base physical pages. The cost of a gather is roughly equal to the cost of the following sub-operations: (s * (page find + page lock + page copy + page unlock + page free + x * (PTE invalidate + TLB shootdown)) + physical page block allocate), where the superpage size is s base pages and x is the average number of aliases per base physical page. The cost of each sub-operation depends on operating system structure, *e.g.*, a page table manager could batch multiple TLB shootdowns, and available hardware support, *e.g.*, hardware support for efficient copying [Yung94].

2. Alternatively the mappings can be marked read-only and removed after the copy is completed.

One complication that arises is that some base pages may be locked while involved in I/O or when pinned in memory by applications (*e.g.*, UNIX **mlock** system call) and cannot participate in a gather operation. This will cause page promotions to fail.

6.2.3 *Page Promotion/Demotion Mechanisms*

Page promotion is the mechanism that coalesces a set of pages to a larger superpage. It involves verifying that all the base pages are superpage-compatible, unloading any existing base page mappings from the page tables and TLBs, allocating contiguous physical memory, and copying the base pages to contiguous memory (a *gather* operation), doing additional I/O (a *populate* operation), and updating page tables and TLBs. A superpage mapping cannot be used until all the base pages within a page block chosen for page promotion are present in memory and page promotion may cause the program to wait while the operating system fetches the missing pages from backing store (or zeroes the page if it is an uninitialized heap page).

The number of base pages brought into memory during page promotion is an additional cost for superpage systems over single-page-size systems. The cost includes use of additional physical memory, additional I/O, and time the program spends in populate operations. However, not all pages brought into memory by the populate operation are wasted. The program may later reference some base pages prefetched by the populate operation—thus avoiding page faults on these prefetched base pages. A populate operation that brings in x base pages is more efficient than servicing x base page faults as the operating system may be able to combine or batch multiple operations to neighboring pages, *e.g.*, disk I/O. The tradeoff, or the threshold at which page promotion is more efficient, depends on the costs of doing I/O and servicing page faults.

Foxtrot reduces page promotion costs in two ways. First, it avoids gather operations by using page reservation. Second, it avoids delay due to populate operations by using prefetching to fetch base pages in the background—overlapping I/O latency with computation. However, prefetching can result in more I/O than in a single-page-size system.

An alternate way to implement page promotion, which I do not implement, is to first perform the populate operation into a newly allocated page block followed by a gather operation of the rest of the base pages. This approach avoids the cost of extra I/O due to prefetching. This can be as efficient as Foxtrot's policy if the operating system can overlap the I/O for page promotion with computation of *other* processes.

Partial-subblock systems do not use page promotion operations as the TLB block includes individual valid bits for base pages and do not require all base pages to be present in memory to share a partial-subblock TLB block. This is a key advantage of partial-subblock systems.

Page demotion is the mechanism that breaks up a superpage into either base pages or smaller superpages. It only involves unloading the superpage mapping from the page table and TLB, possibly replacing it with new base page or smaller superpage mappings. A page-size assignment policy may use page demotion to use smaller page sizes when there is a shortage of free memory. Page demotion also occurs when attributes change for portions of superpages, *e.g.*, copy-on-write (Section 6.3.3).

6.2.4 Monitoring Reference Patterns

A dynamic page-size assignment policy requires the operating system to implement a mechanism to measure the reference pattern of programs to decide the best page-size assignment. The exact information required, however, depends on the policy.

Working-set threshold policies require information about which physical pages are present in memory. This can be retrieved by searching for the physical page descriptors for the base pages corresponding to the virtual address range under consideration. This search can be inefficient, and instead, the page fault handler can maintain counters for each virtual (or physical) page block. The counters are more efficient to search but take up some memory. Foxtrot implements counters in segment drivers to make decisions between two page sizes. A counter scheme similar to Romer *et al.*'s can support multiple page sizes with working-set threshold policies also.

Competitive policies that tradeoff TLB miss costs against page promotion costs require information about which pages are incurring a large number of TLB misses. The TLB miss handler could maintain such statistics, *e.g.*, as proposed by Romer *et al.* [Rome95].

Other policies may monitor characteristics of the system, *e.g.*, amount of free memory, number of free physical page blocks, whether a program is long-lived and could benefit from use of superpages.

6.2.5 Physical Memory Allocation—Page Reservation

Most operating systems carefully select pages to replace, but treat **free** physical pages as interchangeable when allocating a new page. This approach effectively treats physical memory as a fully-associative cache of pages and allocates random physical base pages. With random allocation, page promotions require gather operations. *Page reservation*, a new allocation algorithm I propose, allocates physical pages that are already *properly placed* instead of first allocating random pages and subsequently moving them. Page reservation sets aside a properly placed physical page block for pages that a program may reference soon. If the program references these pages, the memory allocator allocates these reserved and properly placed pages—avoiding the need for a gather operation. When memory demand is high, the memory allocator revokes the reservation on reserved pages that were not referenced within a certain time after the reservation.

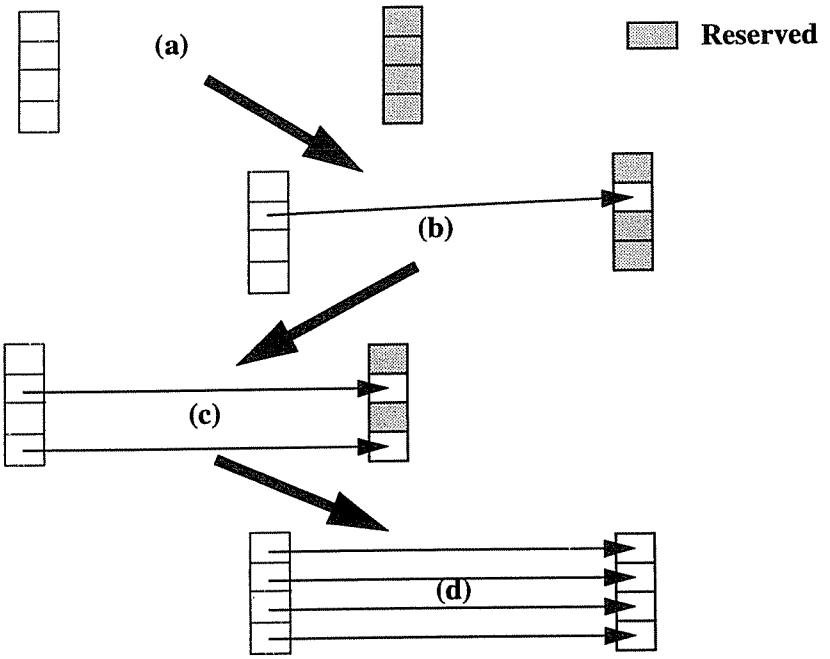
The default physical memory allocator works as follows. The operating system divides physical memory into equal-sized pages, marked as either **free** or **busy**. A **busy** page has the contents of one page of an object (*e.g.*, disk file, heap). The operating system maintains index structures to map physical pages to their identity (<object identifier, offset>) and vice versa. Before allocating a new page, the physical memory allocator searches the index structure to avoid duplicate allocations. If it finds none, it chooses a **free** page and updates the index structures. As more than one process may map the same physical page using different virtual addresses, the physical memory manager uses an unique object page identity instead of virtual addresses.

Page reservation adds a new state for pages—**reserved**. A **reserved** page has an identity and resides in the index structures. However, the contents of a **reserved** page are not valid—similar to an "in-transit" state used during I/O. The operating system maintains **reserved** pages in a *reserved list*—analogous to the free list.

Page reservation works as follows. On the first page fault to a virtual page block, the physical memory manager allocates a physical page block—using techniques such as those described in Section 6.2.1. With page block size 64KB, for example, a page fault to address 0x41034 allocates sixteen base pages for the object pages corresponding to virtual addresses 0x40000, 0x41000, 0x42000,..., 0x4f000. The accessed base page (0x41000) is initialized and marked **busy**. Other base pages are marked **reserved** and added to the end of the reserved list. Subsequent page faults to the same page block will search using the corresponding identity and find these base physical pages **reserved**. The **reserved** physical page will be allocated and marked **busy.** Thus page reservation always places the physical pages at the correct location. Figure 6-1 shows a sample sequence of page faults and pages allocated using page reservation. Some file systems also use similar techniques to reserve disk space [McKu84].

If the physical memory manager cannot find a **free** physical page block, Foxtrot resorts to using random base physical pages. The operating system can use a gather operation to correct these random allocations later when there are free page blocks. An alternate or orthogonal solution would be for the physical memory manager to invoke a memory compactor to free some page blocks when the system is short of free page blocks but has sufficient free base physical pages.

Figure 6-1: Page Reservation



If the physical memory manager runs out of **free** pages, it frees pages by removing them from the reserved list and erasing their identity, *i.e.*, unreserving them. This has two implications. First, the system does not incur additional paging activity if it runs out of free pages but has other reserved pages. Second, for base pages not referenced before the free list becomes empty, the page reservation lapses and later page faults will cause a random page to be allocated. The operating system can use a gather operation to correct these random allocations later when there is sufficient free memory.

Page reservation provides a natural feedback mechanism for improving the effectiveness of superpage and partial-subblock TLBs without unduly increasing memory demand. In peri-

ods of low memory demand, pages will be allocated from **reserved** physical pages, allowing partial-subblock TLB blocks to be shared and subsequent page promotions to proceed without gather operations. In periods of high memory demand, on the other hand, base pages will be rapidly removed from the reserved list and reallocated, gracefully degrading the page allocation policy back to the standard "fully-associative" non-superpage approach. Thus, there is no significant change in the page fault rate from the non-superpage implementation—except for the cost of doing the page reservations.

Since page reservation properly places pages with respect to the size of page block initially allocated, it works best for a partial-subblock system, which has a fixed page block size. It also works well for superpage systems that make decisions between two page sizes. Note, however, that different parts of the virtual address space can choose different superpage sizes for page reservation. Further, page reservation can make decisions between multiple page sizes efficient by avoiding gather costs sometimes, though not always.

For partial-subblock systems, pages allocated through page reservation can share partial-subblock TLB blocks (if attributes match). For superpage systems, page reservation reduces page promotion costs—a key cost of using dynamic page-size assignment policies.

6.2.6 *Multiple-page-size framework*

A question that may be of interest to operating system designers is: How much of a commercial operating system is affected by supporting superpages? I have explored two approaches in implementing Foxtrot—a) changing the virtual memory system to a multiple-page-size framework and b) retaining the original single-page-size framework and implementing the new mechanisms to operate on sets of physical pages. I found it easier to retain the single-page-size framework.

There are two fundamental problems in changing the operating system to a multiple-page-size framework. First, the operating system has the idea of a single page size ingrained at all levels. Most code and data structures assume a constant **PAGESIZE**. Many internal and external interfaces assume a single fixed page size as an implicit parameter (*e.g.*, **vnode** interface [Klei86]). A multi-processor multi-threaded operating system [Camp91, Eykh92, Khal94] has to synchronize concurrent operations and using superpages requires a redesign of the synchronization protocols. Some file systems assume that the page size is smaller than the file block size, and so on. In moving Foxtrot to a multiple-page-size framework, I had to modify large parts of the virtual memory system and file systems[3]. The key data structure change required is to allow physical pages descriptors to describe either a base physical page or a physical page block—much of the operating system operates on physical pages and the physical page structure is often a parameter.

The second problem is with physical pages that have both base page and superpage mappings. When different processes share physical page blocks the operating system may have to support operations on individual base physical pages, *e.g.*, locking one base page, adding a base page mapping to the mapping list. There are at least three ways to support such operations: a) demote the physical page blocks to base pages or b) promote the base page operations to operate on the full page block or c) maintain individual subblock-fields in the page structure. However, UNIX semantics do not always make it possible to demote page blocks or promote base page operations to page blocks, and maintaining subblock-fields is not much

3. I did not complete it as it was easier to implement the mechanisms within a single-page-size framework.

different from using the original single-page-size framework.

The other option is to use the single-page-size framework in the operating system and implement all the new mechanisms to coexist with the old mechanisms. The key is to remember that a superpage size is an attribute of the *virtual* address and not the *physical* address. Superpage and partial-subblock support can be added to an operating system by only requiring the virtual memory system to properly place pages in physical memory—*e.g.*, using page reservation or using gather operations—and a page table manager that can recognize and coalesce base page PTEs into superpage or partial-subblock PTEs. Foxtrot, for example, implements allocating a page block as allocating a set of base pages from the freelist, implements page reservation as adding multiple base pages to existing hash tables, and, implements page promotion as fetching unreferenced base pages within the page block into prereserved base physical pages. This has the advantage that the changes can be localized[4]. It has two disadvantages. First, the operations incur no less overhead than in a single-page-size system. An operating system with a multiple-page-size framework can employ several optimizations, *e.g.*, acquire a single lock for a superpage, or initiate a single disk I/O for a superpage. Second, operations that operate on sets of pages acquire multiple locks and could cause deadlocks. Foxtrot uses a single-page-size framework and required changes to segment drivers and the physical memory layer.

A clustered page table, I propose in Section 7.3, is especially suited for constructing superpage and partial-subblock PTEs in an incremental fashion and for coalescing base page mappings into superpage mappings. Adding superpage mappings to a set of base physical pages also complicates the design of the mappings lists, or synonym table. Section 7.5 describes how a synonym table can be modified to support an arbitrary mix of superpage and partial-subblock mappings to a set of base physical pages.

Partial-subblock TLBs are especially easy to support without using a multiple-page-size framework—they require only variable size freelist management and proper physical memory allocation. Superpage TLBs can use a multiple-page-size framework to implement some operations efficiently but require substantial operating system modifications.

6.3 Interactions with other OS mechanisms and policies

The page-size assignment policy and the mechanisms described in Sections 6.1 and 6.2 are sufficient to incorporate superpage and partial-subblock support in an operating system. The effectiveness of superpage and partial-subblock TLBs can be improved by properly managing interactions with the virtual address allocation policy, shared objects, copy-on-write implementation, file system read-ahead and clustering, page replacement policy, and page coloring that a conventional operating system already implements.

6.3.1 Virtual address allocation

Many operating systems support mapped files or the flexibility to specify starting addresses of segments or both. Choosing the correct virtual address is important to be able to use superpages or partial-subblocking. Assigning starting virtual addresses aligned with re-

4. "Localized" changes is a relative term. In implementing Foxtrot, I had to modify about 100 source files in Solaris 2.1 and rewrote all of the physical page layer and large parts of the virtual memory system. I chose to emulate the superpage TLBs and did not modify the page tables (hat layer). In retrospect, it is possible to restrict the changes to the physical page layer by accepting a slightly inefficient implementation.

spect to the largest superpage size expected to map the object, allows superpages to be used more often than the default random virtual address allocation. Mapping a 4MB frame buffer at VA 0x4000, for example, prevents the use of a 4MB superpage (even if with properly aligned physical pages). The same mapping at VA 0x40000 allows the use of one 4MB superpage. Virtual address allocation is more important than proper physical memory allocation as virtual addresses once allocated cannot be changed—gather operations can correct erroneous physical memory allocations. Paged-segmented architectures [Radi82, Chan90, Lee89b] can reassign virtual addresses by modifying the segment table but cannot avoid the problem completely as the segment offset cannot be changed.

Foxtrot chooses a virtual address aligned with respect to the largest page size that is smaller than the size of the object, if the user does not specify a fixed address. This does not choose the optimal alignment for a growing segment (*e.g.*, heap). Instead, user programs, can use UNIX library **libmapmalloc** (or **mmap** /dev/zero) to allocate large data structures to trigger Foxtrot's heuristics for allocating aligned virtual addresses to increase the effectiveness of superpage usage. The naive solution of always allocating objects at virtual addresses aligned with respect to the largest supported superpage size (*e.g.*, 16MB) is not attractive as it results in very sparse address space usage that can affect page table performance, *e.g.*, in a linear page table.

6.3.2 Shared Objects and Libraries

Many processes share mapped files, dynamically-linked libraries, and SystemV shared memory pages. This raises three interesting problems in page-size-assignment.

First, the first process that maps a shared object (*e.g.*, **libc**) will result in the operating system allocating properly placed physical memory for superpages with respect to the virtual address this process uses. Processes which map the same object later must choose aligned virtual addresses with respect to the superpage mapping established by the first process or else must use base page mappings. Foxtrot's virtual address allocation based on the file size allocates correctly aligned virtual addresses in each process without any inter-process coordination, when mapping the full file.

Second, the dynamic linker in Solaris 2.1 maps only the first page of the shared library and later maps the full file (after reading the header information). This two-step process breaks superpage memory allocation for shared files. On the first mmap of a single base page, page-size assignment would allocate a random base physical page as it cannot distinguish this from any other base page-sized mapped file. When mapping the full file later, this first base page causes page promotion to fail or requires a gather operation. Foxtrot optimizes this common case by doing page-reservation for a full page block when the first page of a file is mapped—this allows the use of a superpage for the full file.

Third, a page-size assignment policy monitors the memory usage or TLB miss rate using virtual addresses of a single process. Shared objects share the same physical pages across multiple processes and a good page-size assignment policy accounts for such sharing while calculating memory usage or TLB miss cost. Most shared libraries can be mapped with superpages as the first process already properly places the file in physical memory. Foxtrot does not account for this sharing as it implements page-size assignment policy in segment drivers, which are per-virtual address space entities. This results in each process independently executing the page-size assignment policy and deciding whether to map the shared file with superpages. An alternate implementation would be to maintain per-file usage counts that can

account for such sharing and incurs less overhead. The disadvantage, however, is the modifications needed to multiple file system implementations, *e.g.*, **ufs, nfs, cachefs, afs, tmpfs, swapfs**.

Shared files also result in multiple virtual addresses per physical page, *aliases*. Alias management becomes complicated if some processes use superpage mappings and others use base page mappings for the same physical pages. Section 7.5 describes three solutions to handle this situation.

The above situations occur when processes sharing pages are simultaneously active. Page-size assignment is also affected when a process uses pages created by another process, *e.g.*, a compiler creates an executable that executes soon after. If the compiler process used random base pages for output files, a gather operation is required to use superpages for the text segment when executing the program. This situation occurs quite frequently but it is not practical to coordinate the virtual addresses and physical addresses used by two non-contemporaneous programs.

6.3.3 Copy-on-write

Many operating systems use the copy-on-write optimization [Rash88] to reduce memory demand by sharing read-only pages until written (*e.g.*, program data segment). A copy-on-write operation remaps a virtual page to a copy of the original physical page adding read-write permissions. As this changes both the protection and PPN for one base page within a page block, a copy-on-write causes page demotion and has two implications on superpage use:

First, the page-size assignment policy must account for possible remapping of base pages in page blocks mapped copy-on-write. If a page block is expected to be rarely written (*e.g.*, program text), then the default policy for the file can be used. If a page block is expected to be sparsely overwritten (*e.g.*, program data segment), then either avoid page promotion or invoke page demotion on the first copy-on-write. If a page block is expected to be completely or mostly overwritten (*e.g.*, **bss** segment), then a copy-on-write operation on the full superpage is more efficient than individual base page copy-on-writes. In partial-subblock systems, the page block is initially properly placed in physical memory. Copy-on-written pages get randomly allocated base pages but other base pages continue to share a single TLB block. A further optimization would be to properly place all the destination copy-on-write pages also. Thus, two partial-subblock TLB blocks suffice—the original, unwritten pages share one TLB block and the written pages share another—while a superpage TLB would use all base page mappings. Foxtrot implements page demotion on first copy-on-write always and does not implement further optimizations.

Second, a copy-on-write operation that results in a page demotion in one process results in a physical page having some superpage and some base page mappings. If the operating system does not support that, one solution is to demote the superpage in all processes sharing the page block, which has the undesirable characteristic of one process affecting other processes' TLB performance. Another solution is to do copy-on-write for the full superpage always, which wastes memory. The naive solution of never using superpages for copy-on-write regions is not acceptable as many text segments allow copy-on-write for self-modifying code or dynamic linking and data segments are inherently copy-on-write. Copy-on-write operations also occur frequently when starting processes using the UNIX **fork** system call.

6.3.4 File system read-ahead and clustering

Operating systems include some file system read-ahead and clustering to prefetch neighboring base pages. There are two reasons for such file system implementations. First, programs are likely to access neighboring base pages soon due to spatial locality and prefetching often helps reduce page fault latency on later page faults. Second, some file systems cluster I/O operations to store files contiguously on disk and prefetching the next sector often comes for free in a disk I/O. This interacts with superpages in two ways:

First, read-ahead operations make it likely that base pages reserved on the first page fault to a page block will be used soon. Thus if the read-ahead is likely to result in bringing in a superpage worth of data, page reservation is a better alternative to allocating random base pages immediately followed by a page promotion and a gather. Solaris 2.1 implements straightforward read-ahead but Foxtrot implements *read-around*—reads base pages that are within the same page block instead of blindly reading from the next page block. For example, on a page fault to VA 0x45000, Foxtrot initiates read-around for VA ranges 0x40000..0x44fff and 0x46000..0x4ffff.

Second, some read-ahead operations result in prefetching base pages that belong to a page block not yet referenced by the program. As Foxtrot does page reservation only on page faults to the page block, these pages may be loaded into random base pages. Foxtrot prevents such read-ahead. Other options include doing page reservation on demand or allocating random base pages followed by a gather operation during page promotion.

6.3.5 Page replacement

LRU-based replacement policies, *e.g.*, Clock [East79], work as in a single-page-size system if superpage mappings duplicate the reference and modified bits in all the base physical page descriptors. Thus, base pages with superpage mappings are treated similarly—all replaced or none replaced. An optimization would be to replace all the base pages in one atomic operation, which may result in efficient disk I/O. If a page block has a mix of base and superpage mappings, the reference bits may be such that it replaces only some base pages within a superpage, causing page demotions.

The main interaction with page-size assignment is that page replacement resets the policy for a page block. Page replacement frees all the mappings and physical memory. A later page fault restarts the page-size assignment policy usage counts. Another option is to remember the old page-size assignment policy outcome and reuse it for the life of the program. This has the advantage of avoiding the overhead of redetermining the optimal page size but has the disadvantage that page blocks once promoted, never get demoted.

Page replacement also could interact with variable-size freelist management to reduce external fragmentation. By preferentially freeing pages that would create free page blocks, for example, page replacement can help freelist management to have more physical page blocks to allocate. I have not explored this interaction or its effect on system performance.

6.3.6 Page Coloring

Page coloring [Tayl90, Kess92, Chiu92] also carefully selects physical pages for virtual addresses but for a different purpose and in a different way than page reservation. Page coloring for physical-indexed physical-tagged caches seeks to reduce cache conflict misses by

partitioning virtual and physical pages into equivalence classes and reducing the probability of allocating virtual pages from different VPN equivalence classes to the same PPN equivalence class. Page coloring for virtual-indexed physical-tagged caches seeks to reduce cache flushes by attempting to allocate physical pages to the same VPN equivalence class as the previous mapping to the physical page.

Page coloring, however, does not attempt to place consecutive virtual pages into consecutive physical pages, as page reservation does. The page coloring algorithm in Solaris 2.1 uses a round-robin scheme but also searches ahead 100 buckets to allocate a page from the least-used bucket. This randomizes physical memory allocation and, even if consecutive page faults occur to consecutive virtual pages, rarely allocates consecutive physical pages. Foxtrot disables page coloring in superpage and partial-subblock systems. I have not studied the effect of this on cache behavior.

6.4 Conclusion

Superpage and partial-subblock TLBs are completely ineffective if operating systems do not support them. Worse, set-associative superpage and partial-subblock TLB implementations have significantly worse performance than equivalent conventional single-page-size TLBs. This chapter makes two important contributions.

First, I identify the operating system policies and mechanisms required to support such TLBs. In particular, a new policy and upto six new mechanisms may be required (Table 6-1). Besides describing alternate policies and implementations for the mechanisms, I list their interactions with existing operating systems policies and mechanisms.

Second, I propose a new physical memory allocation algorithm, page reservation, that allocates physical memory such that superpage and partial-subblocks mappings can be used without incurring the cost of copying base pages into contiguous memory.

Foxtrot, my operating system prototype, implements a functional set of policies and mechanisms to support two page sizes and partial subblocking.

Chapter 7 Page Table Structures

7.1 Introduction

A *page table* stores translation, protection, attribute, and status information for virtual addresses [Huck93, Chan88, Levy82, Silh93, Lee89b]. A *page table entry* (PTE) stores the information for one page. The TLB miss handler accesses the page table on a TLB miss to load the appropriate PTE into the TLB. An ideal page table would facilitate a fast TLB miss handler, use little virtual or physical memory, and flexibly support aliases. Section 7.2 reviews conventional page tables—linear, forward-mapped, and hashed—and discusses the challenges of extending conventional page tables to support 64-bit address spaces. It explains why both linear and hashed page tables are viable, and why forward-mapped page tables are probably impractical as each TLB miss requires about seven memory references. Many processors now support TLB miss handling in software with some hardware assist, *e.g.*, MIPS [Kane92], Alpha [Site93], UltraSPARC [Yung95], PA7100 [Aspr93]. This makes page table design an operating system issue and gives operating system designers more flexibility than traditional hardware-defined page tables.

Section 7.3 introduces the main contribution of this chapter: a *clustered page table.* It is a new page table structure that can be viewed as a hashed page table augmented with *subblocking*, in a manner analogous to subblocking for TLBs (Chapters 4 and 5). Hashed page tables associate a tag with every base page PTE. Clustered page tables associate a single tag for a page block. Clustered page tables are effective when spatial locality makes it likely that consecutive pages are in contemporaneous use. For the assumptions in Section 7.3, for example, clustered page tables with sixteen pages per page block use less memory than hashed page tables if, on average, six or more pages are populated. Experimental results (Table 7-6) show that clustered page tables use less memory than the best conventional page tables—linear page tables for dense address spaces and hashed page tables for sparse address spaces. Clustered page tables will be included in an upcoming release of Solaris, a commercial operating system from Sun Microsystems [Khal95a].

Chapters 3, 4, and 5 described use of *superpages* and *subblocking* in TLBs. These techniques are very effective at improving TLB performance. However, without support in the page table to store superpage and subblock PTEs or a TLB handler to traverse such page tables, these TLB techniques are completely ineffective.

Section 7.4 presents the second contribution of this chapter: extending page tables to support superpage and subblock PTEs. Replicating the superpage or partial-subblock PTEs at each base PTE site extends any conventional page table to support the new PTE formats without affecting TLB miss penalty. I discuss alternate solutions that have drawbacks but are usable in specific situations. I then show how clustered page tables are ideal for supporting medium superpages or subblocks, as they result in smaller page tables, while retaining fast TLB miss handling and flexibility. . When TLBs do not support superpages or subblocking, page tables can use superpage or partial-subblock techniques to reduce page table size by an order of magnitude (Table 7-6)

Operating systems using a private address space model, *e.g.*, UNIX [Thom74], maintain one page table per process or associate a process identifier with each PTE in a shared page table. Operating systems using a private address space model must support mappings for

shared objects, *e.g.*, shared libraries [Ging87a]. When two different virtual pages map to a physical page, the two virtual pages are known as *synonyms* (or *aliases*). An operating system data structure, which I call the *synonym table*, keeps track of these aliases. Section 7.5 explores ways to incorporate superpage and partial-subblock PTEs in a synonym table.

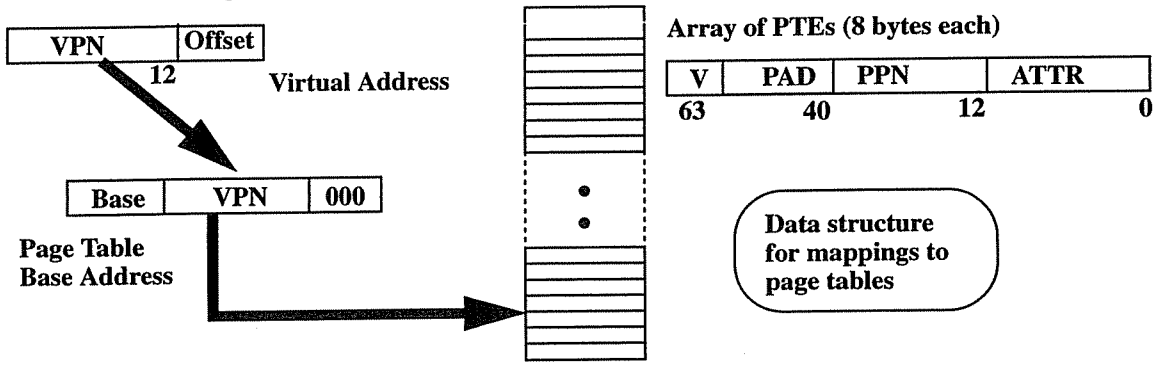
The page table techniques described in this chapter are equally applicable to single address space systems, *e.g.*, Opal [Chas94] or MONADS [Rose85], and segmented systems that use global effective virtual addresses, *e.g*, HP [Lee89b]. Hashed and clustered page tables are especially attractive in these systems as they have a very sparse address space.

Section 7.6 gives performance numbers, where I show that clustered page tables use less memory than any other page table and are faster to access when using superpage or subblock PTEs. Section 7.7 reiterates the contributions.

7.2 Conventional Page Tables for 64-bit Address Spaces

This section reviews commonly-used page tables—linear, forward-mapped, and hashed—and discusses extending them to support 64-bit virtual addresses. A detailed description can be found in Huck and Hays [Huck93]. For all page table designs, 64-bit address mapping information will require eight bytes, *e.g.*, PowerPC [May94], Alpha [Site92], UltraSPARC [Yung95]. The upper-right corner of Figure 7-1 illustrates example mapping information that contains one valid bit, a 28-bit PPN (40-bit physical address with 4KB pages), 12 bits of software or hardware attributes, and PAD bits for future use.

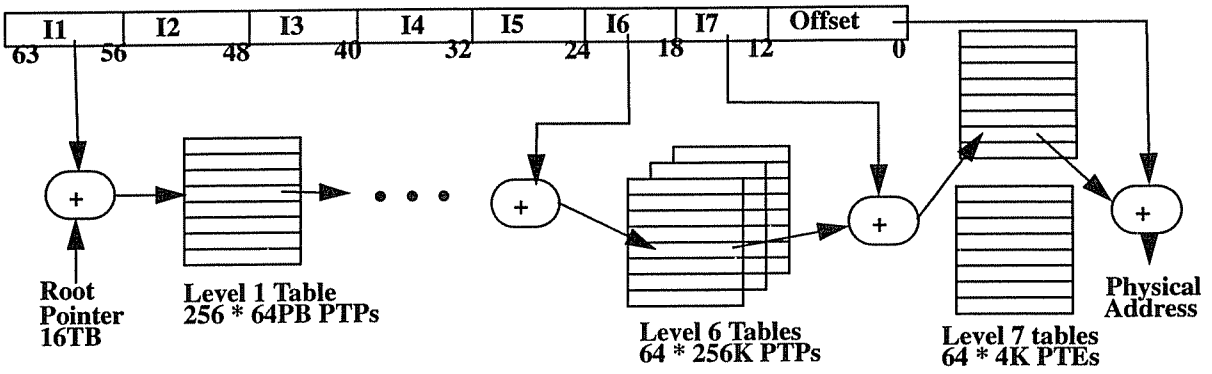
Figure 7-1: Linear Page Table Organizations and PTE format



A *linear page table* conceptually stores all PTEs for a process in a single array. The virtual page number(VPN) indexes the array, as shown in Figure 7-1. Complete linear page tables are very large and are only partially populated. Consequently, they reside in virtual address space, using page faults to populate the table dynamically (*e.g.*, VAX-11 [Levy82], MIPS R4000 [Kane92], Alpha [Site92]). As PTEs are allocated a page at a time, space overhead is high if an address space usage is sparse. A separate data structure stores mappings to the page table itself, *e.g.*, a multi-level tree of linear page tables. Ultrix uses a two-level tree and OSF/1 uses a three-level tree on the MIPS R3000 [Nagl94b]. A straightforward extension of linear page tables to 64-bit addresses uses a virtual array with 4×10^{15} entries and a six-level tree. This design is practical, as a portion of the TLB is reserved for mappings to the page tables [Nagl94b] and the tree is rarely traversed. Alternatively, a linear page table could be backed by other data structures, *e.g.*, a hashed page table or a forward-mapped page table [Site92], described next.

Forward-mapped page tables store PTEs in n-ary trees, with each level of the tree indexed using fixed address fields in the VPN (Figure 7-2). The leaf nodes store PTEs while intermediate nodes store pointers to the next level, page table pointers (PTPs) (*e.g.*, SPARC Reference MMU [SPAR91]). A 64-bit address space extends the number of levels to seven (a 32-bit address space uses three). Forward-mapped page tables are impractical for 64-bit address spaces, as an overhead of seven memory accesses for every TLB miss is not acceptable. There are techniques to short-circuit some levels. Guarded page tables [Lied95] are sometimes effective but would still require three to four levels. An intermediate node cache can accelerate page table access, *e.g.*, PTP cache in SuperSPARC [Blan92], Region Lookaside buffer in HaL [Chan95].

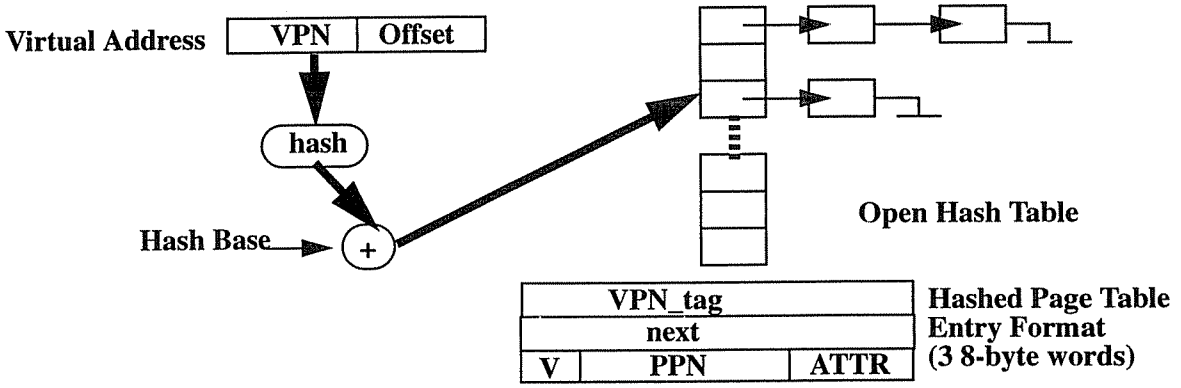
Figure 7-2: Forward Mapped Page Tables



Large address space systems often use *hashed (inverted) page tables* [Lee89b, Chan88, Huck93, May94] as they use memory proportional to the number of active virtual pages[1]. A simple implementation uses an open hash table and a hash function that maps a VPN to a bucket, *e.g.*, $h(PID, VPN) = ((((PID \ll 4) \oplus (VPN)) \bmod (nbuckets)) \times sizeof(PTE)) + HashBase$. Each PTE in the hash table stores mapping information for one base page, a tag identifying the VPN, and a **next** pointer. The hash table handles overflows with open chaining (Figure 7-3). The hash function indexes into an array of hash nodes, the first elements of the hash buckets, and traverses the hash bucket for a PTE with a tag matching the faulting address:

```
for (ptr = &hash_table[h(VPN)]; ptr != NULL; ptr = ptr->next)
        if (tag_match(ptr, faulting_tag))  return(ptr->mapping);
pagefault();
```

Figure 7-3: Hashed Page Tables and PTE format

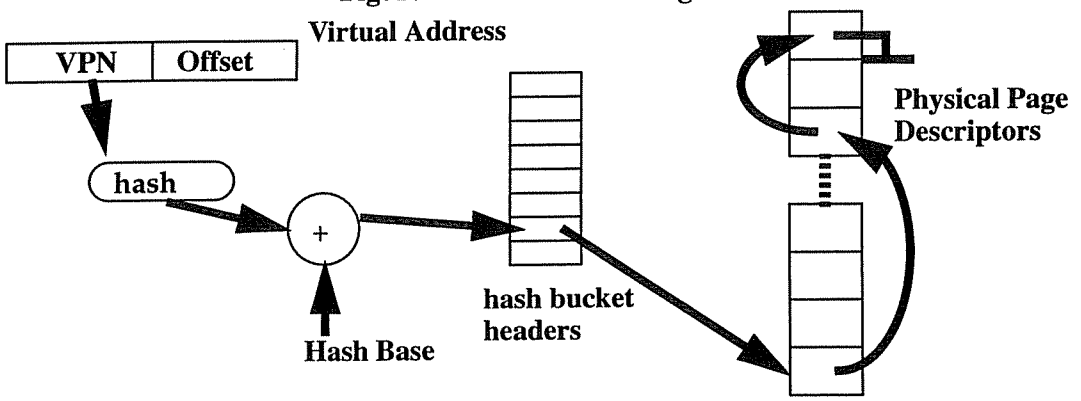


1. In the absence of aliases, hashed page tables use memory proportional to the number of physical pages.

Extending hashed page tables to 64-bit addresses is straightforward. A drawback is that the tag and **next** pointer are now eight bytes each, resulting in sixteen bytes of overhead for each eight bytes of mapping information. One optimization is to pack both into eight bytes by using a shorter **next** pointer and not storing tag bits that can be inferred from indexing the table [Huck93]. This optimization restricts page table placement and can slow software TLB miss handling. I do not consider it further, because clustered page tables—proposed in Section 7.3—offer more effective ways to reduce overhead.

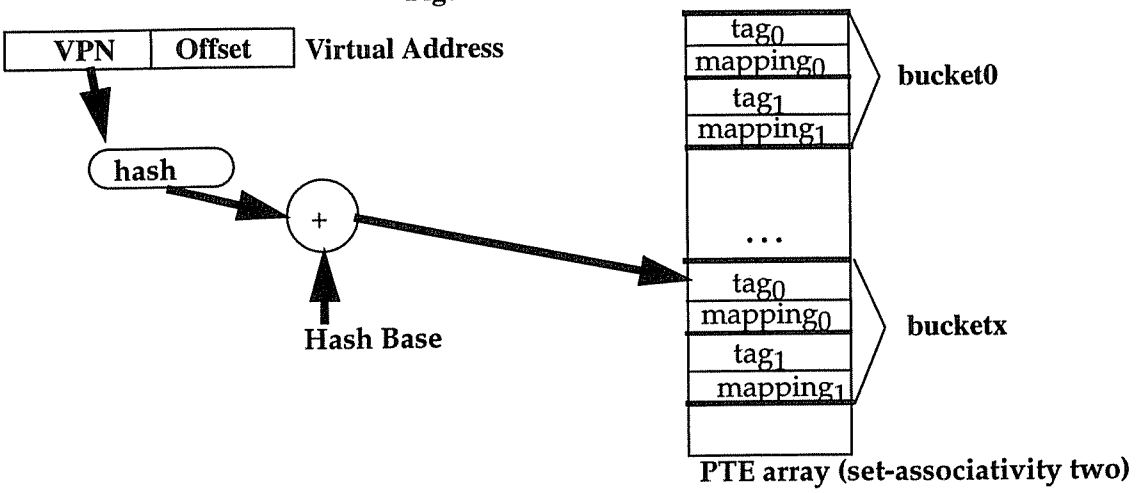
Two variations of hashed page tables include inverted page tables and software TLBs. *Inverted page tables, e.g.,* in IBM System/38 [IBM78], hash to an array of pointers that when dereferenced obtain the first element of the hash bucket (Figure 7-4). The extra level of indirection slows TLB miss handling as it often results in one additional cache miss [Huck93]. There are two advantages of the indirection [Rama81]. Inverted page tables usually use the physical page descriptors as the hash nodes. They can save memory by not storing the PPN in a PTE as it can be inferred from the position of the page descriptor in the array. Also, page table access time improves by dynamically moving the most recently accessed element to the head of the hash bucket list [Rama81, Huck93]. An inverted page table easily incorporates this optimization by maintaining the hash buckets as circular lists and updating the head pointer after every page table lookup.

Figure 7-4: An Inverted Page Table



Software TLBs (*e.g.,* swTLB[Huck93], TSB [Yung94], STLB[Bala94], PowerPC's page table [May94]) eliminate a hashed page table's **next** pointers by pre-allocating few PTEs per bucket. Figure 7-5 shows a software TLB with associativity two. They are so-named, because they can be viewed as memory-resident level-two TLBs with overflow handled in many ways, *e.g.,* hash-rehash schemes [Agar88, Thak86] or set replacement [May94]. While software TLBs can be the native page table structure, *e.g.,* page tables for the PowerPC, they are more popular—and effective also—as a cache of recently used translations. They may reside between the TLB and a native page table to reduce average access time for a slow native page table, *e.g.,* a forward-mapped page table [Huck93, Bala94, Yung95]. The extensions I develop for hashed page table, described next, are applicable to inverted page tables and software TLBs also, as I show in Section 7.4.7.

Figure 7-5: A Software TLB



Which page table should 64-bit systems use? Linear page tables work well when most PTEs in each page of the page table are valid, but perform poorly for sparse address spaces. Hashed page tables have fixed overhead—regardless of whether address space use is dense or sparse—but this overhead is 200% (sixteen bytes for eight bytes). An ideal page table would have the low-overhead of linear page tables in the common case of dense address space use, while retaining the more graceful degradation of hashed page tables for sparse use. I next introduce clustered page tables to achieve this goal.

7.3 Clustered Page Table

Clustered page tables are hashed page tables that store mapping information for several consecutive pages (*e.g.*, sixteen) with a single tag and **next** pointer. Thus, for dense address space use, spatial overheads are much less than with hashed page tables. For sparse address space use, overheads are much less than with linear page tables because few (*e.g.*, sixteen) not many (*e.g*, 512 = 4KB/8B) mappings need be allocated. In addition, clustered page tables perform ideally in cases where several consecutive pages are used together (*e.g.*, medium-sized objects and buffers). This section introduces clustered page tables for 4KB base pages. Section 7.4.5 extends them to work with superpage- and subblock-TLBs.

Clustered page tables use subblocking to extend hashed page tables. Each node in the hash table stores one tag but stores mappings for multiple base pages that belong to the same page block—similar to a complete-subblock TLB (Chapter 4). The number of base pages in a page block is the *subblock factor*. Figure 7-6 shows the format of a clustered PTE with a subblock factor of four and an open hash table constructed using them. Many page table operations are similar to those in a hashed page table. During page table lookup, the virtual page number splits into a virtual page block number (VPBN) and a block offset (Boff). The VPBN participates in the hash function and the block offset indexes into the array of mappings in the PTE with a matching tag. The TLB miss handler is identical with that of a hashed page table when traversing the hash list and differs only *after* finding a PTE with matching tag:

```
for (ptr = &hash_table[h(VPBN)]; ptr != NULL; ptr = ptr->next)  /* hashed page table: h(VPN) */
    if (tag_match(ptr, faulting_tag))
        return(ptr->mapping[Boff]);          /*hashed page table: return(ptr->mapping) */
pagefault();
```

Figure 7-6: Format of Base Clustered PTE (subblock factor 4) and Hash Table

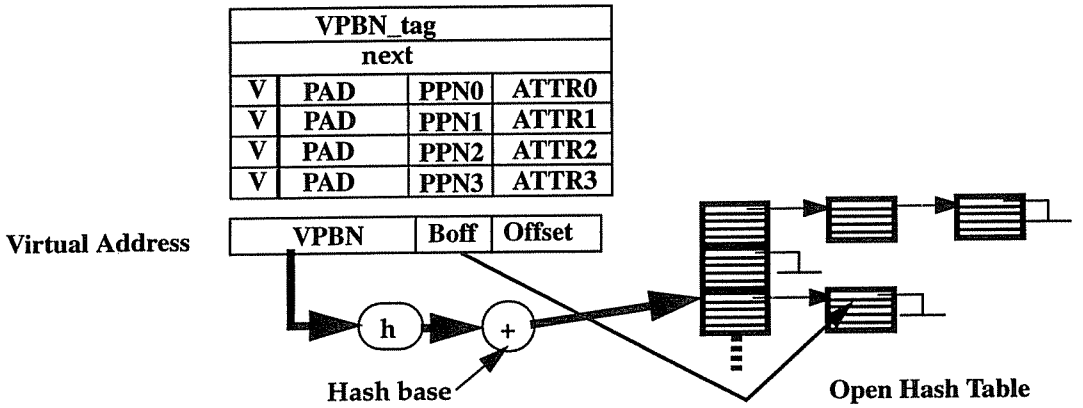


Figure 7-6 uses a subblock factor of four to simplify the illustration. Real implementations may use a larger subblock factor (*e.g.*, sixteen) determined by two issues. First, larger subblock factors reduce memory overhead when most entries are used, but increase memory overhead when mappings are sparse. Second, larger subblock factors pack mappings for consecutive pages close together, improving their spatial locality and potentially reducing cache misses while servicing TLB misses. If the size of the array of mappings is larger than a cache line, however, it may place the VPBN tag and mapping information in two different cache lines, potentially causing an additional cache miss during TLB miss handling.

Clustered page tables have at least four advantages over hashed page tables.

- First, they amortize per-PTE overhead over many potential mappings. Page table size is smaller if enough mappings within a page block are used. For subblock factor sixteen, for example, a clustered page table uses the same memory as a hashed page table when six mappings are used, and about one-third the memory if all are used. This has an analogy to a complete-subblock TLB requiring smaller chip area than a single-page-TLB with the same TLB reach (Section 4.4.1).

- Second, storing mappings for multiple base pages in a single PTE reduces the number of PTEs in a page table. This results in shorter hash table lists, a hash table with fewer buckets, or both. Shorter hash table lists reduce hash table search time on TLB misses [Knut68b, Morr68, John61].

- Third, clustered page tables amortize the overhead of allocating memory for a PTE and inserting in the hash list over multiple PTE insertions. Hashed page tables incur a fixed overhead of memory allocation, list insertion and tag initialization for each PTE added to the page table. A clustered PTE amortizes this overhead over multiple base page mappings that belong to the same page block. This is a significant benefit as page table manipulations are expensive, especially in multi-threaded operating systems where multiple locks must be acquired [Khal94].

- Fourth, operations on a virtual address range are more efficient. The operating system often updates PTEs for a contiguous range of addresses, *e.g.*, unmapping an object or changing protections for a segment. Hashed page tables require one page table traversal per base page, whereas clustered page tables require one per page block. It is also efficient to do range operations in linear and forward-mapped page tables with a linear array scan or a depth-first tree search.

Clustered page tables can perform worse than hashed page tables, however, if address space use is very sparse or if cache performance on TLB misses is worse when tag and mapping information reside in separate cache lines. Experimental results, however, show that the advantages of clustered page tables overcome their disadvantages. Before presenting these results (Section 7.6), however, I next discuss extending page tables to support superpage and subblock TLBs. This will demonstrate additional advantages of clustered page tables.

7.4 Adapting Page Tables for Superpage and Subblock PTEs

This section presents the second contribution of this chapter: discussing page table changes to make superpage and subblock TLBs effective. There are two *potential* advantages of adding support for these TLBs. First, using the new TLBs reduces the number of TLB misses by 50% to 99% (Chapters 2-4). Second, superpage and partial-subblock PTEs (described below) store mapping information more compactly than conventional PTEs, and can decrease page table memory usage. This section examines adapting conventional and clustered page tables to support superpages, partial-subblocking, and preloading into complete-subblock TLBs.

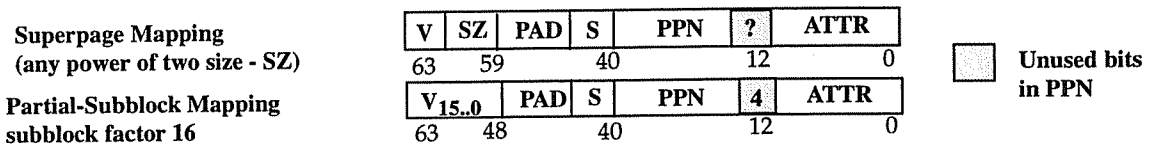
7.4.1 Superpage and Partial-Subblock PTEs

The naive way to service TLB misses in superpage and partial-subblock TLBs stores only base page PTEs in the page table. The TLB miss handler scans PTEs for neighboring base virtual pages to construct superpage or subblock mappings dynamically. As explained in Sections 3.3 and 5.3.1, this is very inefficient and can more than offset any performance benefits from a reduction in the number of TLB misses.

A simple solution is to have the operating system construct and store special superpage and partial-subblock PTEs in the page table. This simplifies the TLB miss handler as it can load a PTE into a TLB without additional processing. As TLB misses occur more frequently than page table updates, the impact of an efficient TLB miss handler is more significant. Further, the operating system often can decide when to use such PTEs, *e.g.*, during page promotion, or can construct such PTEs lazily, *e.g.*, during garbage collection.

Figure 7-7 shows sample superpage and partial-subblock PTEs that I propose storing in the native page table. A superpage PTE adds a size field that specifies a power-of-two size that the PTE maps. A s-bit field can specify one of 2^s different page sizes. Multiple base page PTEs, or a single superpage PTE if possible, map a page block. A partial-subblock PTE adds a valid bit vector that specifies a subset of base pages within a page block that this PTE maps. A s-bit field allows a page block size of upto s base pages. It is possible to map a page block using either multiple base page PTEs, or a single partial-subblock PTE, or multiple partial-subblock PTEs with disjoint valid bit vectors, or a combination of base page and partial-subblock PTEs. The S field—for Subblock/Superpage—distinguishes a partial-subblock and superpage PTEs from base page PTEs and each other, since all reside in the same page table.

Figure 7-7: Superpage and Partial-subblock PTE format (mapping portion)



The new PTE formats, however, require a page table able to store such mappings and a TLB

miss handler that can traverse such page tables. The page tables must support finding a PTE on a TLB miss using the faulting address (without apriori knowing the page size) and without significantly increasing the TLB miss penalty. To the best of my knowledge, current commercial operating systems do not include such page table support, rendering the hardware TLB extensions useless.

Superpage and partial-subblock PTEs significantly reduce page table size and can result in faster TLB miss handling or better CPU cache behavior. A software TLB miss handler allows a page table to store PTE formats different from the hardware TLB block format with some extra work in the TLB miss handler to transform the PTE. For example, using superpage or partial-subblock PTEs with a hardware single-page-size TLB or using superpage PTEs with page sizes larger (*e.g.*, 16MB) than the maximum supported page size (*e.g.*, 4MB) reduce page table size with little affect on the TLB miss handler.

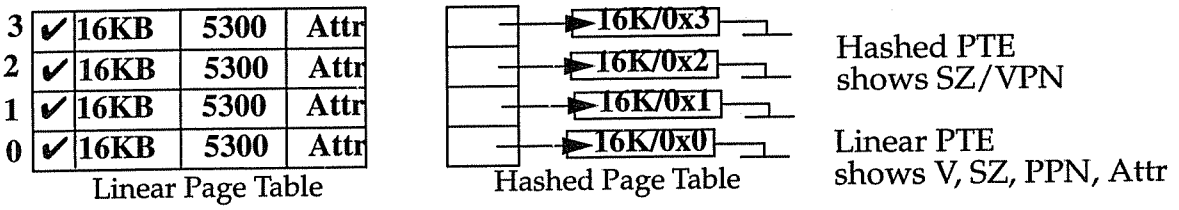
7.4.2 Supporting Superpages

Here I discuss adapting conventional page tables to support superpage PTEs. There are, at least, two solutions for supporting superpages that work for any page table: *Replicate PTEs* and *Multiple Page Tables*.

Replicate PTEs. This solution stores a superpage PTE at the page-table site of every base-page PTE covered by the superpage. Thus, the information for a 64KB superpage gets repeated at sixteen base page sites. On a TLB miss, the handler finds the mapping as if the address has a base page size, but loads a mapping for the whole superpage. Figure 7-8 shows a 16KB superpage PTE, mapping the virtual address range 0x0..0x3fff, stored in a linear and a hashed page table using this approach.

This simple solution is satisfactory. It results in better TLB performance than with conventional TLBs by permitting superpage PTEs to reduce the frequency of TLB misses without affecting the TLB miss penalty. It has two drawbacks. First, it does not allow use of superpages to make page tables smaller. Second, the replicated PTEs make adding a superpage PTE or atomic PTE update more complex, especially in multi-threaded, multiprocessor operating systems [Eykh92, Clar95, May94].

Figure 7-8: Storing superpage mapping for (VA3-VA0) in a hashed page table



Multiple Page Tables. This solution creates separate page tables for each page size in use. On a TLB miss, the handler accesses and searches the page tables in some predetermined order. The page tables probably should be sequenced from the page size most- to least-likely to cause a TLB miss—often the order is the smallest to the largest page size.

This solution appears less good than the first solution. Its principal disadvantage is that it will make TLB miss handling slower, unless most TLB misses go to one page size. Furthermore, the spatial overhead of supporting many page tables mitigates its potential to improve page table size. With linear page tables, PTEs for different page sizes cannot share page table

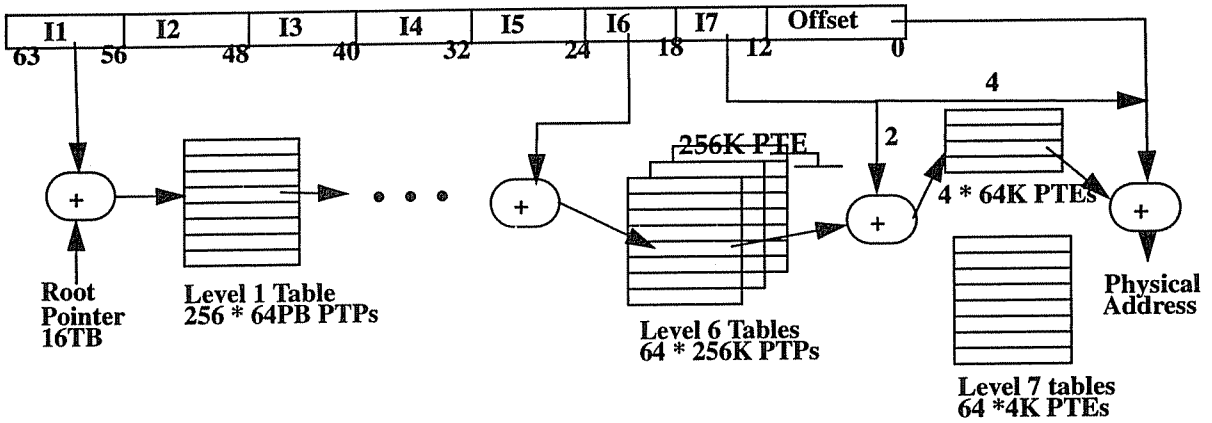
pages. With hashed page tables, hash buckets must be set up for each page size. Alternatively, all logical page tables could share the same buckets at a cost of longer hash chains. This approach maybe attractive for storing superpage PTEs in software TLBs where a superscalar processor can execute a TLB miss handler for two page sizes in comparable time to a single page size TLB miss handler [Kong92, Yung95, Khal93a].

There are also some superpage strategies that only work for specific page tables.

Linear Intermediate Nodes. Linear page tables that use a multi-level tree structure can store superpage PTEs at intermediate tree nodes. With 4KB base pages and eight byte PTEs, for example, each entry in the last level of intermediate nodes points to a page of 512 PTEs. This solution allows the intermediate node entry to store a superpage PTE covering the same virtual space (2MB = 512*4KB).

This solution supports superpages with a modest increase in TLB miss handling time (to decide whether an intermediate node is a superpage PTE or points to the next level). The TLB miss handler would, however, still first access the base page PTE site and incur a nested TLB miss. Its key disadvantage is the lack of flexibility. It only supports page sizes that correspond to intermediate nodes, *e.g.*, 2MB, 1GB, 512GB, 256TB, and 64PB. In particular, it supports no medium-size superpages.

Figure 7-9: Forward Mapped Page Table with Superpage mappings (64KB and 256KB PTEs)



Forward-Mapped Intermediate Nodes. Forward-mapped page tables are a multi-level tree structure and can store superpage PTEs at intermediate tree nodes. A level 6 table entry, for example, could store a 256KB superpage PTE, as shown in Figure 7-9. Furthermore, the tree's branching factor can be altered to support any superpage size. The branching factor of a software-traversed forward-mapped page table is flexible (unlike linear page tables where the number of PTEs per page fixes the branching factor or hardware traversed page tables that have fixed branching factors). In Figure 7-9, for example, the branching factor for level 7 can be either four or sixty-four depending on whether a 256KB region uses 64KB PTEs or 4KB PTEs. Implementing the forward-mapped page table as a B-tree [Come79] allows each intermediate node to map a variable amount of memory and can result in fewer levels. However, a B-tree requires a binary search at each level of the tree instead of indexing with fields from the virtual address. I have not explored the tradeoff further.

Superpage-Index Hashed. One way to support superpages in conventional hashed page tables is to always assume a specific superpage size in the hash function and to associate with

a bucket all appropriate superpage and base page PTEs. Chapter 3 describes a similar scheme for hardware superpage TLBs, where set-associative TLBs support two page sizes using the superpage index. When hashing on 64KB superpages, for example, a particular 64KB region could be mapped by (a) one 64KB superpage, (b) sixteen 4KB base pages, or (c) two 16KB superpages and eight base pages. This would result in one, sixteen, or ten PTEs chained to the same bucket (besides any other PTEs mapping to the same bucket). This solution is not so good, because the longer hash chains will increase TLB miss handling time. In addition, superpages larger than the size selected for hashing must be handled another way. The performance of this solution depends on the ability of the operating system to use superpages. If application or memory constraints restrict use of superpages, the hash lists get very long with base page PTEs and the performance is worse than a conventional hashed page table. Set-associative superpage TLBs show similar performance (Table 3-2).

In summary, the replicated-PTE method is probably the best method so-far for supporting medium-sized superpages in conventional page tables. It decreases frequency of TLB misses *without* increasing the TLB miss penalty. Large superpages can use any of the other methods described also, as there are few such mappings and they miss less often in a TLB.

7.4.3 Supporting Partial-Subblocking

This subsection applies superpage page tables to supporting partial-subblock PTEs (bottom of Figure 7-7). Page table support for partial-subblock PTEs is similar to supporting a base page size and a single superpage size equal to the base page size times the subblock factor. A partial-subblock PTE resides in a page table exactly where a corresponding superpage PTE would have resided. Page blocks that cannot use partial-subblock PTEs can use base page PTEs. Another option is use multiple partial-subblock PTEs or a combination of base page PTEs and partial-subblock PTEs to map a page block. This is analogous to a partial-subblock TLB that use multiple TLB blocks to store incompatible mappings for a page block(Section 5.1). Section 7.4.6 shows more complex optimizations.

The advantages of supporting partial-subblock PTEs over superpage PTEs are four-fold. First, partial-subblock TLBs are more effective than superpage TLBs (Chapter 5). Second, partial-subblock PTEs reduce page table size more effectively than superpage PTEs (Table 7-6). Third, partial-subblocking requires simpler operating system support than superpages (Chapter 6). Fourth, a partial-subblock PTE is a natural intermediate format for page tables that construct superpage PTEs incrementally. The disadvantage is that large subblock factors, *e.g.*, 32 or larger, are not practical due to the limited number of valid bits in a PTE. The extensions described in Section 7.4.2 for superpage PTEs are also applicable to storing partial-subblock PTEs.

Replicate PTEs. This solution stores a partial-subblock PTE with multiple valid bits set at the page table site of every base page PTE covered by the partial-subblock PTE, just as a superpage PTE. This solution is, however, less attractive for partial-subblock PTEs than for superpage PTEs. Superpage PTEs require modification of multiple PTEs only during relatively infrequent operating system directed page promotion or demotion operations. When an operating system adds or deletes base page mappings from the page table, it can create and maintain partial-subblock PTEs incrementally. This requires modifying multiple PTEs on almost every page table operation.

Multiple page tables. This solution creates one page table for base pages and another for the page block size (along with any for larger superpage sizes). Since a partial-subblock PTE

can often store superpage PTEs of a smaller superpage size than the page block size, fewer page tables are needed than the number of superpage sizes supported and improves page table access time. Further, with support for proper physical memory placement, the order of searching the page tables should favor the partial-subblock PTEs over the base page table as partial-subblock PTEs will be accessed more often than base page PTEs.

Linear/Forward-mapped intermediate nodes. Linear and forward-mapped page tables can rarely store partial-subblock PTEs at their intermediate nodes. First, partial-subblock PTEs have a small subblock factor (8 or 16) while common branching factors for the lowest level of the tree are much larger (64 or 512). Second, all the other solutions allow base page PTEs to be used for mappings that cannot share the partial-subblock PTE. Replacing the intermediate node with a partial-subblock PTE forces use of base pages for the full page block if even one mapping is incompatible. While this is no worse than the equivalent superpage PTE solution, it does not exploit all uses of partial-subblock PTEs.

Superpage-index hashed. Partial-subblock PTEs reduce the length of the hash lists even when using superpages. Multiple base page PTEs that could not use a superpage PTE add to the same hash bucket. One or two partial-subblock PTEs can often replace the base page PTEs, shortening the lists. However, when the operating system does not do proper physical memory allocation, long hash lists for base page mappings will still occur.

7.4.4 Preloading Support for Complete-Subblock TLBs

Another hardware technique for increasing the address space mapped by a TLB is complete-subblocking. A complete-subblock TLB requires no special operating system or page table support. On a TLB miss, the handler merely searches any page table for the base page PTE and loads it into the TLB—exactly as in a single-page-size system.

A complete-subblock TLB, however, incurs *block* misses and *subblock* misses (Section 4.2). Block misses allocate a new TLB block, often replacing an old TLB block. Subblock misses add a new PPN and attribute information to an existing TLB block, without causing a replacement. Subblock misses can be eliminated, however, if each block miss preloads all mappings associated with its tag, as the MIPS R4000 does for two PTEs [Kane92]. Subblock preloading never pollutes a TLB by replacing more useful mappings, because it never causes extra replacements [Hill87], but reduces the number of TLB misses significantly (Chapter 4).

A drawback of subblock preloading is the increased time to service TLB block misses. This penalty is large for hashed page tables, as it requires multiple hash probes. This penalty is reasonable for linear, forward-mapped, and clustered page tables, as the additional mappings reside in adjacent page table memory. The penalty reduces further if the clustered PTE format matches the format of the hardware complete-subblock TLB block.

7.4.5 Partial-Subblock and Superpage PTEs in Clustered Page Tables

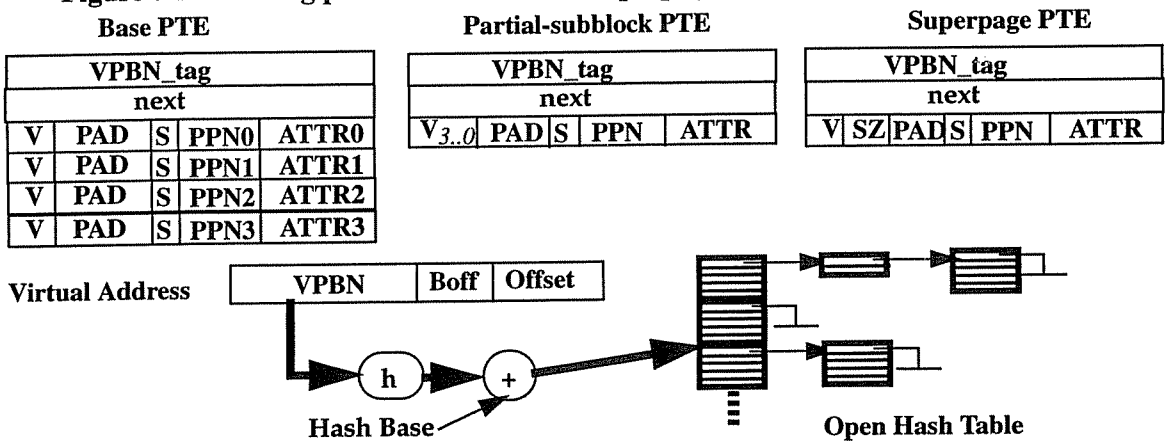
The section first examines incorporating partial-subblocking into clustered page tables. This step is natural, since a node in a clustered page table (for base pages only) strongly resembles a complete-subblock TLB block. This section then incorporates superpages into clustered page tables.

The match between partial-subblock TLBs and clustered page tables is best when both use the same subblock factor. Figure 7-10 (left) illustrates a base clustered PTE with subblock factor

four, and, therefore, has an array of four base page mappings. Figure 7-10 (center) illustrates a partial-subblock PTE. A partial-subblock PTE is used if all the valid mappings within a base clustered PTE are properly placed and have the same attributes. On a TLB miss, the handler hashes on the VPBN and walks the hash chain as usual. On a tag match, the handler consults the new S field and then reads the appropriate mapping. The key here is that the TLB miss handler sees no difference from a base clustered page table while traversing the hash list matching tags and only differs when reading the mapping. Thus, servicing TLB misses to both partial-subblock and base page PTEs does not increase the TLB miss penalty, but uses less memory for partial-subblock PTEs.

```
for (ptr = &hash_table[h(VPBN)]; ptr != NULL; ptr = ptr->next)
        if (tag_match(ptr, faulting_tag))
                return(ptr->mapping[0].S ? ptr->mapping[0] : ptr->mapping[Boff]);
pagefault();
```

Figure 7-10: Storing partial-subblock and superpage PTEs in a clustered page table



Superpage support is also straightforward. Figure 7-10 (right) illustrates support for a medium-sized superpage, whose size is the same as the virtual page block. The superpage PTE is similar to a partial-subblock PTE, except it only has one valid bit. A superpage PTE replaces a base clustered PTE if all mappings in a base clustered PTE are valid and can be condensed to a superpage PTE. The TLB miss handler sees no difference between the three variations of clustered PTEs while checking tags and traversing the hash lists.

Superpage PTEs for page sizes smaller than or equal to the page block size can co-reside in a clustered page table without replication. Further, there is no increase in TLB miss penalty when accessing the superpage PTEs. Smaller superpages use the SZ field to identify them. The above example also could allow a node with two 8KB superpages. One also can mix smaller superpage and base page mappings to map a page block by using multiple clustered PTEs (Section 7.4.6).

Storing superpage PTEs for page sizes larger than the page block size involves a space/time tradeoff as in conventional page tables but clustered page tables are more efficient. Larger superpages can be supported in at least two ways. First, one can use the "Replicate PTEs" solution, but replicate once per clustered PTE instead of once per base page PTE. For subblock factor sixteen, for example, a clustered page table supports large superpages with a factor of sixteen less overhead than conventional page tables. Second, the multiple page table approach

is a reasonable alternative. As explained in the previous paragraph, a single clustered page table can store superpage PTEs with page sizes less than or equal to the subblock factor times the base page size. A second clustered page table can store PTEs for a range of larger superpage sizes (*e.g.*, upto 1MB). Conventional page tables would require as many page tables as the number of page sizes supported, *e.g.*, five in the MIPS R4000.

Supporting superpages and partial-subblocks in clustered page tables offers several advantages over extending hashed page tables. First, its hash chain remains short, whereas hashed page tables require longer hash chains when using base pages. Second, partial-subblock and superpage PTEs reduce both hashed and clustered page table size but clustered page tables do not increase TLB miss penalty whereas hashed page tables do. Third, clustered page tables simplify incremental creation of partial-subblock and superpage PTEs by storing mappings for consecutive base pages together. If the operating system, notices that all base page mappings in a node are valid, it could decide to promote them to a superpage. Gathering this information in hashed page tables is less efficient.

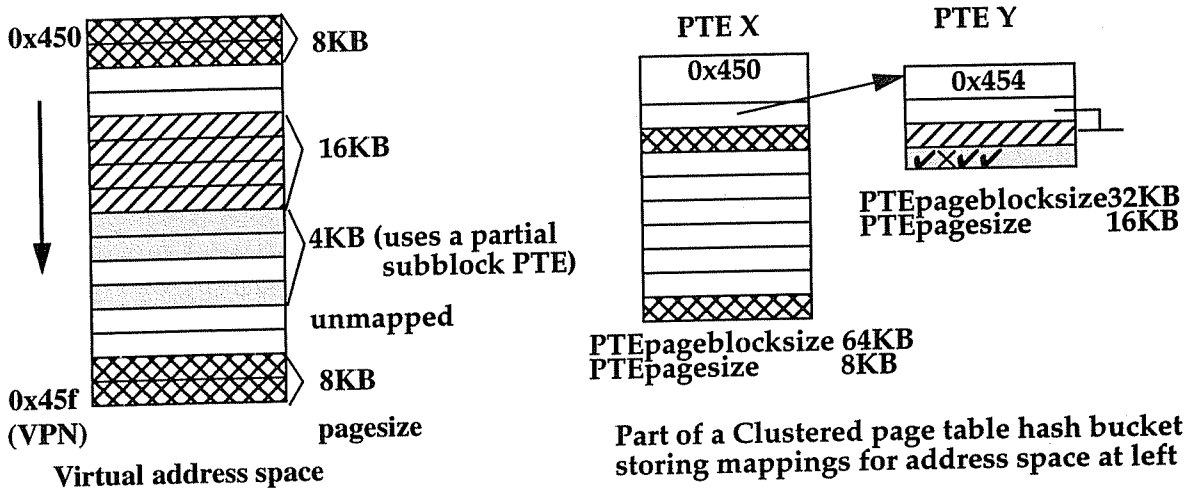
In summary, clustered page tables for base pages use less memory than hashed page tables by combining mappings for neighboring base virtual pages that have nearly identical tags into a single PTE with a single tag. In this section, I took clustered page tables one step further to support superpages and partial-subblocking as well by combining mappings for neighboring base virtual pages that also have nearly identical PPNs into a single mapping.

7.4.6 Generalized Clustered Page Table

Clustered page tables are quite flexible and can store base page PTEs, base clustered PTEs, superpage PTEs, and partial-subblock PTEs—each with different subblock factors and superpage sizes—in various combinations. A generalized clustered PTE has a VPBN_tag, a **next** pointer and an array of mappings each having a fixed *PTEpagesize*—the number of base pages mapped by one mapping. A clustered PTE can either store an array of base page mappings (PTEpagesize of 4KB) or a combination of superpage and partial-subblock mappings (superpage size equal to 4KB times subblock factor of partial-subblock PTE). Mappings that belong to the same page block but have different PTEpagesize use different clustered PTEs, *e.g.*, superpage PTEs of 8KB and 16KB page sizes use different clustered PTEs. *PTEpageblocksize* is the number of base pages mapped by the VPBN_tag—PTEpagesize times the size of the array. The 64KB page block (at VPN 0x450) shown in Figure 7-11, for example, has two 8KB superpage PTEs, one 16KB superpage PTE, and three 4KB base PTEs that can share a partial-subblock PTE with subblock factor 4. At right is a clustered page table storing the mappings in two clustered PTEs—PTE X has a PTEpageblocksize of 64KB with PTEpagesize of 8KB and PTE Y has a PTEpageblocksize of 32KB with PTEpagesize of 16KB. The key to storing arbitrary combinations of PTEs is that clustered page tables can store multiple hash nodes that map overlapping virtual address ranges, *e.g.*, both PTE X and PTE Y could store a mapping for VPN 0x454.

```
for (each of multiple page tables, with different page block size)
  for (ptr = &hash_table[h(VPBN)]; ptr != NULL; ptr = ptr->next)
    if ((ptr->VA <= faultVA) && (ptr->VA + ptr->PTEpageblocksize > faultVA)) {
      pte = ptr->mapping[(faultVA - ptr->VA)/ptr->PTEpagesize];
      if pte_valid(pte, faultVA, ptr->VA, ptr->PTEpageblocksize, ptr->PTEpagesize)
        return(pte);
    }
pagefault();
```

Figure 7-11: Generalized clustered page table (Example)



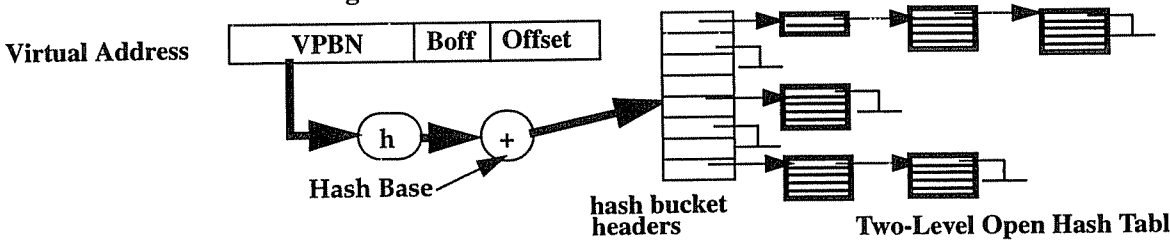
There are four parts of TLB miss handling that an operating system can independently configure. First, the hash function determines a maximum page block size for a hash table. Each hash table stores PTEs with PTEpageblocksize less than or equal to this maximum page block size. Two clustered page tables, for example, suffice to support all combinations of page sizes (or subblock factors) from 4KB to 1MB. The operating system is free to choose any subblock factor for each virtual address region. Second, when traversing the hash list, The TLB miss handler checks each PTE's tag for a match with the faulting address. This can be simplified if all clustered PTEs in a hash table have the same PTEpageblocksize. This does not restrict PTEpagesize and the operating system still has flexibility to choose mappings of any superpage size. Third, if all mappings in a clustered PTE have the same PTEpagesize, simple indexing chooses the appropriate mapping from the array. Fourth, the valid bit in the PTE must be verified. This is required as there can be more than one PTE in the page table that could store the mapping, *e.g.*, both PTE X and PTE Y could have stored the mapping for VPN 0x454 and the search cannot terminate after a tag match that fails to find a valid mapping. Base page, superpage, and partial-subblock mappings differ only in the valid bit check and are otherwise handled identically.

7.4.7 Two-Level and Software TLB variations of Clustered Page Tables

I described clustered page tables, so far, to use an array of PTEs directly indexed by the hash function with open chaining to handle overflow. There are at least two possible variations that may be better suited for some implementations—Two-Level and Software TLB.

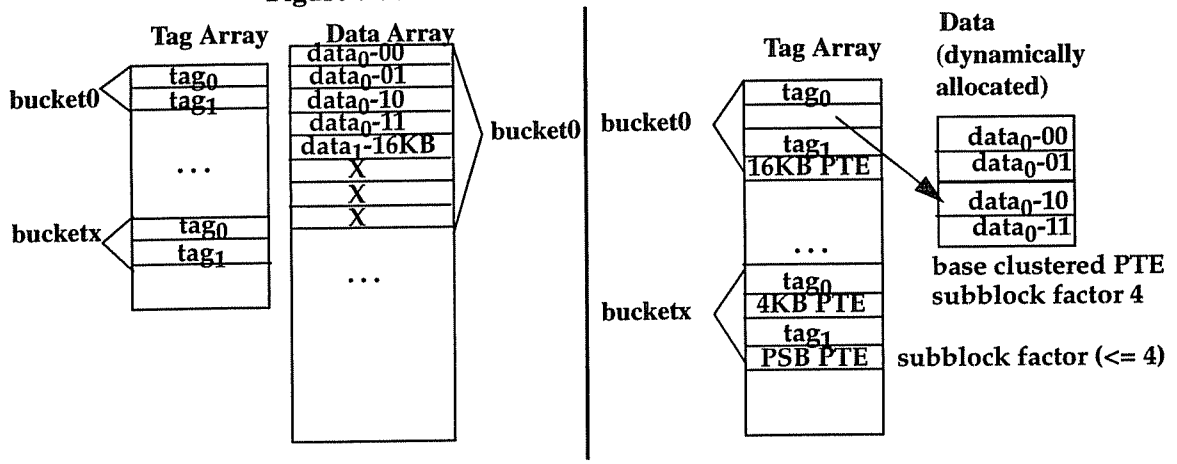
Hash tables can be either one-level (*e.g.*, hashed page tables) or two-level (*e.g.*, inverted page tables), depending on whether the hash function accesses the first element of a hash bucket or a pointer to the first element respectively. Two-level clustered page tables (Figure 7-12) have the disadvantage of taking an extra memory access during page table traversal but have two advantages. First, two-level tables save memory for buckets that are empty or have only superpage or partial-subblock PTEs. A base clustered PTE (*e.g.*, 144 bytes) is larger than a pointer (*e.g.*, 8 bytes), a superpage PTE, or a partial-subblock PTE (*e.g.*, 24 bytes). Second, two-level tables allow easy movement of the most recently referenced PTE to the head of a hash bucket using circular lists [Rama81]. This optimization is important for clustered page tables with single-page-size TLBs where spatial locality makes it likely that mappings for neighboring base pages in the same PTE will be referenced soon.

Figure 7-12: Two-Level clustered page table



Subblocking techniques are also applicable to software TLBs. A clustered software TLB, analogous to set-associative hardware subblock TLBs (Chapters 4 and 5), consists of two preallocated arrays in memory—tag and data arrays. The TLB miss handler first reads the tags from the selected bucket and on a tag match reads the corresponding mapping from the data array. The left half of Figure 7-13 illustrates a clustered software TLB with maximum bucket size two (associativity) and subblock factor 4. Separating the tag and data allows multiple tags in a bucket to fit in a single cache line as the tags are much smaller than base clustered PTEs. While this results in an extra memory access to fetch the data, the number of memory accesses for the tags reduces and can result in fewer overall number of memory accesses. A single superpage or partial-subblock PTE can replace multiple base PTEs corresponding to a tag, as shown in the left half of Figure 7-13. This, however, does not save any memory in the preallocated data array.

Figure 7-13: Base Clustered Software TLB variations



Another way to construct clustered software TLBs stores with the tag a pointer to the mappings for a page block. The right half of Figure 7-13 illustrates this for tag_0 of $bucket_0$ using a subblock factor 4. This scheme seems less attractive than the first scheme as it uses more memory in the tag array. It, however, has advantages when storing superpage or partial-subblock PTEs. The tag can store the mapping itself—instead of a pointer—if a single superpage, partial-subblock, or base page PTE maps a page block (the right half of Figure 7-13). This clustered software-TLB has properties similar to a superpage-indexed hashed page table (or software TLB) but avoids long hash lists (or conflict misses in software TLBs) by using clustered PTEs for page blocks using multiple base page PTEs.

7.5 Synonym Table

A page table and a synonym table are two indices built on a database of translation information, the PTEs. A page table accesses PTEs using a virtual address as the key and uses data structures as described in Section 7.2. A synonym table accesses PTEs using a physical address as the key, *e.g.*, during page replacement to collect reference and modified information or during PTE insertion to determine cacheability of aliases in virtually-indexed caches [Whee92]. This section first describes the basic structure used for synonym tables and then addresses how to store superpage and partial-subblock PTEs in the synonym table.

A synonym table is trivial in a global address space model as it disallows aliases and each physical page descriptor stores the corresponding virtual address [Chan88]. In implementations that support aliases, the synonym table builds a one-to-many relation with a physical page descriptor storing either multiple alias descriptors or a pointer to a list of alias descriptors. An alias descriptor has a pointer to the PTE or a copy of the PTE itself. Each PTE either includes the virtual address or the virtual address can be inferred from the position of a PTE in the page table. Alias lists are straightforward to implement and require one alias pointer (two for doubly linked list) per alias descriptor.

Hardware defined page tables usually do not include enough space in a PTE to store the alias pointer and force the operating system to maintain the synonym table using a pointer to the PTE[2]. This is inefficient to store and update as often a PTE and a pointer are not much different in size, *e.g.*, four or eight bytes. Software-defined page tables can change the PTE format to add alias pointers per mapping. This combines the page table and synonym table in a single data structure. While this increases page table size, by eight bytes per PTE, it saves memory compared to separate tables. Figure 7-14 shows a combined hashed page table and synonym table. The solid lines represent the virtual address hash list pointers and the dotted lines represent alias list pointers.

Figure 7-14: Combined Hashed Page Table and Synonym Table

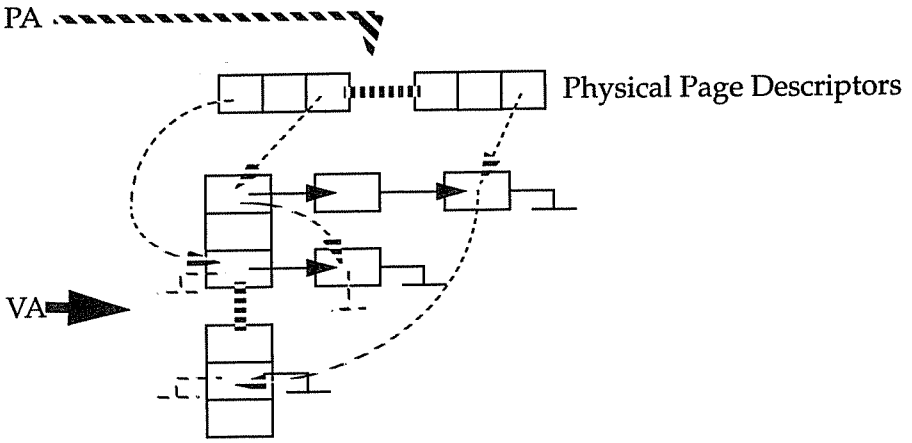
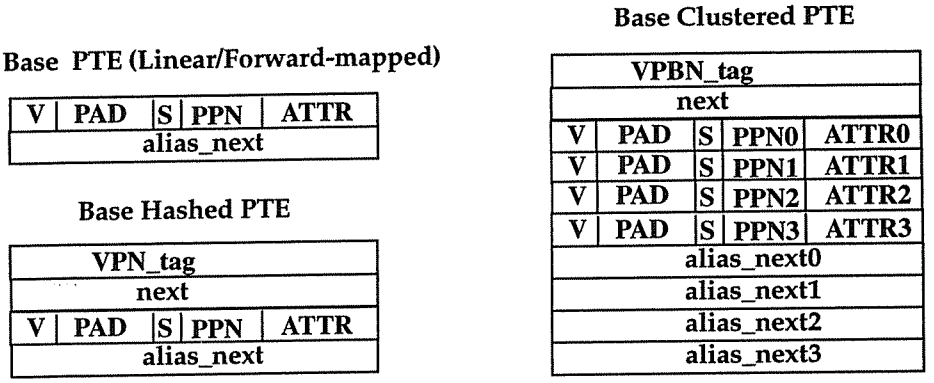


Figure 7-15 shows base PTE formats for different page tables with added alias pointers. A clustered PTE has only one **next** pointer for a page block but has multiple alias pointers for a

2. An alternative approach is to allocate memory for alias descriptors and PTEs such that the address of one can be determined from the other. This can increase internal fragmentation in linear page tables, or restrict dynamic allocation of PTEs or have a worse cache performance than the combined approach.

page block[3]. Adding the alias pointers increases the break-even point for clustered page tables by requiring 9, instead of 6, mappings to be used within a page block for the clustered page tables to use less memory than hashed page tables. In the next two sections, I discuss how to include superpage and partial-subblock PTEs in a synonym table. The solutions I discuss can be used with either separate synonym tables or with the combined page table/synonym table described in this section.

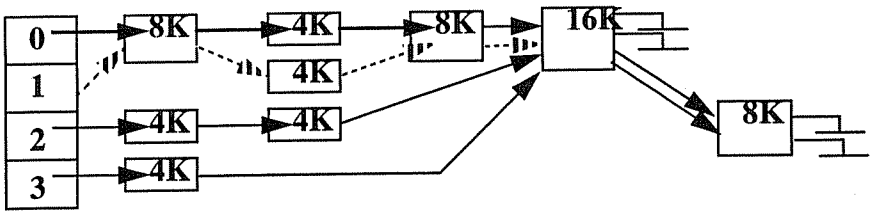
Figure 7-15: Base PTE formats with alias pointers



7.5.1 Naive Synonym tables for Superpage and Partial-subblock PTEs

Superpage and partial-subblock PTEs map multiple physical pages and reside on the alias lists of multiple physical page descriptors. A naive way to include these PTEs in the synonym table is to attach multiple alias pointers, one per base page, with each superpage or partial-subblock PTE. Using the multiple alias pointers per PTE, it is straightforward to store superpage or partial-subblock PTEs in a synonym table. Figure 7-16 shows the synonym table with base PTEs, 8KB, and 16KB superpage PTEs.

Figure 7-16: Synonym table with mixed base and superpage PTEs



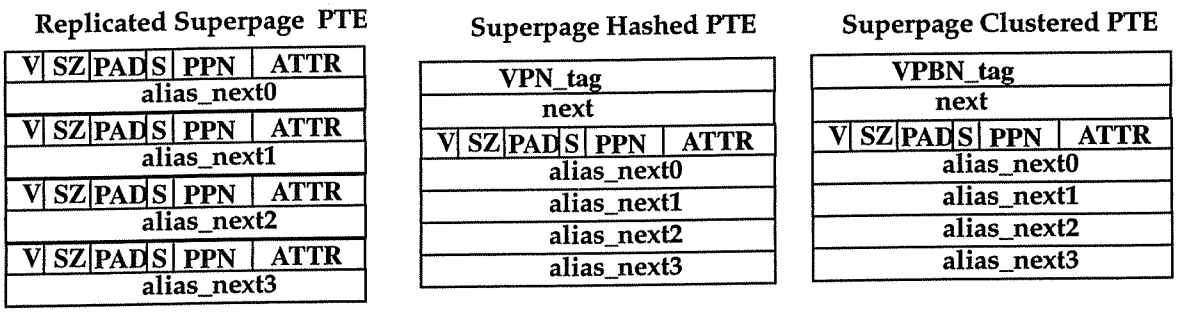
Superpage PTEs stored using the replicated PTE approach, store the corresponding base page's alias pointer in each PTE but replicate the mapping information (left figure in Figure 7-17). Superpage PTEs stored using approaches that allow a single copy add multiple alias pointers per PTE (center and right figures in Figure 7-17). The format for partial-subblock PTEs is identical with superpage PTEs except for the valid bit vector.

Introducing superpage and partial-subblock PTEs (in any page table type) complicates the synonym table in three ways. First, a PTE can be on multiple alias lists and requires storage for multiple alias pointers. Solaris, for example, associates 64 alias descriptors with a 256KB su-

3. The alias pointers are not interspersed with the mappings. This allows more efficient preloading of mappings into a subblock TLB, *e.g.*, all four mappings are less likely to be in a single cache line if the alias pointers were interspersed. This also allows the TLB miss handler to be independent of the alias pointers.

perpage PTE and is part of the alias lists for 64 base physical pages. Second, adding or deleting a PTE may require atomic update of multiple pointers. Third, traversing an alias list requires choosing the correct alias pointer for every PTE. Traversing the alias list for physical page 3, for example, requires choosing **alias3** pointer for the 16K PTE and **alias1** pointer for the 8K PTE. The next section explores ways to reduce alias pointer overhead for superpage and partial-subblock PTEs.

Figure 7-17: Superpage PTE formats with alias pointers

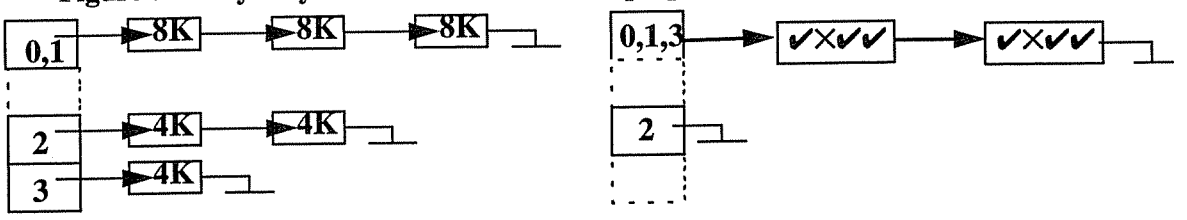


7.5.2 *Alternate ways to store superpage and partial-subblock aliases*

In this section, I discuss three ways to store superpage and partial-subblock PTEs in a synonym table such that they require a *single* alias pointer per PTE. The first restricts physical pages to having mappings with the same page sizes or same subblock valid bit vectors. The next two solutions allow a general alias structure but are more complex to maintain. The main advantage of all these schemes is the significant memory savings from using a single alias pointer per superpage or partial-subblock PTE.

The first solution restricts aliases to a physical page to have the same page size or the same subblock valid bit vectors. The left figure in Figure 7-18 shows a synonym table with three superpage PTEs attached to pages 0 and 1, assuming aligned virtual addresses. The synonym table handles partial-subblock PTEs similarly, as shown at the right of Figure 7-18. The main advantage is the simplicity of the synonym table, which looks similar to a single-page-size synonym table. There are two disadvantages. First, the physical page descriptors correspond to different amounts of physical memory, which requires substantial modifications to an operating system that assumes a physical page descriptor per base page (Section 6.2.6). Second, it does not allow mappings to different sizes to be mixed, *e.g.*, adding an 8K superpage mapping to cover physical pages 2-3 requires either removing existing base page PTEs or demoting the 8K PTE to base page PTEs.

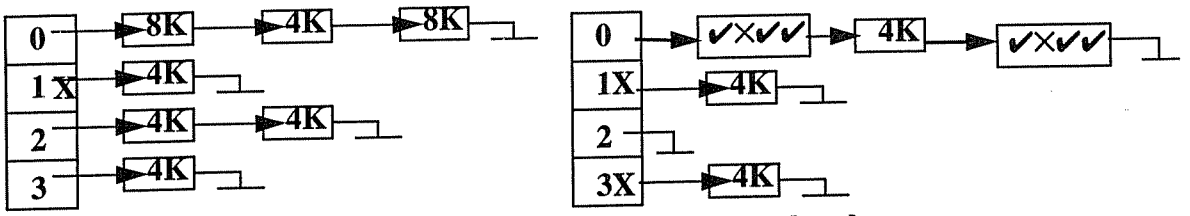
Figure 7-18: Synonym table with fixed size superpage or partial-subblock mappings



The second solution attaches superpage PTEs to only a single physical page descriptor that is part of the superpage and sets flags in other physical page descriptors to indicate that they have superpage mappings not directly attached to their alias lists. The left of Figure 7-19

shows the 8KB superpage mappings attached only to the physical page descriptor of page 0. The physical page descriptor for page 1 has a flag indicating that there are some superpage mappings on page 0's alias list that also belong to page 1. The advantage of this approach is that it allows any combination of superpage or base page aliases for a physical page. The main disadvantage is that multiple alias lists need to be traversed. If a physical page had aliases of multiple superpage sizes, one alias list per superpage size is traversed and each physical page descriptor stores one flag per supported superpage size. Traversing the alias list for physical page 1, for example, encounters some aliases for physical page 0 also and requires additional checks to skip them. The solution can be extended to partial-subblock PTEs also as shown at the right of Figure 7-19.

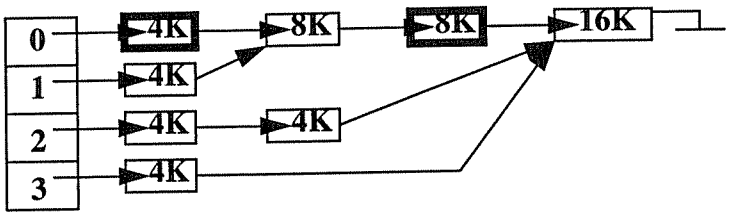
Figure 7-19: Synonym table with mixed base and superpage mappings



X => superpage/subblock PTEs elsewhere

The third solution avoids the multiple list traversals of the previous solution but requires alias lists to be kept sorted in order of *increasing page size*[4]. Figure 7-20 shows the synonym table with aliases of three different page sizes for physical pages 0-3. This approach has two advantages. First, the traversal of an alias list is simple as there is a single path from the head to the tail of the list. Second, it requires multiple pointer updates only when inserting or deleting the first superpage mapping in a list—deleting the superpage mapping in bold is no different from deleting the base page mapping in bold. The main disadvantage is that deleting a superpage mapping sometimes requires updating multiple pointers—with similar complexity to maintaining the flags in the previous scheme.

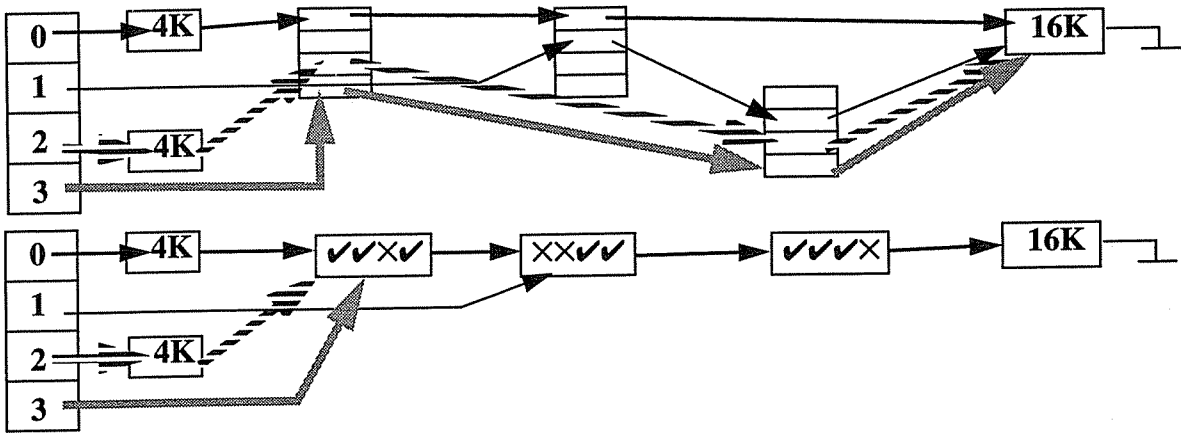
Figure 7-20: Synonym tables with sorted alias lists



The same techniques used for superpages apply to inserting partial-subblock PTEs in the alias lists. Partial-subblock PTEs reside in the sorted list after base PTEs but before superpage PTEs. Figure 7-21 illustrates a complicated example, where the top figure shows the alias lists with multiple alias pointers per partial-subblock PTE and the bottom figure shows the same using the sorted alias list approach. When traversing alias lists, the valid bits must be checked to see if each PTE does have a mapping for the physical page in question.

4. This idea comes from Bill Shannon at Sun Microsystems.

Figure 7-21: Synonym Table example with superpage and partial-subblock PTEs



In summary, a synonym table is an essential part of an operating system that supports aliases. I show that there exists a simple solution to store superpage and partial-subblock PTEs in the alias lists—by maintaining multiple alias pointers per PTE. I also show three ways to structure a synonym table such that superpage and partial-subblock PTEs require a single alias pointer, saving memory but complicating the synonym table management.

7.5.3 Concurrent access to a page table

In a multi-threaded, multi-processor operating system (*e.g.*, [Eykh92]) it is important that the page table and synonym table support concurrent accesses in parallel. The operating system uses a locking protocol to synchronize concurrent accesses (*e.g.*, [Bala92, May94]) and has a significant impact on performance [Khal94]. Some results in the study of concurrent access to database index data structures may be applicable to page tables [Come79, Baye77, Litw93, Elli87, Fagi79, Hsu86, Kuma90, Gutt84]. A page table synchronization protocol has to address at least three issues. First, both the page table and the synonym table must be updated atomically (*e.g.*, with two-phase locking). Second, TLB miss handlers access the page table asynchronously without acquiring any locks, requiring more elaborate page table algorithms [May94]. TLB miss handlers both read (load translation info) and write the page table (update reference/modified bits). Third, all the TLBs in a multiprocessor must be kept consistent with page table updates, requiring a TLB consistency algorithm (*e.g.*, TLB shootdown [Blac89, Tell90]) as part of the synchronization protocol. Addition of superpage and partial-subblock PTEs complicates the synchronization protocol. In practice, certain kinds of TLB-TLB and TLB-PTE inconsistencies are not fatal to the operation of the system and can be allowed. When changing a PTE's permissions from read-only to read-write, for example, TLBs storing a stale copy need not be kept consistent always during the modification. Such optimizations are important as they allow use of more efficient page table synchronization protocols. I do not pursue this topic further.

7.6 Performance Evaluation

This section presents performance results for the page tables discussed so far. I use Foxtrot to obtain estimates of page table access time (the TLB miss penalty) and page table size (Chapter 1). My numbers are approximate, for example, because I do not compute the cache misses saved by a smaller page table. Nevertheless, I show that clustered page tables use less space and can be accessed faster than conventional page tables. Further, their performance rel-

ative to conventional page tables improves even more when supporting superpages or partial-subblocking.

7.6.1 Page Table Access Time: Methodology, Metric & Results

This section extends the trap-driven simulation used for the TLB simulations described in Section 2.1—that counted TLB misses—to also estimate page table access time. Here I assume 64-block fully-associative TLBs, 4KB base pages, and—as appropriate—subblock factor sixteen and superpage size 64KB. I assume 4096 hash buckets in hashed and clustered page tables. Both subblock TLBs—partial and complete—assume preloading in the TLB miss handler. I assume a 256-byte (level-two) cache line size for accessing page tables and later discuss the effect of cache line size. I use Foxtrot's page reservation and page-size assignment policy (Chapter 6) when considering superpages and partial-subblocking.

My metric for page table access time is the average number of cache lines accessed to handle one TLB miss. This metric would be proportional to page table access time if the (level two) cache rarely contains page table data and other overheads are small. There are at least three drawbacks to this metric. First, and most important, it ignores that some page table data may still be in cache, particularly for page tables that are smaller and store PTEs to exploit spatial locality. Yoo and Rogers [Yoo93], for example, observed a 10% improvement in execution time mostly due to cache/TLB effects of reducing the page table size for a commercial database workload. Thus, I expect the access times for clustered page tables, which use less page table memory, to be better than the results reported here. Second, the metric ignores the initial overhead of a TLB miss, but this penalty is independent of page table type. Third, it neglects the time to execute the TLB miss handler instructions to process page table information. This allows the metric to account for hardware TLB miss handlers that typically take time proportional to the number of memory accesses. Even with software TLB miss handling, instruction overhead for hand-coded TLB miss handlers is expected to be small on next generation superscalar processors that can execute three, four, or more instructions per cycle, compared to a main memory access of about a hundred cycles.

I estimate the average number of cache lines accessed on a TLB miss as follows. I modified Foxtrot to maintain hashed and clustered page tables exactly (in parallel with each other and the native page table). On each TLB miss, Foxtrot traps to the operating system, providing the faulting address and an exact page table traversal calculates the number of cache lines accessed. I estimate the cache lines by further assuming each PTE starts on a cache line boundary. Linear page tables always access one cache line and occasionally access higher tree levels. I approximate this by reserving eight of 64 TLB blocks for higher tree levels and assuming each TLB miss to the remaining 56 TLB blocks accesses one cache line. I optimistically assume that TLB misses for intermediate nodes in a linear page table incur zero memory accesses. I assume forward-mapped page tables access one cache line for each tree level without using any short-circuiting optimizations. When storing superpage and partial-subblock PTEs, I assume that linear and forward-mapped page tables use the replicate PTE approach, and hashed page tables use separate page tables for 4KB base pages and 64KB page blocks, with the 4KB page table searched first.

I discuss page table access time (on TLB misses) for various workloads and page tables using the coarse metric: *average number of cache lines accessed to handle a TLB miss.* Each table assumes a different TLB design. Table 7-1 assumes a conventional single-page-size TLB, *i.e.*, no TLB support for superpages or subblocks. Results show that forward-mapped page tables per-

form unacceptably but other designs are similar. This is not surprising since the metric does not reward the more-compact clustered page tables. Clustered page tables have shorter hash lists than hashed page tables reducing the number of accesses, *e.g.*, **ML** and **compress**. Results for linear page tables are optimistic due to the assumptions discussed in Section 7.6.1.

Table 7-1: Average Number of cache lines accessed (single page size TLB)

Workload	Linear	Forward-mapped	Hashed	Clustered
coral	1.06	7.00	1.02	1.01
nasa7	1.16	7.00	1.01	1.00
compress	1.30	7.00	1.41	1.00
fftpde	1.00	7.00	1.03	1.03
wave5	1.01	7.00	1.00	1.00
mp3d	1.05	7.00	1.01	1.00
spice	1.50	7.00	1.00	1.01
pthor	1.03	7.00	1.05	1.00
ML	1.18	7.00	1.20	1.01
gcc	1.36	7.00	1.00	1.00

Table 7-2 present results when the TLB and page table support superpages. Not shown is that the use of superpages reduces TLB miss frequency by 50% to 99%, which is the main reason for supporting superpage PTEs in the page table. Table 7-2 shows the average number of cache lines accessed by the remaining misses. Results are modestly worse for linear page tables (due to the availability of fewer TLB blocks), unchanged for forward-mapped, and much worse for hashed page tables. Hashed page tables take longer to access superpage PTEs as I first search the 4KB page table and then the 64KB page table. For example, the poor performance of hashed page tables for **coral** and **fftpde** is due to a higher fraction of TLB misses to superpage PTEs than for **gcc** or **compress**. Results for clustered page table continue to be close to 1.0, showing that they handle the remaining TLB misses without increasing TLB miss penalty..

Table 7-2: Average Number of cache lines accessed (4KB/64KB superpage TLB)

Workload	Linear	Forward-mapped	Hashed	Clustered
coral	1.10	7.00	2.28	1.03
nasa7	1.59	7.00	1.67	1.02
compress	1.43	7.00	1.59	1.22
fftpde	1.02	7.00	2.28	1.05
wave5	1.03	7.00	1.44	1.03
mp3d	1.26	7.00	1.80	1.02
spice	1.63	7.00	1.74	1.05
pthor	1.09	7.00	2.34	1.02
ML	1.35	7.00	1.98	1.04
gcc	1.83	7.00	1.36	1.01

Table 7-3 presents results for a partial-subblock TLB. To the first order, they are similar to results using a superpage TLB. However, as the workloads use partial-subblock PTEs more often than superpages, hashed page tables have worse performance. For these workloads tra-

Table 7-3: Average Number of cache lines accessed (partial-subblock TLB subblock factor 16)

Workload	Linear	Forward-mapped	Hashed	Clustered
coral	1.10	7.00	2.16	1.03
nasa7	1.37	7.00	2.22	1.05
compress	1.02	7.00	3.05	1.07
fftpde	1.02	7.00	2.17	1.06
wave5	1.03	7.00	2.20	1.02
mp3d	1.33	7.00	2.10	1.02
spice	4.11	7.00	2.17	1.14
pthor	1.11	7.00	2.30	1.01
ML	1.47	7.00	1.86	1.03
gcc	1.38	7.00	2.20	1.01

Table 7-4: Average Number of cache lines accessed (complete-subblock TLB subblock factor 16)

Workload	Linear	Forward-mapped	Hashed	Clustered
coral	1.10	7.00	16.82	1.02
nasa7	1.25	7.00	32.06	1.04
compress	1.08	7.00	48.38	1.01
fftpde	1.01	7.00	17.71	1.05
wave5	1.01	7.00	21.91	1.02
mp3d	1.30	7.00	18.11	1.02
spice	1.50	7.00	36.12	1.23
pthor	1.10	7.00	17.50	1.00
ML	1.42	7.00	19.08	1.02
gcc	1.00	7.00	27.81	1.00

versing the page tables in reverse order—the 64KB page table followed by the 4KB page table—would be a better option. Further, with partial-subblock TLBs, linear page tables could have used fewer reserved TLB blocks. Partial-subblock TLB blocks allow page table allocation size to be 4KB whereas superpage PTEs are usable only after populating all base pages in 64KB of the page table array.

Finally, Table 7-4 gives complete-subblock TLB results, assuming the preloading as described in Section 7.4.4. As expected, hashed page tables perform terribly due to the high cost of multiple probes (sixteen). Linear and clustered page tables continue to be close to 1.0 as they place the mappings for consecutive base pages nearby.

The performance of hashed and clustered page tables can be improved further in two ways. First, the load factor of the hash table can be reduced by increasing the number of hash buckets. Reducing the load factor reduces the average number of hash nodes searched during a traversal but increases the amount of memory used if some buckets are empty. Second, constructing hashed or clustered page tables as a software-TLB can reduce the number of cache lines accessed. A disadvantage of hashed and clustered page tables is the unpredictability of the hash table distribution that depends on the state of the current set of active processes. One solution is to use a per-process or per-process group page table instead of a single shared page table.

A clustered page table's access time is sensitive to the cache line size. A superpage or partial-subblock PTE occupies 24 bytes but a base clustered PTE occupies 144 bytes (subblock factor 16) and may span multiple cache lines. This would increase the average number of cache lines accessed when using base clustered PTEs, *e.g.*, by 0.625 for 64 byte cache lines. However, the good news is that using superpage or partial-subblock PTEs in a clustered page table, even with a single-page-size or complete-subblock TLB, eliminates most of this penalty. Table 7-5 shows the average number of cache lines accessed for different clustered page tables assuming 64-byte cache lines and different TLBs. The superpage and partial-subblock TLBs see modest increases in the average number of cache lines accessed but far smaller than the reduction in the number of TLB misses. Another solution is to use a smaller subblock factor, *e.g.*, 4 or 8, which makes the space/time tradeoff of increasing memory usage to reduce TLB miss penalty. In practice, the performance is better than illustrated here as the expected cache hit rate is higher for clustered page tables, which use less memory.

Table 7-5: Average Number of cache lines accessed for different 64-block fully-associative TLBs with variations of clustered page tables (assuming 64-byte cache lines)

Workload	single-page-size TLB			superpage TLB	partial-subblock TLB	complete-subblock TLB		
	Base	superpage	PSB			Base	superpage	PSB
coral	1.63	1.01	1.01	1.04	1.04	3.02	1.05	1.04
nasa7	1.62	1.02	1.04	1.41	1.31	3.05	2.85	1.60
compress	2.17	1.65	1.65	1.83	1.37	3.05	2.91	1.86
fftpde	1.67	1.09	1.06	1.07	1.08	3.05	1.08	1.08
wave5	1.62	1.01	1.00	1.44	1.11	3.01	2.39	1.19
mp3d	1.63	1.01	1.01	1.31	1.14	3.02	1.72	1.10
spice	1.61	1.01	1.01	1.22	1.53	3.08	2.96	1.57
pthor	1.65	1.06	1.03	1.12	1.07	3.00	1.12	1.02
ML	1.65	1.10	1.09	1.29	1.28	3.02	1.16	1.10
gcc	1.67	1.17	1.15	1.58	1.33	3.00	2.83	1.62

7.6.2 Page Table Size: Methodology, Metric & Results

The next measure of merit of a page table is the page table size, which I measure in a two step process. First, I take a snapshot of each program's mappings—VPNs, PPNs, and attributes—at a point near the program's maximum memory use. Second, I use this information to generate alternate page tables using the following additional assumptions. Mapping information takes eight bytes. Linear page tables use the minimum possible six-level tree. Table 7-6 also shows "1-level" numbers that assume a data structure that takes zero space to store the intermediate nodes. Forward-mapped page tables use a seven-level tree. Hashed and clustered page tables have an overhead of sixteen bytes per PTE to store a tag and next pointer. I compute page table size for multiprogrammed workloads as the sum of page table sizes for the constituent programs. I again assume 4KB base pages, subblock factor sixteen and superpage size 64KB, and Foxtrot's page-size assignment policy.

The first column of Table 7-6 shows that page table sizes are not large enough to cause additional page faults, but they can significantly affect cache behavior. When using a private address space model and per-process page tables, smaller page table size for each process translates to significant savings on a large server system with thousands of active processes. Table 7-6 displays relative page table sizes for various workloads—normalized by the size for a

hashed page table using only base page mappings. The size of a hashed page table is directly proportional to the number of active virtual pages. Base clustered page tables use less memory than the best conventional page tables for all the workloads. For dense address spaces, *e.g.*, **coral, ML, kernel**, a clustered page table is comparable or better than linear and forward-mapped page tables. For sparse address spaces, *e.g.*, **gcc** and **compress**[5], clustered page tables use less memory than hashed page tables als.

Table 7-6: Memory used by different page tables for 4KB base pages

Workload	Hashed Page Table Size	Linear		Forward-mapped	Hashed	Clustered (subblock factor 16)
		6-level	1-level			
coral	119KB	1.02	0.54	0.64	1.00	0.40
nasa7	21KB	4.27	1.53	2.04	1.00	0.44
compress	8KB	27.63	7.65	12.70	1.00	0.81
fftpde	88KB	1.29	0.64	0.74	1.00	0.39
wave5	86KB	1.32	0.65	0.75	1.00	0.40
mp3d	29KB	3.07	1.10	1.55	1.00	0.42
spice	22KB	4.28	1.66	2.03	1.00	0.47
pthor	92KB	1.23	0.61	0.74	1.00	0.39
ML	194KB	0.54	0.38	0.45	1.00	0.38
gcc	34KB	26.62	8.17	11.82	1.00	0.84
kernel space	186KB	0.65	0.56	0.51	1.00	0.48

Table 7-7 shows the relative page table sizes when storing mappings to multiple base pages in the superpage variation of hashed page tables and superpage or partial-subblock variations of clustered page tables. Use of superpage PTEs in clustered page tables reduces memory usage upto 75% and with partial-subblock PTEs by upto 80%. Further, as Tables 7-2 and 7-3 show, clustered page tables support superpage and partial-subblock mappings without increasing the TLB miss penalty. Hashed page tables also can use superpage or partial-subblock PTEs to reduce page table size—with multiple page tables or multiple probes to the same page table—but with increased TLB miss penalty (Tables 7-2 and 7-3). Corresponding reductions in page table size are not possible in linear or forward-mapped page tables as I assume they replicate superpage and partial-subblock PTEs.

In summary, clustered page tables improve significantly on hashed page tables by supporting superpage and subblock TLB architectures without increasing the TLB miss penalty while reducing page table size.

7.7 Conclusion

As the computer industry makes the transition from 32-bit to 64-bit systems, TLBs and page tables are affected. While linear and hashed page tables are still practical, forward-mapped page tables are not because accessing them is too slow. Linear page tables have fast and simple TLB miss handling but incur significant memory overhead and TLB pollution for sparse address spaces. Hashed page tables seem the logical choice for sparse 64-bit address spaces, but have a large per-PTE memory overhead. This thesis makes two key contributions in the area of page table design.

5. These workloads have a sparse address space as they had multiple active processes, many of which were small. The other workloads measure page table usage of a single program with a large heap.

Table 7-7: Memory used by hashed and clustered page tables for 4KB base pages and 64KB superpages or partial subblocking with subblock factor 16

Workload	Hashed		Clustered (subblock factor 16)		
	Base	superpage	Base	superpage	PSB
coral	1.00	0.12	0.40	0.10	0.08
nasa7	1.00	0.20	0.44	0.17	0.12
compress	1.00	0.63	0.81	0.65	0.32
fftpde	1.00	0.10	0.39	0.09	0.07
wave5	1.00	0.11	0.40	0.10	0.08
mp3d	1.00	0.16	0.42	0.13	0.09
spice	1.00	0.27	0.47	0.22	0.13
pthor	1.00	0.10	0.39	0.09	0.07
ML	1.00	0.12	0.38	0.09	0.07
gcc	1.00	0.71	0.84	0.71	0.40
kernel space	1.00	0.39	0.48	0.27	0.27

The main contribution is a new page table organization, the *clustered page table*, which augments hashed page tables with subblocking to address their disadvantages. Specifically, clustered page tables are hashed page tables that store mapping information for several consecutive pages (*e.g.*, sixteen) with a single tag and **next** pointer. Clustered page tables use less memory than other page table organizations, are often faster to access during TLB miss handling and are flexible to changes needed to support operating system needs.

The second contribution is a study of how to store superpage and partial-subblock PTEs in different page tables. In chapters 2, 3, and 4, I evaluate the use of superpages and subblocking in TLBs to increase the TLB reach of a TLB block. These TLB enhancements are largely useless if page tables and operating systems do not support them with proper memory allocation and TLB miss handling. This chapter shows that there exists a straightforward way to store such mappings in a page table—replicate the mappings—that uses the new TLB architectures to reduce the number of TLB misses and does not increase the TLB miss penalty. This chapter also shows that clustered page tables support medium superpage and partial-subblock TLBs without increasing the TLB miss penalty and—at the same time—reduce page table size. This chapter also shows how a synonym table, in systems that support aliases, can store superpage and partial-subblock PTEs while reducing memory usage.

It remains to be seen if commercial operating systems will incorporate the memory allocation and page-size assignment support needed for these new TLBs. Nevertheless, I suggest the use of superpage and partial-subblock PTEs in a page table even if the TLB does not require such support. The advantage being that using these mappings can result in smaller page tables that are faster to access. Clustered page tables provide natural support to store such PTEs and get the memory savings without increasing TLB miss penalty.

Chapter 8 Conclusion and Future Work

8.1 Conclusions

Various workload, technology, and architecture trends have exposed the limitations of conventional single-page-size TLBs. In particular, cycle time restrictions constrain hardware designers from building large single-page-size TLBs to accommodate several important applications. Hardware designers are supporting superpages as a way to increase TLB reach. Large superpages (≥ 256KB) are useful in some applications. My thesis addresses the issues involved in using medium-sized superpages and suggests two subblock TLB architectures as alternate ways to increase TLB reach.

Superpages can be used only when all base pages within a page block are properly placed in physical memory and have the same attributes. To make this happen, however, requires substantial operating system support, including a page-size assignment policy and upto six new mechanisms. My thesis studies the issues involved in building superpage TLBs, shows their effectiveness at reducing the number of TLB misses, suggests operating system policies and mechanisms that are required to support medium-sized superpages, and revisits page table design (including proposing a new page table) to store superpage PTEs. Most microprocessors support superpages in some fashion. While I am not aware of a commercial operating system that supports them for general use, there is some evidence that upcoming releases of some commercial operating systems will.

Subblock TLBs associate a single tag with a page block and allow base pages within a page block to share a single TLB block. Subblock TLBs are attractive because they require simpler (or no) operating system support and incur fewer TLB misses than medium-sized superpage TLBs—fully-associative subblock TLBs easily support large superpages also.

The straightforward subblock-TLB, which I call a complete-subblock TLB, provides space to store individual base page mappings in each TLB block. Complete-subblock TLBs exploit spatial locality, a natural property of many programs, to reduce the amount of tag memory over a single-page-size TLB of similar TLB reach. Microprocessor designers are unable to use an increasing amount of chip area and transistors available to them to build larger single-page-size TLBs due to cycle time constraints. Complete-subblock TLB designs can use the extra chip area to increase TLB reach without significant increases in access time. In particular, I show that for large chip areas, complete-subblock TLBs are faster to access and incur fewer TLB misses than single-page-size TLBs of comparable chip area. To their advantage, complete-subblock TLBs require no additional operating system or page table support.

My main contribution to TLB architecture is the partial-subblock TLB design. A partial-subblock TLB block stores a single tag for a page block, individual valid bits for the base pages, and a single PPN and attribute field for the page block. If the operating system properly places base pages in physical memory, the base pages can share a single TLB block. Base pages with different attributes or improperly placed base pages can still coreside in the TLB but use a different TLB block. My thesis studies the issues involved in building a partial-subblock TLB, handling TLB misses, providing the operating system support, and storing partial-subblock PTEs in a page table. Partial-subblock TLBs are much smaller and faster than complete-subblock TLBs of equal TLB reach but have comparable performance. Partial-subblock TLBs have similar implementation complexity as superpage TLBs but require simpler operating

system support—only physical memory management—and incur fewer TLB misses than superpage TLBs.

Table 8-1 summarizes the comparison between the different TLBs. All the TLBs (except the n-block single-page-size TLB) have the same TLB reach but have different cost and TLB performance profiles. I rank the rows in each column from 1 through 5, where 1 is the best and 5 is the worst for that metric. Complete-subblock TLBs have a double ranking depending on whether they use preloading. Partial-subblock TLBs assume use of preloading as preloading results in simpler hardware and better performance. TLB miss penalty is same for all the TLBs when using replication to store superpage and partial-subblock PTEs in a single-page-size page table. Set-associative implementations of superpage and partial-subblock TLBs are not attractive when the operating system does not use superpages or properly place pages in physical memory, as they have worse performance than a single-page-size TLB with the same number of blocks and associativity.

Table 8-1: Simplified comparison of the different TLB types

TLB Type	Area	Access Time	Effective TLB reach	#TLB misses	TLB miss penalty	OS support	Set-Assoc TLBs
Single Page Size (n blocks)	1	1	5	5	1	1	YES
Single Page Size (n * s blocks)	5	5	1	1	1	1	YES
Complete-subblock (n blocks, subblock factor s)	4	4	2	4 2 if preloading	1 5 if preloading	1	YES
Partial-subblock (n blocks, subblock factor s)	3	3	3	2	1	4	NO
Superpage (n blocks, s * base page size)	2	2	4	3	1	5	NO

I illustrate the performance advantages of the new TLB architectures by comparing three alternate fully-associative TLBs that occupy comparable area and have comparable access time to a 64-block fully-associative single-page-size TLB. Table 8-2 shows the normalized execution time speedup relative to when using a 64-block fully-associative single-page-size (4KB) TLB—a 62-block superpage TLB that supports a 4KB base page size and a 32KB superpage size, a 57-block partial-subblock TLB with subblock factor 16 and preloading in the TLB miss handler, and a 35-block complete-subblock TLB with subblock factor four and no preloading.

Table 8-2: Key TLB performance results—normalized execution time speedup relative to using 64-block fully-associative single-page-size (4KB) TLB

64-block Single-page-size (4KB) TLB	62-block Superpage (4KB/32KB) TLB	57-block partial-subblock TLB (subblock factor 16)	35-block complete-subblock TLB (subblock factor 4)
1.00	1.18	1.21	1.04

The important conclusion from Table 8-2 is that there are alternate TLB designs to a monolithic single-page-size TLB that are of comparable implementation complexity but can deliver good execution time speedups. The speedups are not gigantic (4% to 21%), however, even with my overemphasis on workloads that spend significant time in TLB miss handling.

The new TLB architectures are important as future 64-bit and object-oriented workloads that have larger and sparser address spaces, may spend more time in TLB miss handling and have potential for higher speedups. Today's microprocessors are used in a range of designs from laptops to servers and it is important that the TLBs support large workloads.

I intentionally chose workloads that spend significant time in TLB miss handling. My results overemphasize the execution time speedups and reduction in the number of TLB misses for workloads and systems that have many small processes. While the new TLB architectures do not help improve the execution time of small and short-lived programs, it is important to note they do not slow them down either.

Another important contribution of my thesis is in operating system design. Superpage and partial-subblock TLBs are largely ineffective if operating systems do not support them. I identify the new policies and mechanisms required to support these TLBs and their interactions with other operating system policies and mechanisms. I have also implemented a working version of the policies and mechanisms in a commercial operating system, Solaris 2.1. I also propose a new physical memory allocation algorithm, page reservation, that places physical pages in their "proper place" when first allocating them instead of allocating them randomly and copying into contiguous memory later.

My thesis also makes important contributions in the area of page table design. I study ways to store superpage and partial-subblock PTEs in conventional page tables. One alternative, replicating the superpage or partial-subblock PTE at each base page PTE, supports the new TLBs without increasing TLB miss penalty. I also propose a new page table, clustered page table, that is smaller in size, has faster access times, and is more efficient at storing superpage and partial-subblock PTEs than conventional page tables. A clustered page table is a hashed page table augmented with subblocking and uses techniques similar to those used by superpage, complete-subblock, and partial-subblock TLBs.

Table 8-3 summarizes the key results of my thesis. There are two factors that influence TLB performance: operating system support, and chip area used for the TLB. If operating system changes are inappropriate, complete-subblock TLBs give the best performance. If the physical memory manager in the operating system can be modified, partial-subblock TLBs will reduce TLB area or perform better than complete-subblock TLBs. While very large superpages are useful, my results show that supporting medium-sized superpages is not worthwhile, because they require more operating systems changes and perform less well than partial-subblock TLBs.

Table 8-3: Key Results

TLB Type	Additional OS support	TLB performance with fixed chip area
Single-page-size	None	Worst
Complete-subblock	None	Medium
Partial-subblock	Best-effort	Best
Superpage	Invasive	Good

8.2 Future Work

My thesis suggests and evaluates revolutionary changes to TLB architecture that increase

TLB reach within chip area and access time constraints. Incorporating such support to increase TLB reach, even in the simplest form as described here, in future microprocessors would nearly eliminate all but compulsory TLB misses for many applications. Use of large superpages would address most of the remaining applications' needs. Much work, however, remains to be done in operating systems to use the increased TLB reach---through support for superpages or partial-subblocking—and page table designs to reduce TLB miss penalty. The operating system and page table studies in my thesis identify the issues involved in implementing the policies and mechanisms to support these TLBs and provide "a" working set of policies and mechanisms that use the larger TLB reach. There are many potential areas of research to explore, a few of which I list in this section.

TLBs can benefit from a significant body of research in cache design. My thesis shows how subblocking, a feature often used in cache design, can improve TLB and page table performance. Other cache optimizations may be reexamined in the context of TLBs.

Operating system support for superpage and partial-subblock TLBs is an open research area. One must study the behavior of the mechanisms, policies and TLBs when the system is paging, *i.e.*, there is insufficient physical memory. Research to date assumes sufficient physical memory to prevent paging. Further research is required to find a page-size assignment policy that incurs low overhead, can choose between multiple page sizes, and can adapt to changing reference patterns or available physical memory. In particular, with an operating system that supports the mechanisms needed for superpages, user-level policies that have intimate knowledge of the workload seem attractive. An important first step in enabling such research is a commercial operating system that implements and supports the mechanisms identified in my thesis.

Reducing the number of TLB misses only addresses part of the problem and ways to reduce the TLB miss penalty are another avenue for research. Two issues are of particular interest—efficient software TLB miss handling and page table design. Software TLB miss handling allows flexible page table design, but pays a penalty over hardware TLB miss handling, *e.g.*, overhead to drain the pipeline, calculating the PTE address in a trap handler. Pipeline designs that handle TLB misses without draining the pipeline or in parallel to other operations can reduce TLB miss handling costs more than increasing the TLB reach does. A linear page table, for example, often takes only one memory access to fetch the PTE—a few cycles on cache hits—but trap overheads can take an order of magnitude more number of cycles! Calculating the address of the PTE requires bit manipulations—additions, bit masking, bit extraction, hashing—that result in sequences of dependent operations with little instruction-level-parallelism in a TLB miss handler. Designing hardware that can do such address calculations efficiently without restricting the software to a single page table design is a topic for research.

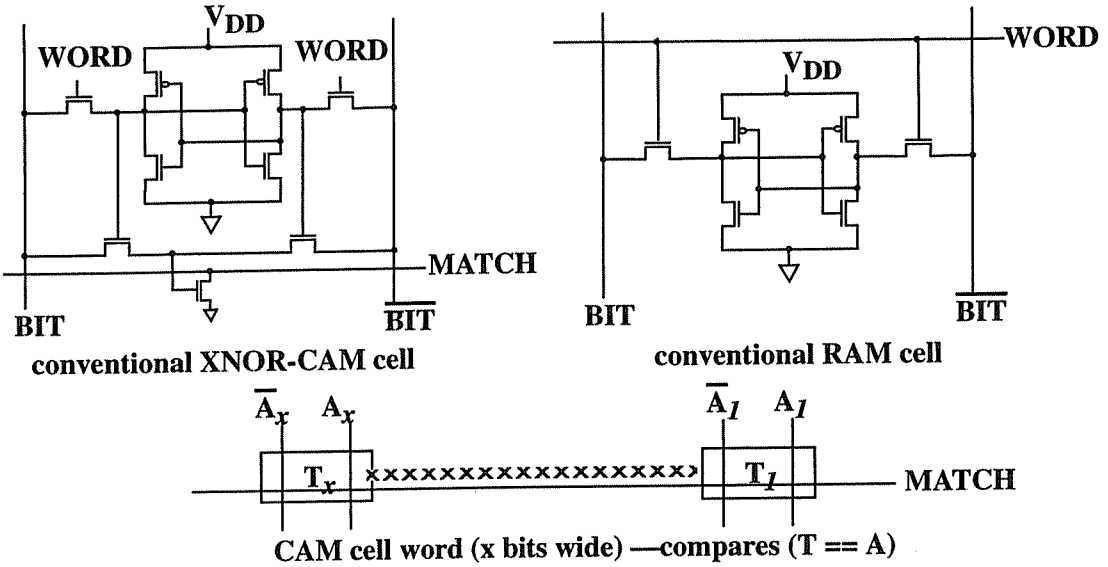
Page table design also includes some interesting areas of research. First, TLB miss handlers (hardware or software) typically access (and update) page tables without acquiring locks. With multi-processor systems and multi-threaded operating systems or applications, an integral part of page table design is a synchronization protocol that updates the page table in a manner consistent with unsynchronized TLB miss handler accesses while maintaining multi-processor TLB consistency—a hard problem. Second, operating systems typically store alias descriptors in a separate synonym table and it remains to be seen if there are benefits to combining the page table and synonym table. Third, new page tables that can store superpage and partial-subblock PTEs could borrow ideas from database research in indices that support range queries.

Appendix A: Sample Memory Cell Designs

This appendix shows some standard custom VLSI circuits that could be used in fully-associative and set-associative TLBs. I illustrate only static designs throughout the thesis. Dynamic designs are more popular. They primarily differ by adding precharge and discharge transistors [West88].

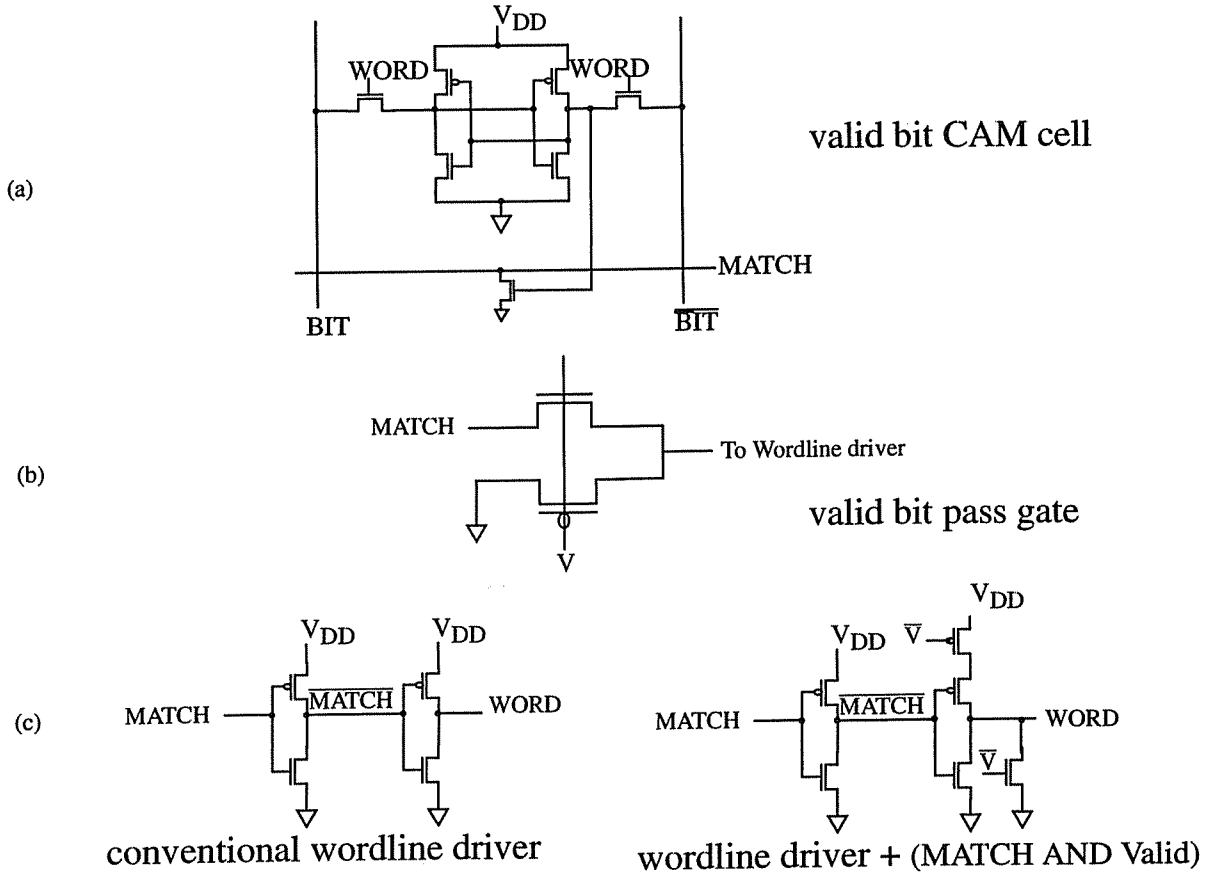
Figure A-1 shows a conventional CAM cell that implements the XNOR function [West88], a 6-transistor RAM cell, and a CAM cell word composed of x CAM cells. Multiple such CAM words combine to form a CAM array—bitlines pass through all the CAM words. The bitlines carry the tag portion of the virtual address to be compared with the contents of the CAM words. The BIT lines carry the inverted address bits and the $\overline{\text{BIT}}$ lines carry the non-inverted address bits. If any of the bits in the virtual address do not match the tag bits stored in the CAM cell array, the previously precharged match line discharges. Therefore, the MATCH line is asserted only if *all* the bits stored in the CAM match the input.

Figure A-1: fully-associative TLB memory cells.



I next show three ways to implement a valid bit in fully-associative TLBs (Figure A-2). A valid bit CAM cell (the top of Figure A-2) can combine with the CAM cell word of Figure A-1 and discharges the MATCH line if the bit stored is 0, *i.e.*, invalid. This, however, increases the capacitance on the MATCH line and slows the tag comparison. Simpler implementations are possible by storing the valid bits in a separate RAM or registers, as the next two options illustrate. The middle of Figure A-2 illustrates the use of a pass gate controlled by the valid bit stored separately [Lee89a]. Pass gates, however, degrade signals passing through them and require more powerful drivers or precharge circuitry. The bottom of Figure A-2 shows a third alternative that combines the valid bit as part of a standard multi-stage wordline driver circuit. Drivers use multi-stage circuitry instead of a single large driver to reduce input capacitance and increase fanout capabilities of the driver. Incorporating simple logic functions, such as combining the valid bit, into a multi-stage driver adds little or no overhead. I model the valid bit CAM cell for fully-associative TLBs and the other designs presented here may be faster and cheaper alternatives.

Figure A-2: Valid bit implementation alternatives (fully-associative TLB)

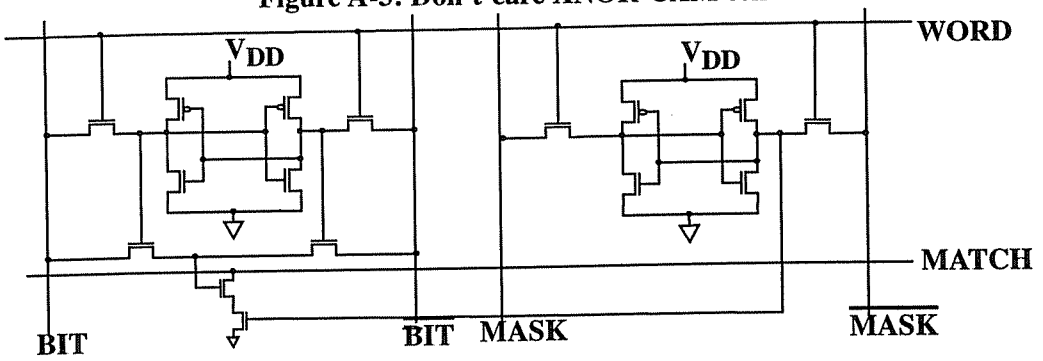


Set-associative TLBs can use similar techniques in the tag comparator. However, if the tag array is on the critical path, a faster access time is possible by storing valid bits in the data array and combining them with the multiplexor driver logic. This optimization is not popular in cache designs where the tag array is often not on the critical path.

A superpage TLB includes don't-care cells for the low-order tag bits. Each don't-care CAM cell (Figure A-3) stores two bits—a tag bit and a mask bit—and implements three states—match, no match, don't-care. If the mask bit is "1" then the cell never discharges the MATCH line, irrespective of the value of the tag bit stored in this cell. Such don't-care cells can be part of the CAM cell word. The additional transistor on the discharge path increases the capacitance and resistance, affecting the RC time constant and access time. Further, the don't-care cells add a significant amount of capacitance due to a longer MATCH line that has to pass through both the bits and affect tag compare time.

The area and timing models in Section 2.2 and Section 2.3 assume that the $\lg_2$(superpage size/base page size) low-order bits use don't-care cells for the MASK bits. Such a configuration can support more than two page sizes—all power-of-two sizes between the base page size and the maximum superpage size. I, however, assume the use of only two of the supported page sizes in superpage TLBs. One can optimize the design when supporting fewer page sizes. When supporting superpage sizes that are multiples of four of each other, for example, a single MASK bit can control two neighboring CAM cells. This halves the number of MASK bits in the CAM cell array. However, it would require larger transistors to drive an increased fanout of the MASK signal.

Figure A-3: Don't-care XNOR-CAM cell



Appendix B: Implementation of subblock-valid bits

Most single-page-size TLB implementations assume that only a single TLB block can succeed in tag comparison during TLB lookup. This assumption allows for simpler circuitry on the data side of the TLB. Hardware or software typically guarantees that two or more TLB blocks will not have the same tag. Further, a valid bit in each TLB block disables any spurious matches due to invalid blocks (Section 1.5). In a subblock TLB there are multiple subblock-valid bits per TLB block. The decoded block offset field of the virtual address selects the appropriate subblock-valid bit. This appendix discusses three alternate implementations of the subblock-valid bits. A simple block-valid bit approach suffices for complete-subblock TLBs, but partial-subblock TLBs require the subblock-valid bits to be incorporated in the tag comparator.

A complete-subblock TLB block can store in the tag a block-valid bit that stores the logical OR of the subblock-valid bits. This suffices to prevent invalid blocks from generating spurious matches. Further, hardware or software must guarantee that more than one TLB block will not have the same tag. Subblock miss checking (Section 4.2), for example, can provide this guarantee. A partial-subblock TLB, on the other hand, must allow for multiple TLB blocks with the same tag, but disjoint subblock-valid bits, to be present in the TLB simultaneously (Chapter 5). Thus, a single block-valid bit in the tag does not suffice to prevent two TLB blocks from succeeding in tag comparison. The subblock-valid bits must be incorporated in the tag comparator.

The block-valid bit approach, which is the best choice for a fully-associative complete-subblock TLB, is to use a single block-valid bit as part of the tag. The block-valid bit stores the logical OR of the individual subblock-valid bits. The block valid bit can be either part of the tag memory or combined with the wordline driver as in a single-page-size TLB (Appendix A). The data RAM stores the subblock valid bits and reads out only the subblock valid bit and mapping corresponding to the subblock (Figure 4-4 in Chapter 4). The subblock valid bit determines if the access was a TLB hit or a subblock miss. On a TLB hit it enables the output driver. The subblock valid bit access and output driver enable path is likely to be in the critical path for TLB access. A set-associative TLB can use the same mechanism if the data RAM access is on the critical path. The block-valid bit approach has area and access time advantages over the two alternatives I discuss next that combine the subblock-valid bits with the tag comparator. The main disadvantage is that multiple TLB blocks with the same tag but disjoint subblock valid bits are not supported—hardware and/or software must guarantee against this happening during TLB miss handling.

For complete-subblock TLBs that do not implement subblock-miss checking and for partial-subblock TLBs, the block-valid bit approach is not sufficient and the subblock-valid bits must participate in the tag comparison. This appendix next discusses two alternate implementations. The first alternative, valid bit tag comparator, stores the subblock-valid bits in the tags and extends the tag comparison logic to use the decoded block offset to select the appropriate valid bit. The tag comparator implements the function: (VPBN == Tag-VPBN) && ((Decoded block-offset & Tag.Valid_bits) != 0)—where VPBN is the tag to be looked up, Tag-VPBN and Tag.Valid_bits are stored in the tag. The second alternative, valid bit RAM, optimizes the design by observing that a decoded block offset is a one-hot encoding, *i.e.*, it chooses only one subblock valid bit. It uses a separate RAM indexed by the decoded block offset to store the subblock valid bits. This allows tag comparison and valid bit selection to proceed in parallel.

Figure B-1 illustrates one way to extend a fully-associative tag comparator to include multiple subblock-valid bits. In fully-associative TLBs, a conventional CAM cell implements the XNOR function. A CAM cell can be modified to discharge the match line if ($\overline{V}$ && $\overline{BIT}$)—where the cell stores value V and the other operand is driven onto $\overline{BIT}$. Subblock-valid CAM cells combine to discharge the match line if the valid bit corresponding to the decoded block offset is not set. The combined tag array, with subblock valid bits and regular XNOR-CAM cells, implements the function: (A == B) && ((C & D) != 0), where C has only one bit set[1].

Figure B-1: Fully-associative Valid bit tag comparator array

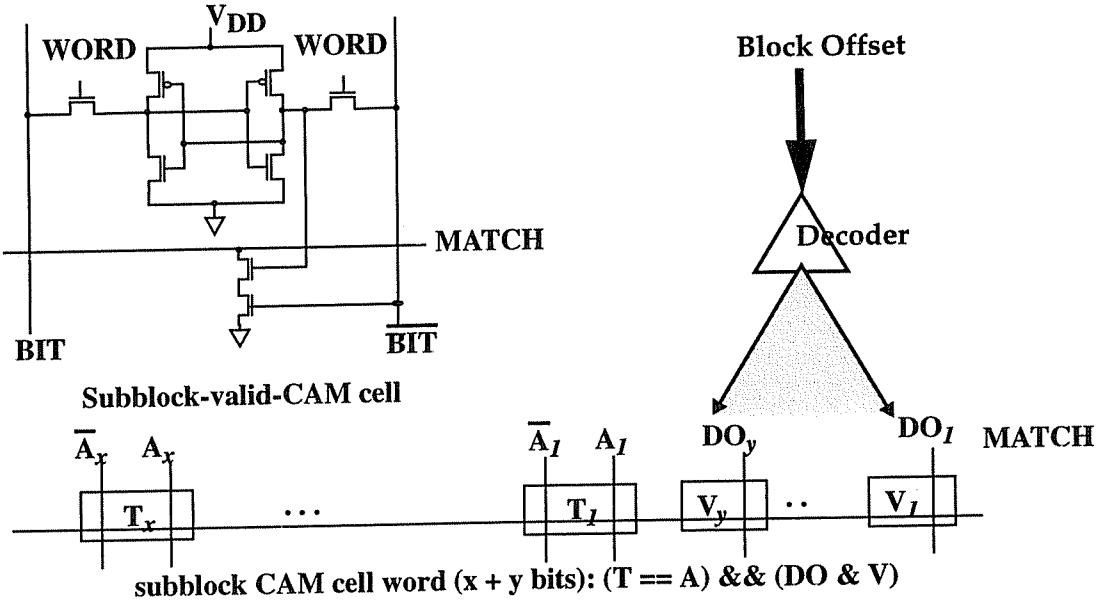
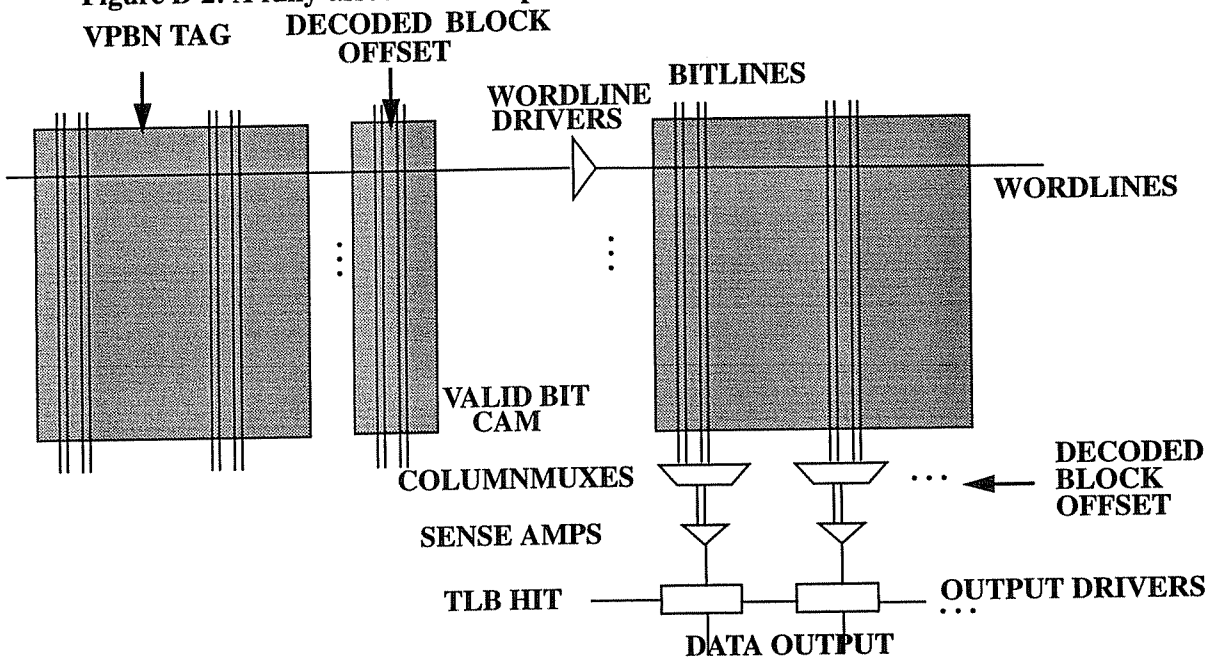


Figure B-2: A fully-associative complete-subblock TLB with valid bit tag compares



1. A Decoded offset, by definition, has only one bit set. The circuit behaves as follows in other situations: If a decoded offset has no bits set, the valid bits do nothing and the tag match is the result of the XOR. If a decoded offset has more than one bit set, the match succeeds only if ALL the corresponding valid bits are set.

Figure B-2 illustrates a fully-associative TLB implementation of the valid bit comparator approach. The tag memory stores Tag.VPBN in standard XNOR-CAM cells and subblock valid bits in the special CAM cells. The tag to be looked up is driven to the XNOR-CAM bitlines and the decoded block-offset bits are driven to the subblock-valid-CAM $\overline{BIT}$ lines (Figure B-1).

A set-associative TLB's tag compare circuitry can be similarly modified to account for the subblock valid bits (Figure B-3). The tag bits ($t_1..t_x$) read out of the tag RAM are compared with the VPN bits ($a_1..a_x$). The valid bits ($v_1..v_y$) also read out of the tag RAM, inverted, combine with the decoded block offset bits ($do_1..do_y$) to complete the MATCH function. Figure B-4 shows the structure of the set-associative TLB with valid bits moved to the tag RAM and the comparators replaced by the special comparator array shown in Figure B-3. The final implementation is very close to that of a single-page-size set-associative TLB (Figure 1-5).

Figure B-3: Set-associative tag comparator with valid bits in the tag

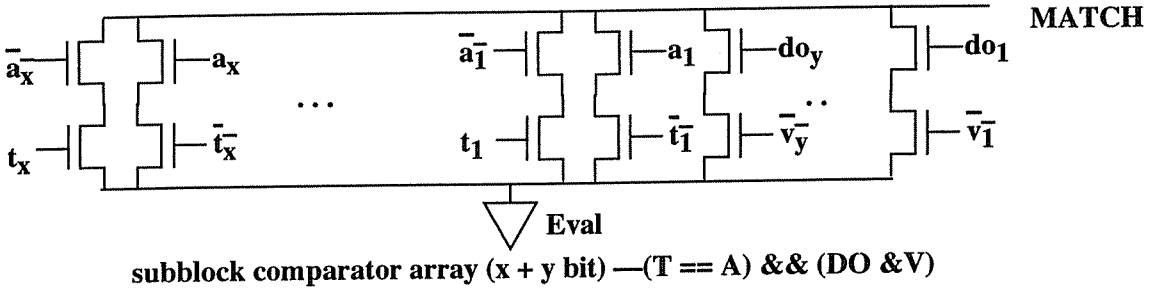
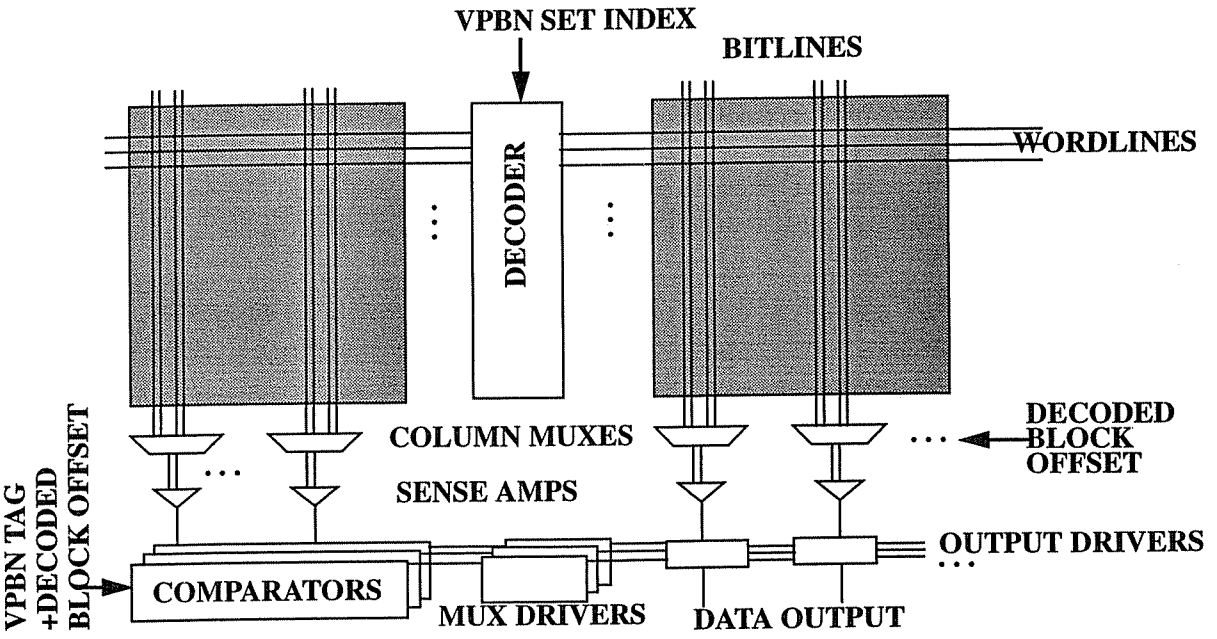


Figure B-4: Structure of a set-associative complete-subblock TLB with valid bits in the tag



Combining the valid bits with tag comparison, however, increases the TLB access time as TLB tag comparison is typically in the critical path. Table B-5 illustrates the percentage reduction in TLB access time for complete-subblock TLBs using the block valid bit compared to using the valid bit tag comparators. Note that the access time estimate in Table B-5 does not include the time taken to decode the block-offset field, which must be done BEFORE tag com-

parison can begin[2] in fully-associative TLBs. In a set-associative TLB, the decode occurs in parallel to the tag-RAM access.

Table B-5: Reduction in access time for complete-subblock TLBs with block valid bits compared to implementation with subblock-valid bit tag comparators

TLB	subblock factor			
	2	4	8	16
64-block fully-associative	0.2%	0.7%	1.6%	3.3%
128-block fully-associative	0.2%	0.6%	1.5%	3.1%
256-block fully-associative	0.1%	0.6%	1.4%	2.9%
512-block fully-associative	0.1%	0.5%	1.2%	2.5%
128-block 4-way set-associative	0.2%	0.9%	2.3%	4.5%
256-block 4-way set-associative	0.2%	0.9%	2.3%	4.4%

The valid bit comparator described above is, however, an overkill. Only one subblock valid bit is required to participate in tag comparison. This observation allows an optimization that uses a separate valid bit RAM to store the subblock-valid bits and combines the selected valid bit with the tag compare output. The decoded block offset field indexes this RAM.

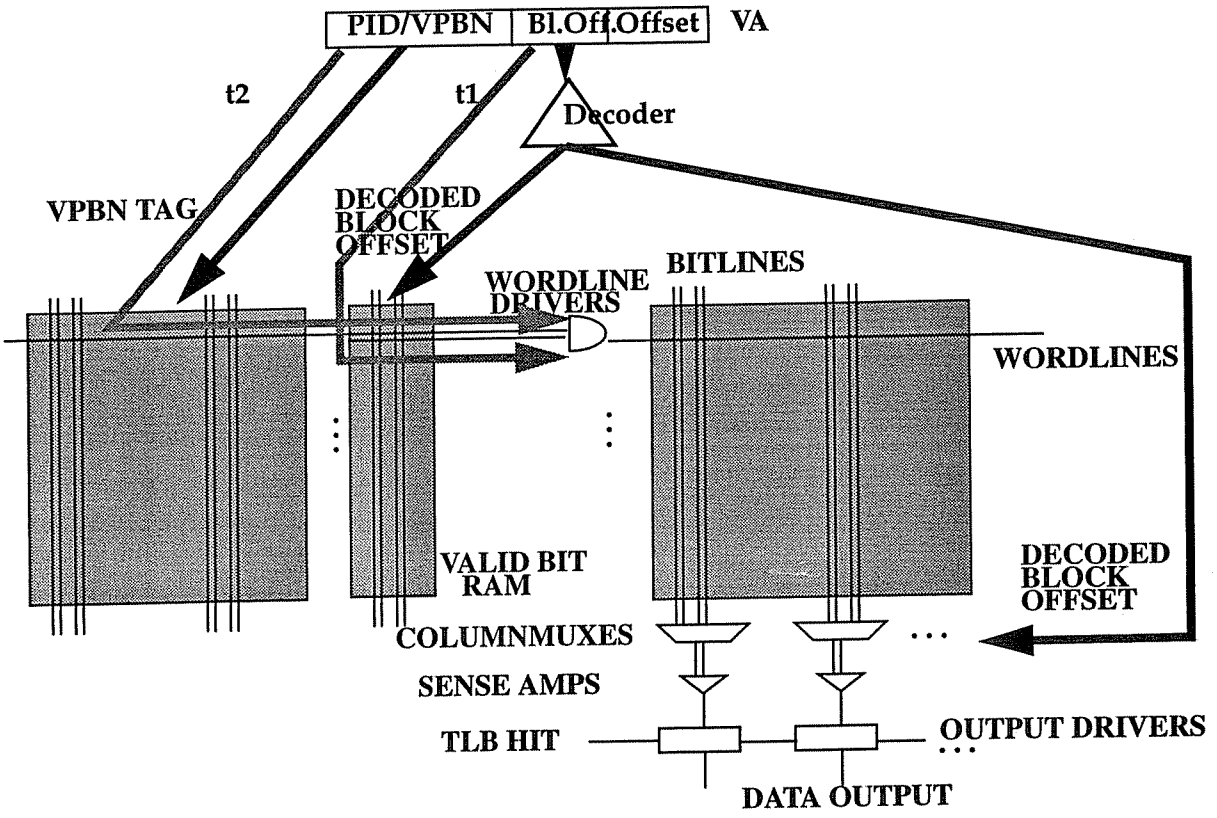
Figure B-6 illustrates the fully-associative implementation of the valid bit RAM approach. Parallel to the fully-associative tag lookup, a decoder decodes the block-offset field of the virtual address and selects the appropriate valid bit for each TLB block. The wordline driver then combines the valid bit with the match signal from the tag comparator (Appendix A). This implementation does not add to the critical path if the valid-bit RAM delay (t1) is less than the tag match delay (t2).

Figure 4-5 in Chapter 4 shows the set-associative implementation of the valid bit RAM approach with the valid bits implemented in the data RAM itself. The multiplexor driver combines the valid bit and the tag compare output. This option is very attractive for set-associative TLBs where the tag match delay is often in the critical path[3] and including the subblock valid bits in this fashion does not affect the access time. Instead, it reduces access time—the subblock TLB tags are shorter making both the tag RAM access and tag compare times shorter and reduces the critical path.

2. The block offset bits may be predecoded in the preceding CPU pipeline stage by combining a decoder with pipeline latches or logic as suggested to me by Robert Yung, Sun Microsystems Laboratories.

3. This is not true for caches where the data RAM is often on the critical path [Wilt93]. Subblock-cache typically store the subblock valid bits along with the tag.

Figure B-6: A fully-associative complete-subblock TLB with separate valid bit RAM



The separate valid bit RAM has two key advantages over the valid bit tag comparator. First, a valid bit RAM occupies less area than a fully-associative valid bit CAM implementation. Second, it results in a faster tag comparison as the tags are shorter without the valid bits.

In my thesis, access time and area models for set-associative and fully-associative complete-subblock TLBs assume the use of block-valid bits. Access time and area models for set-associative and fully-associative partial-subblock TLBs assume the valid bit tag comparator approach. I do not model the valid bit RAM approach in my area and access time models. I instead use the slower and more expensive valid bit tag comparator approach for partial-subblock TLBs, which make my results pessimistic.

In summary, there are three different ways subblock valid bits can be stored in a subblock TLB—using block-valid bits, valid bit tag comparators or in a valid bit RAM. In set-associative TLBs, storing the subblock valid bits as part of the data RAM is often the best option. In fully-associative complete-subblock TLBs, use of block valid bits and storing the subblock valid bits in the data RAM is the best option. In fully-associative partial-subblock TLBs, using the valid bit RAM is often the best option.

Appendix C: Implementation of subblock multiplexor

A subblock multiplexor selects the appropriate subblock mapping from a complete-subblock TLB block (Figure 4-3) read out of the data RAM. Decoded block-offset bits from the virtual address control this multiplexor. The decode occurs in parallel to the row decode and is not on the TLB access critical path. The multiplexor itself is often on the critical path. I discuss two alternate implementations in this section.

First, the multiplexor can be placed between the data RAM sense amps and the output driver. Often a multiplexor exists already here to support bypass-mode or large superpage physical address generation and the subblock multiplexor can be combined with it. However, this multiplexor is on the critical path and adding more inputs impacts access time.

RAM designs sometimes include a column multiplexor with the bitlines. Column multiplexors help divide the address decoding logic between the rows and columns instead of a single large row decode. Complete-subblock TLBs can use column multiplexors to select the appropriate subblock. Figure C-1 shows how this scheme works using a single column multiplexor to select one of four bitlines. A column multiplexor, while on the critical path for fully-associative TLBs, adds less overhead than extending the output multiplexor. Further, in a complete-subblock TLB, only a fraction of the bits read out of the data RAM are useful. In a TLB with subblock factor of 16, for example, only 6% of the bits are useful. The savings in sense amps from using column multiplexors is significant.

The key to using column multiplexors lies in the format used to store multiple mappings in one word of the data RAM. An efficient way to store the mappings is to use an interleaved format as illustrated in Figure C-2. All the valid bits are stored contiguously, bit_0 of all the PPN fields are stored together, and so on. The interleaved format allows all four valid bits to share a single sense amp. A non-interleaved format would require criss-crossing wires to use column multiplexors, which is impractical to implement.

The advantages of using column multiplexors are many. First, they add less delay to the bitlines and data RAM access time, than a full fledged multiplexor would. Second, with s times fewer sense amps, the area savings are significant. Alternatively, the sense amps can be made s times larger and larger sense amps are faster. Third, in set-associative TLBs the output multiplexor is on the critical path whereas the data RAM access is not and adding column multiplexors often does not affect overall TLB access time. Table C-3 illustrates the percentage reduction in TLB access time with the use of column multiplexors compared to using the output multiplexor. Set-associative TLBs show a dramatic reduction than fully-associative TLBs because the output multiplexor driver is in the critical path whereas the column multiplexor adds delay to a non-critical path. All measurements in my thesis assume the use of column multiplexors in complete-subblock-TLBs.

Figure C-1: Column Multiplexor use in complete-subblock TLBs

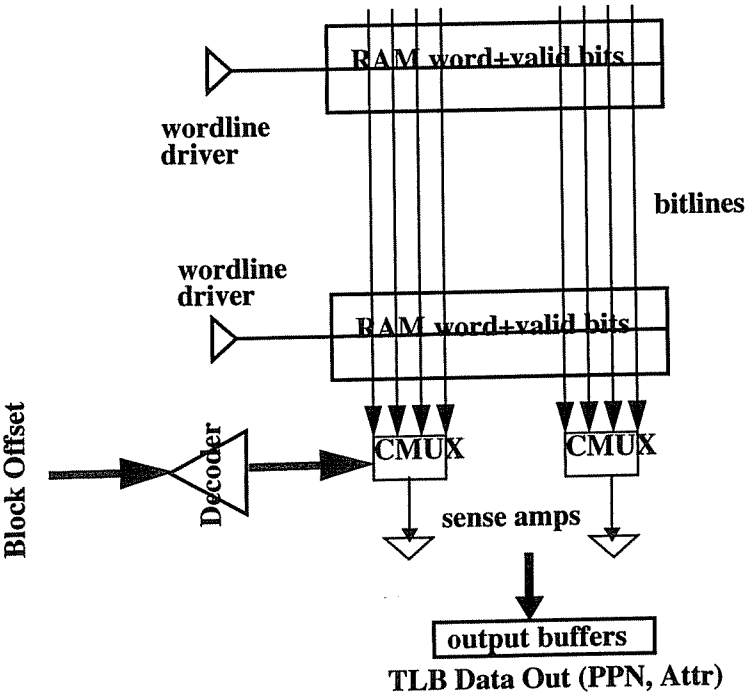


Figure C-2: Interleaved Layout of Data RAM

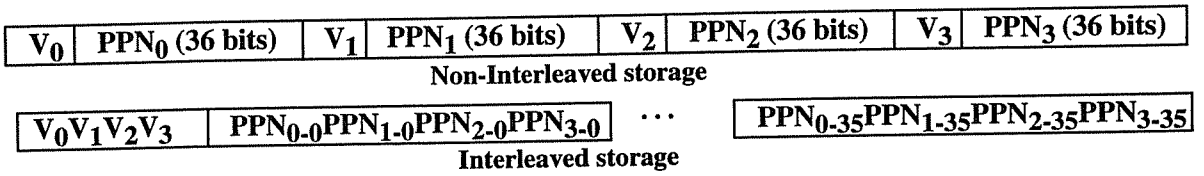


Table C-3: Reduction in access time for complete-subblock TLBs with column multiplexors

TLB	subblock factor			
	2	4	8	16
64-block fully-associative	0.1%	0.6%	1.5%	3.0%
128-block fully-associative	0.2%	0.7%	1.5%	3.0%
256-block fully-associative	0.2%	0.7%	1.5%	3.0%
512-block fully-associative	0.2%	0.6%	1.3%	2.7%
128-block 4-way set-associative	2.6%	8.0%	15.8%	26.5%
256-block 4-way set-associative	2.6%	7.9%	15.5%	26.2%

Appendix D: Preventing loading multiple copies in preloading

If a base page is not present in memory when mappings for a page block are preloaded into a subblock TLB, a subsequent reference to the base page causes a TLB miss. If the TLB miss handler blindly preloads all the mappings to the page block, it can load multiple copies of a mapping in the TLB. This causes electrical problems in most TLB implementations.

Figure D-1 illustrates this. State-1 shows the TLB block after a TLB miss to page 001, when page 002 was not present in physical memory. The TLB miss handler preloads all the mappings to pages in the page block except for page 002. Later when the program references page 002, a TLB miss occurs. Blindly preloading all mappings to the page block (page 002 is now in memory) will result in the TLB having two copies of the mappings for pages 000, 001 and 003 (State-2).

Figure D-1: Preloading on subblock miss

State 1 - After TLB miss on page 001

00	✓	PPN_0	$ATTR_0$
	✓	PPN_1	$ATTR_1$
	×		
	✓	PPN_3	$ATTR_3$

State 2 - After TLB miss on page 002

00	✓	PPN_0	$ATTR_0$
	✓	PPN_1	$ATTR_1$
	×		
	✓	PPN_3	$ATTR_3$
00	✓	PPN_0	$ATTR_0$
	✓	PPN_1	$ATTR_1$
	×	PPN_2	$ATTR_2$
	✓	PPN_3	$ATTR_3$

This problem can be addressed in two ways—by invalidating existing mappings before preloading (or when loading a superpage mapping) or using a separate non-preloading TLB miss handler for subblock misses. The invalidation option requires the TLB miss handler to issue a demap explicitly or hardware to implement an *implicit demap* during **load_TLB** operations[4]. The separate miss handler requires hardware to recognize that a subblock miss has occurred.

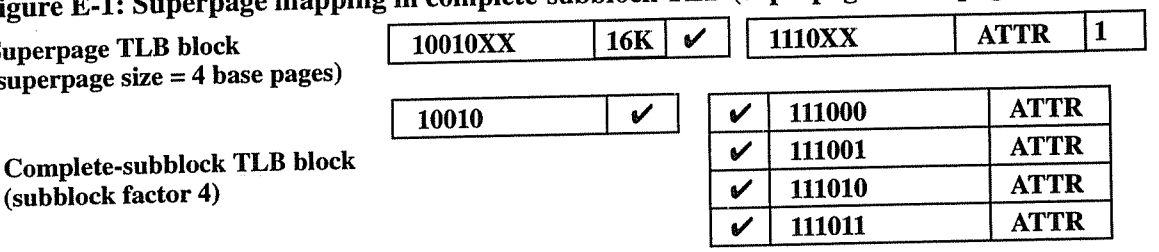
4. It is not sufficient for hardware to demap the TLB block when the TLB miss occurs, if the software can preload mappings that are not from the same page block [Bala94].

Appendix E: Storing superpage mappings in complete-subblock TLBs

Complete-subblock TLBs can store superpage mappings too. A superpage mapping is more efficient for preloading mappings for a page block than a complete-subblock TLB miss handler. Complete-subblock TLB blocks continue to support efficiently the cases were superpages cannot be used. There are three categories of superpage mappings that need to be considered—where the superpage size is equal to, less than or greater than the page block size of the complete-subblock TLB.

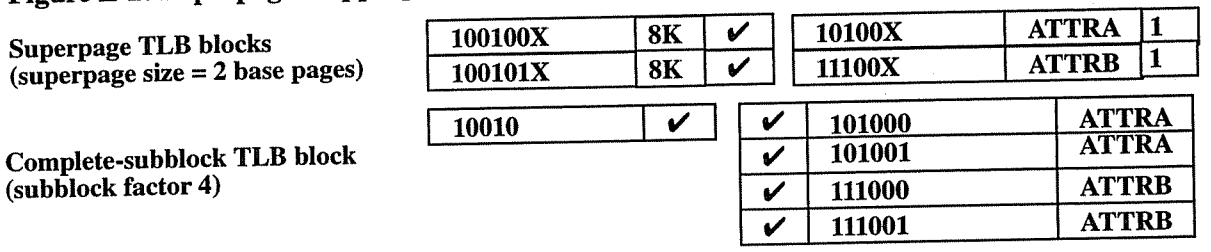
A complete-subblock TLB block can store superpage mappings with the superpage size equal to the page block size by replicating the superpage mapping in all subblocks and adjusting the PPN fields—the low-order bits set equal to the virtual block offset (Figure E-1).

Figure E-1: Superpage mapping in complete-subblock TLB (superpage size = page block size)



A complete-subblock TLB block similarly stores superpage mappings where the superpage size is smaller than the page block size. The PPN bits are adjusted with the virtual block offset field using the superpage size. Figure E-2 shows how a complete-subblock TLB block with subblock factor of 4 stores two 8KB superpage mappings. Both fully-associative and set-associative complete-subblock TLBs can store superpage mappings of any size less than or equal to the page block size. This is important because set-associative superpage TLBs supporting multiple superpage sizes are not practical.

Figure E-2: Superpage mapping in complete-subblock TLB (superpage size < page block size)

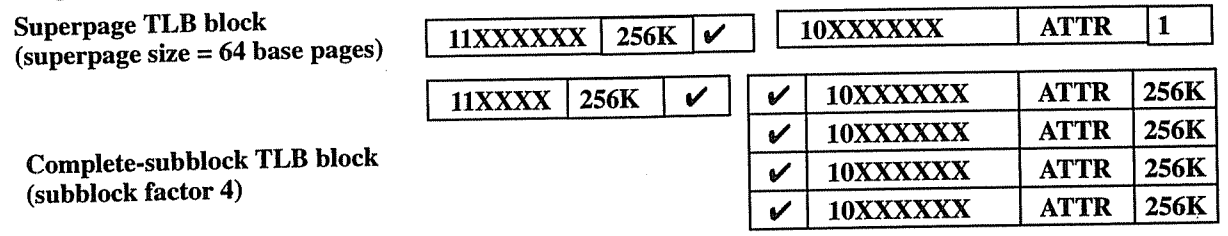


Finally, support for large superpage mappings can be included in fully-associative[5] complete-subblock TLBs. Such large superpage mappings are useful for many applications. Two modifications can be borrowed from superpage TLBs. First, the VPBN in the tag is made up of don't-care bits storing the MASK field as in superpage TLBs. The low-order x tag bits are implemented as don't-care bits to support superpage sizes upto 2^x * page block size. Second, it adds a page size attribute that controls a multiplexor used in physical address generation to select the low order bits of PPBN from either the virtual address or the PPN read from the TLB. A superpage mapping can be stored in a complete-subblock TLB block by duplicating it in all the subblocks such that all virtual address within the superpage, irrespective of the

5. Set-associative complete-subblock TLBs cannot load large superpage mappings due to the difficulty in choosing the set index for these mappings—same problem faced by superpage TLBs.

block offset, read the same mapping (Figure E-3).

Figure E-3: Superpage mapping in complete-subblock TLB (superpage size > page block size)



The key support needed in all three cases is to modify the TLB miss handler to be able to traverse a page table that includes superpage mappings and transforming the mapping to fit in the complete-subblock TLB block. The transformation can be done either in hardware or software, but software transformations are inefficient. The complete-subblock TLB hardware implementation does not have to be modified to support superpage mappings for superpage sizes ≤ page block size.

Even in a complete-subblock TLB system, there are benefits to using superpage mappings in the page table. First, the page table may use less storage to store superpage mappings than storing separate mappings required for the single-page-size and complete-subblock TLBs. Second, loading superpage mappings is an efficient way of preloading in the TLB miss handler—only a single PTE for the page block needs to be fetched from the page table. The TLB miss handler, however, must now traverse a page table that supports superpages and expand the superpage mappings for the complete-subblock TLB.

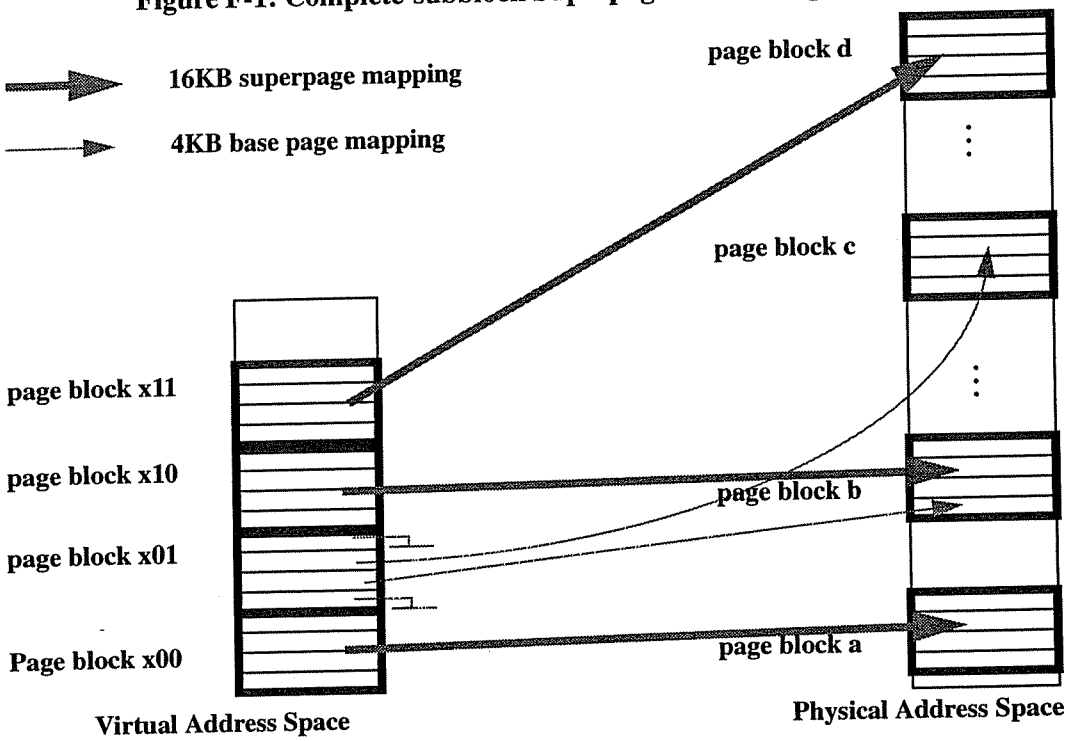
Another alternative for supporting superpage mappings is to use a separate fully-associative TLB to store superpage mappings. The PowerPC, for example, has two TLBs—one for superpages and another for base pages. The advantage of this approach is that it allows the two TLBs to be independently optimized, *e.g.*, the base page TLB can be set-associative. It, however, adds a multiplexor to the TLB critical path and can suffer from load imbalance if the ratio of base page to superpage mappings does not correspond to the hardware resources.

In summary, complete-subblock TLBs easily support mappings for multiple medium-sized superpages less than or equal to the page block size. Large superpage support can be added to complete-subblock TLBs for a similar complexity of adding superpage support to single-page-size TLBs. A complete-subblock TLB is a waste of hardware resources if the operating system supports and frequently uses superpages. However, in systems where the operating system uses only few superpage mappings—either large or small—a complete-subblock TLB can easily support them.

Appendix F: Complete-subblocking for superpage TLBs

This appendix explores the option of building a complete-subblock superpage TLB—where each TLB block has the same subblock factor but varies the page block size. The MIPS R4000 processor, for example, has a complete-subblock TLB with subblock factor two that supports seven page sizes also. Figure F-1 shows how a 64KB region of virtual address space may be mapped using 16KB superpages and 4KB base pages. A regular superpage TLB would use 5 TLB blocks to store the mappings, but a complete-subblock superpage TLB requires only *two* TLB blocks. Just as Chapter 4 shows that complete-subblocking and preloading is very effective for base page mappings, similar arguments can be made for superpage mappings also. However, this requires the operating system to allocate superpage mappings for neighboring virtual page blocks to make the complete-subblock superpage TLB effective.

Figure F-1: Complete-subblock Superpage TLB example



xXX	16K	✔
x01	4K	✔

✔	aXX	attr	×			✔	bXX	attr	✔	dXX	attr
×			✔	b00	attr	✔	c02	attr	×		

Complete-subblock TLB storing mappings for above

Implementing such a TLB, however, is not straightforward. A single-page-size TLB easily extends to include complete subblocking by adding logic and subblock multiplexors to select the appropriate subblock from the data RAM—the block offset bits control the subblock multiplexor. To modify a superpage TLB, however, the block-offset bits are unknown when starting the lookup. They depend on the page size and is different for each TLB block. In the above example, for page block x00 bits 15 and 14 form the block offset field, and for page block x01 bits 13 and 12 form the block offset field. Thus, the matching TLB block must be known to determine the page-size and block-offset bits uniquely.

Figure F-2: A fully-associative complete-subblock superpage TLB

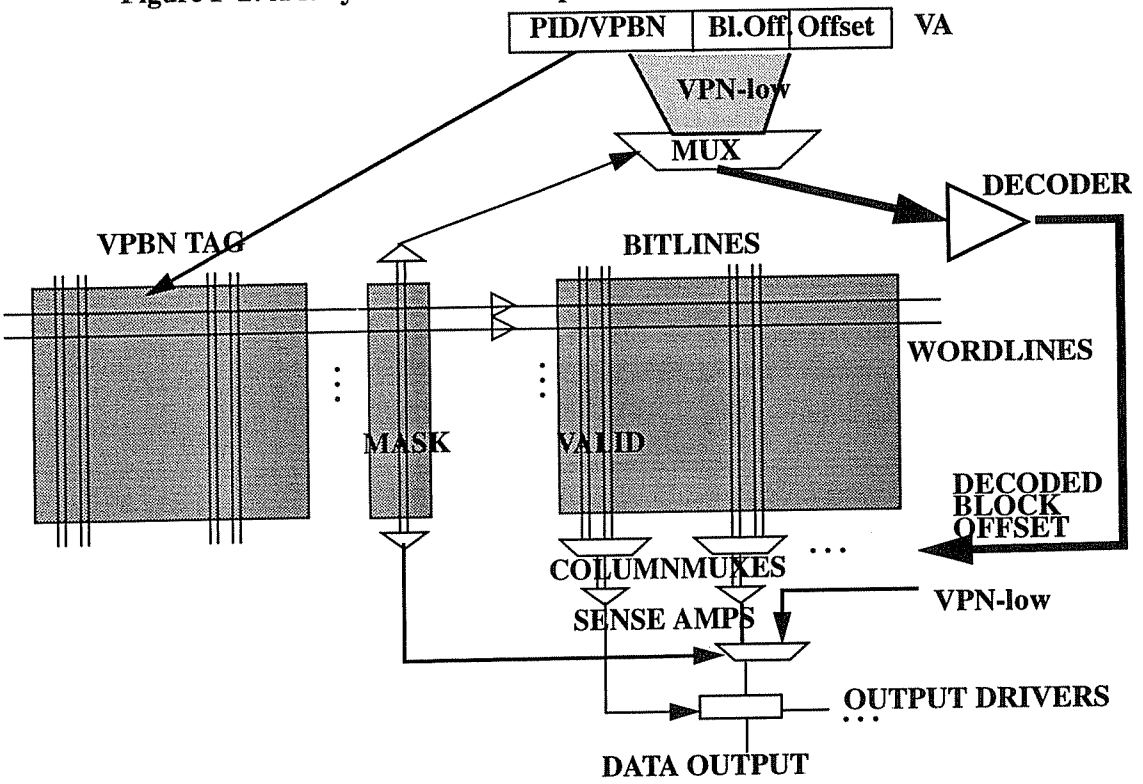


Figure F-2 illustrates the basic operation of one possible implementation of a fully-associative complete-subblock superpage TLB. The CAM array uses the VPBN as the compare input. The MASK field (shown as a separate block for clarity) works as a mask for the tag comparators as in a superpage TLB. The CAM array also outputs the MASK field of the matching TLB block to control two multiplexors. One multiplexor controls physical address generation as in a superpage TLB. Another multiplexor selects bits from the virtual address that form the Block Offset, which after decoding controls the subblock multiplexors. This solution requires that the subblock valid bits be in the data and not part of the tag compare.

This solution is expensive because it serializes decoding of the block-offset and tag comparison. Though the multiplexor controls can be set up in parallel to the wordline drivers and the data RAM access, it may add to the critical path. Further, a standard CAM implementation does not allow bits to be read during a comparison—this implementation requires it. This solution does not work if more than one TLB block can have the same tag and requires the subblock valid bit to determine a hit—as is true in the valid bit RAM and valid bit tag comparator approaches (Appendix B).

Further, it is not practical to implement a partial-subblock TLB that supports subblocking for multiple superpage sizes, *i.e.*, varying the page block size for each TLB block in a partial-subblock TLB[6]. The solution described above for complete-subblock TLBs uses a block valid bit and is not practical for partial-subblock TLBs, which must support multiple identical tags. Partial-subblock TLBs require the block-offset bits to select a subblock valid bit and complete the tag match but the block-offset bits cannot be uniquely determined unless the tag match completes—a circular dependency!

6. Supporting large superpages in partial-subblock TLB varies the page block size for each TLB block but it requires ALL subblock valid bits to be set.

Appendix G: Subblock miss checking in partial-subblock TLBs

Subblock miss checking determines if a mapping, to be loaded into the TLB, can be stored in an existing TLB block. In a partial-subblock TLB, it involves checking all the TLB blocks to see if the new mapping has the same tag, is properly placed with respect to other mappings in that TLB block and has the same attributes. Unaligned mappings always fail subblock miss checking. If any of the TLB blocks succeeds the checks, the new mapping can be loaded into the TLB by simply setting the appropriate valid bit. The pseudo-code for subblock miss checking is as follows:

```
if (unaligned(mapping)) return(FAIL);
for i = 1 to n {          /* n is number of blocks in the TLB set */
    if ((Block[i].tag == mapping.VPBN) && (Block[i].SB == mapping.SB) &&
       (Block[i].Attr == mapping.Attr) && (Block[i].PPBN == mapping.PPBN))
            return (OK);
} return(FAIL);
```

Implementing subblock miss checking is neither easy nor efficient. There are at least four ways to implement subblock miss checking—fully-associative hardware lookup, software, first-tag-hit, or first-tag-hit combined with software.

A fully-associative lookup in hardware uses a CAM for the data part of the TLB also. During TLB lookup the data part functions as a RAM. During subblock miss checking both tag and data CAMs participate in the comparison—the valid bits do not. If any of the TLB blocks match, it sets the corresponding valid bit. This is similar to implementing a writable CAM with the VPBN and data fields as the key and the valid bit as the data to be written. This solution has the advantage that subblock miss checking and loading the new mapping can be completed in a single operation. The disadvantage is the significant hardware cost of implementing a fully-associative data field.

The second alternative executes subblock miss checking in software. The TLB miss handler would read the TLB blocks from either the TLB or a software copy and compare them with the new mapping. However, the TLB miss penalty increases significantly—proportional to the number of TLB blocks.

The third alternative, *first-tag-hit*, checks a single TLB block instead of all possible TLB blocks. Tag comparison logic compares the tags (VPBN), without valid bits, of the TLB blocks with the VPBN of the new mapping. If a single tag matches, which is the common case, then that TLB block is a candidate for subblock miss checking. If more than one tag matches, it chooses one of the blocks as a candidate for subblock miss checking, *e.g.*, the TLB block in the lowest numbered slot. The data field of the candidate TLB block is read and compared with the new mapping to determine a subblock miss. The advantage of this solution is that the data part of the TLB can use a RAM. The main disadvantage of this approach is that it may result in more TLB replacements than in a solution that checks all the TLB blocks. TLB simulations show that using first-tag-hit subblock miss checking does not result in significantly higher number of TLB misses (often less than a 1% increase).

The fourth alternative combines first-tag-hit hardware checking with software miss checking in the uncommon case. In the first-tag-hit check, when there are multiple TLB blocks with

matching tags, one is chosen to complete the subblock miss check. If that fails, a software TLB miss handler can check the other matching TLB blocks. However, instead of scanning all the TLB entries, software need compare fewer blocks, *e.g.*, TLB blocks numbered higher than the block with the first matching tag. The advantage of this approach is that hardware handles the common case fast with simple hardware, leaving the complicated uncommon case to software.

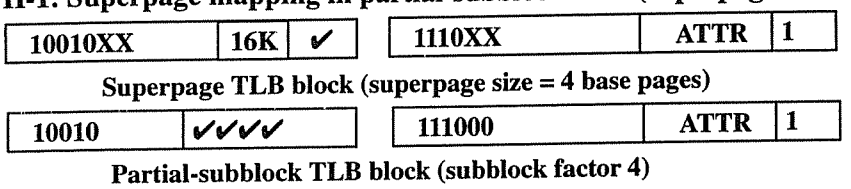
In summary, subblock miss checking is complicated in a partial-subblock TLB. A hardware-only or software-only solution results in complicated hardware or slow TLB miss handling. The first-tag-hit approach implements imperfect checking using simpler hardware and with performance comparable to perfect hardware. Software can augment this approach to implement perfect checking but the cost of a higher TLB miss penalty.

Appendix H: Storing superpage mappings in partial-subblock TLBs

Partial-subblock TLBs can store superpage mappings in a manner similar to how complete-subblock TLBs store superpage mappings (Appendix E). There are three categories of superpage mappings that need to be considered—where the superpage size is equal to, less than or greater than the page block size.

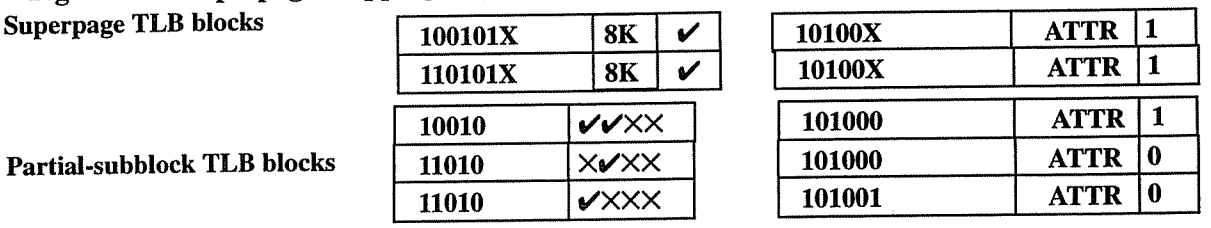
A partial-subblock TLB block stores a superpage mapping with the superpage size equal to the page block size by setting all the valid bits and SB attribute and copying the PPN and Attr fields from the superpage mapping (Figure H-1). Both fully-associative and set-associative partial-subblock TLBs can store such superpage mappings.

Figure H-1: Superpage mapping in partial-subblock TLB (superpage size = page block size)



A single partial-subblock TLB block cannot always store a superpage mapping with a superpage size smaller than the page block size. Superpages use physical base pages properly placed with respect to the superpage size. However, the same physical base pages are unlikely to be properly placed with respect to a larger page block size also. Figure H-2 shows a partial-subblock TLB with subblock factor 4 storing two superpage mappings (8KB superpage size). The first superpage mapping has properly placed physical pages with respect to a page block size of 16K while the second one does not.

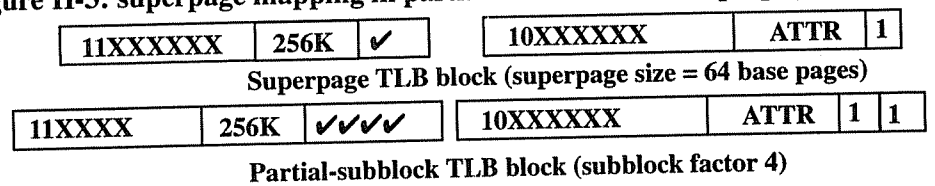
Figure H-2: Superpage mapping in partial-subblock TLB (superpage size < page block size)



Finally, support for large superpage mappings can be included in fully-associative[7] partial-subblock TLBs. Two modifications can be borrowed from superpage TLBs into partial-subblock TLB implementations. First, the VPBN in the tag is made up of don't-care bits storing the MASK field, as in superpage TLBs. The low-order x tag bits are implemented as don't-care bits to support superpage sizes upto 2^x * page block size. Second, it adds a page size attribute that controls a multiplexor used in physical address generation to select the low order bits of PPBN from either the virtual address or the PPN read from the TLB. Note that the multiplexor used to select the block-offset bits based on the SB bit remains unchanged. Now the superpage mapping can be stored in the partial-subblock TLB by copying the VPBN, pagesize, PPN and Attr fields from the superpage mapping, setting the SB bit and all the subblock valid bits (Figure H-3).

7. Set-associative partial-subblock TLBs cannot store large superpage mappings due to the difficulty in choosing the set index for these mappings—as in superpage TLBs (Section 3.2.2).

Figure H-3: superpage mapping in partial-subblock TLB (superpage size > page block size)



If a page table stores partial-subblock PTEs, then support for medium-size superpages is unnecessary—a partial-subblock page table is a superset of a superpage page table (with correct physical memory allocation). Preloading in the partial-subblock TLB is a superior way to loading superpage mappings. Table H-4 shows the percent reduction in the number of TLB misses for a 64-block fully-associative partial-subblock TLB through use of two different enhancements to the TLB miss handler. The first uses preloading. The second does not use preloading but uses superpage mappings whereever possible. Preloading is more effective at reducing TLB misses than superpages[8]. The TLB miss penalty and page table memory requirements for preloading and superpages are roughly comparable. Thus, it is preferable to use preloading and storing partial-subblock PTEs in the page table instead of supporting medium-sized superpages. This is important since many microprocessors support superpages and operating systems are being modified to support superpage mappings in the page tables.

Table H-4: Reduction in number of TLB misses in a 64-block fully-associative partial-subblock TLB with preloading or with superpage mappings

Workload	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
	preloading	superpages	preloading	superpages	preloading	superpages	preloading	superpages
coral	16.61%	16.60%	30.38%	30.10%	37.57%	36.77%	49.15%	47.47%
nasa7	42.33%	42.30%	69.96%	69.84%	81.81%	25.16%	69.98%	54.77%
compress	41.42%	37.96%	37.27%	35.44%	27.90%	16.91%	26.33%	9.45%
fftpde	0.35%	0.34%	0.53%	0.51%	0.64%	0.62%	49.24%	49.19%
wave5	43.44%	43.45%	65.52%	65.15%	76.02%	74.68%	79.42%	76.66%
mp3d	23.69%	22.30%	59.56%	59.52%	66.38%	65.62%	88.84%	86.50%
spice	30.78%	30.78%	64.42%	64.19%	83.90%	81.95%	87.07%	69.66%
pthor	25.16%	25.15%	47.30%	46.82%	64.54%	62.69%	76.05%	74.77%
ML	24.17%	23.78%	46.40%	42.82%	62.43%	60.91%	76.74%	73.18%
gcc	27.12%	27.20%	54.54%	50.92%	51.14%	37.44%	42.30%	15.96%

In summary, partial-subblock TLBs with preloading are slightly better than medium-size superpage mappings. Fully-associative partial-subblock TLBs can easily include support for larger superpage sizes.

8. Foxtrot did NOT do page promotion in this experiment—the page table only considers fully-populated page blocks for superpages. If the operating system used page promotions, superpages would be used more often.

Appendix I: Detailed Speedup Tables

This appendix includes execution time speedups for all the workloads. It includes data for each of the tables in the main text where only the weighted average across all the workloads was presented. The behavior of individual workloads is shown here. Tables in this appendix are numbered to show their correspondence to tables in the body of the thesis. Tables I2-3a through I2-3c, for example, correspond to Table 2-3 in Chapter 2.

Table I2-3a: Sensitivity to TLB miss penalty—execution time speedup for alternate fully-associative TLBs relative to 64-block fully-associative single-page-size (4KB) TLB (TLB miss penalty = 30 cycles)

Workload	128-block single-page-size (4KB) TLB	123-block superpage (4KB/32KB) TLB	114-block partial-subblock TLB (subblock factor 16)	72-block complete-subblock TLB (subblock factor 4)
coral	1.105	1.368	1.461	1.111
nasa7	1.008	1.419	1.419	1.115
compress	1.184	1.193	1.193	1.193
fftpde	1.000	1.001	1.182	1.000
wave5	1.042	1.110	1.110	1.109
mp3d	1.035	1.089	1.090	1.085
spice	1.044	1.053	1.053	1.052
pthor	1.006	1.034	1.039	1.006
ML	1.015	1.030	1.031	1.017
gcc	1.013	1.016	1.016	1.015
Wt. Avg.	1.045	1.132	1.161	1.072

Table I2-3b: Sensitivity to TLB miss penalty—execution time speedup for alternate fully-associative TLBs relative to 64-block fully-associative single-page-size (4KB) TLB (TLB miss penalty = 40 cycles)

Workload	128-block single-page-size (4KB) TLB	123-block superpage (4KB/32KB) TLB	114-block partial-subblock TLB (subblock factor 16)	72-block complete-subblock TLB (subblock factor 4)
coral	1.145	1.559	1.726	1.153
nasa7	1.011	1.649	1.649	1.160
compress	1.262	1.275	1.275	1.274
fftpde	1.000	1.002	1.258	1.000
wave5	1.056	1.152	1.152	1.151
mp3d	1.047	1.123	1.123	1.117
spice	1.059	1.073	1.073	1.071
pthor	1.008	1.045	1.052	1.008
ML	1.021	1.040	1.042	1.022
gcc	1.017	1.021	1.021	1.020
Wt. Avg.	1.061	1.185	1.227	1.098

Table I2-3c: Sensitivity to TLB miss penalty—execution time speedup for alternate fully-associative TLBs relative to 64-block fully-associative single-page-size (4KB) TLB (TLB miss penalty = 50 cycles)

Workload	128-block single-page-size (4KB) TLB	123-block superpage (4KB/32KB) TLB	114-block partial-subblock TLB (subblock factor 16)	72-block complete-subblock TLB (subblock factor 4)
coral	1.188	1.812	2.110	1.200
nasa7	1.013	1.969	1.969	1.208
compress	1.350	1.368	1.368	1.368
fftpde	1.000	1.002	1.344	1.000
wave5	1.071	1.197	1.197	1.197
mp3d	1.059	1.158	1.158	1.150
spice	1.075	1.092	1.092	1.090
pthor	1.010	1.057	1.066	1.011
ML	1.026	1.051	1.053	1.028
gcc	1.022	1.026	1.026	1.026
Wt. Avg.	1.078	1.242	1.301	1.125

Table I2-5: Sensitivity to TLB replacement policy—execution time speedups relative to 64-block fully-associative single-page-size (4KB) TLB using Go-down-stack (GODS) replacement policy

Workload	Clock	Random	FIFO
coral	0.993	0.957	0.941
nasa7	1.001	0.987	0.984
compress	0.992	0.950	0.955
fftpde	1.000	1.022	0.987
wave5	0.999	1.024	1.005
mp3d	1.000	0.977	0.979
spice	0.991	0.949	0.963
pthor	0.999	0.992	0.993
ML	0.998	0.985	0.989
gcc	0.998	0.986	0.988
Wt. Avg.	0.997	0.981	0.975

Table I3-1a: Execution time speedups for fully-associative superpage TLBs relative to single-page-size (4KB) TLBs with same number of blocks (64- and 128-blocks)

Workload	64-block (superpage size)				128-block (superpage size)			
	8KB	16KB	32KB	64KB	8KB	16KB	32KB	64KB
coral	1.150	1.295	1.430	1.564	1.171	1.254	1.369	1.533
nasa7	1.207	1.441	1.645	1.649	1.266	1.632	1.632	1.632
compress	1.263	1.274	1.274	1.271	1.010	1.010	1.010	1.010
fftpde	1.001	1.001	1.001	1.117	1.001	1.001	1.001	1.258
wave5	1.087	1.152	1.152	1.152	1.090	1.090	1.090	1.090
mp3d	1.035	1.100	1.122	1.123	1.066	1.072	1.073	1.073
spice	1.053	1.071	1.072	1.072	1.012	1.013	1.013	1.013
pthor	1.016	1.029	1.039	1.044	1.015	1.027	1.038	1.044
ML	1.017	1.028	1.035	1.039	1.010	1.016	1.019	1.021
gcc	1.015	1.020	1.021	1.021	1.003	1.004	1.004	1.004
Wt. Avg.	1.090	1.148	1.183	1.212	1.072	1.114	1.127	1.172

Table I3-1b: Execution time speedups for fully-associative superpage TLBs relative to single-page-size (4KB) TLBs with same number of blocks (256-blocks)

Workload	256-block (superpage size)			
	8KB	16KB	32KB	64KB
coral	1.065	1.134	1.259	1.384
nasa7	1.366	1.366	1.366	1.366
compress	1.000	1.000	1.000	1.000
fftpde	1.001	1.002	1.002	1.002
wave5	1.000	1.000	1.000	1.000
mp3d	1.004	1.005	1.005	1.005
spice	1.001	1.001	1.001	1.001
pthor	1.016	1.030	1.040	1.041
ML	1.007	1.011	1.012	1.013
gcc	1.000	1.000	1.000	1.000
Wt. Avg.	1.047	1.057	1.071	1.082

Table I3-2a: Execution time speedups for 256-block 4-way set-associative superpage TLBs relative to single-page-size (4KB) TLBs (superpage index)

Workload	with OS support				base pages only			
	8KB	16KB	32KB	64KB	8KB	16KB	32KB	64KB
coral	1.069	1.147	1.274	1.358	0.975	0.904	0.699	0.687
nasa7	1.498	1.503	1.498	1.478	1.159	1.181	0.879	0.850
compress	1.000	1.000	0.981	0.812	0.999	0.992	0.682	0.540
fftpde	1.000	1.192	1.214	1.434	1.000	1.190	1.210	1.431
wave5	1.000	1.000	1.000	0.996	0.995	0.959	0.744	0.672
mp3d	1.009	1.010	1.010	1.010	0.998	0.996	0.934	0.896
spice	1.003	1.003	1.003	1.003	0.992	0.975	0.876	0.794
pthor	1.016	1.022	1.029	0.863	0.999	0.988	0.982	0.823
ML	1.008	1.012	1.010	1.007	0.998	0.992	0.972	0.953
gcc	1.001	1.000	0.934	0.853	0.999	0.993	0.947	0.818
Wt. Avg.	1.058	1.092	1.098	1.070	1.013	1.018	0.871	0.809

Table I3-2b: Execution time speedups for 256-block 4-way set-associative superpage TLBs relative to single-page-size (4KB) TLBs (exact index)

Workload	with OS support			
	8KB	16KB	32KB	64KB
coral	1.069	1.145	1.279	1.408
nasa7	1.497	1.503	1.503	1.503
compress	1.000	1.000	1.000	1.000
fftpde	1.011	1.214	1.243	1.442
wave5	1.000	1.000	1.000	1.000
mp3d	1.009	1.010	1.010	1.010
spice	1.003	1.003	1.003	1.003
pthor	1.016	1.031	1.038	1.040
ML	1.008	1.012	1.014	1.014
gcc	1.001	1.001	1.001	1.001
Wt. Avg.	1.060	1.095	1.113	1.143

Table I3-3a: Execution time speedups for superpage TLBs relative to set-associative single-page-size (4KB) TLBs of comparable chip area (256-block 4-way set-associative)

Workload	Single Page Size TLB	4KB/32KB Superpage TLB				4KB/64KB Superpage TLB			
	162-block fully-associative	156-block fully-associative		256-block set-associative		154-block fully-associative		256-block set-associative	
		with OS	no OS	with OS	no OS	with OS	no OS	with OS	no OS
coral	0.929	1.185	0.923	1.274	0.699	1.330	0.921	1.358	0.687
nasa7	0.978	1.503	0.966	1.498	0.879	1.503	0.961	1.478	0.850
compress	0.997	1.000	0.996	0.981	0.682	1.000	0.996	0.812	0.540
fftpde	1.439	1.442	1.439	1.214	1.210	1.442	1.439	1.434	1.431
wave5	0.990	1.000	0.990	1.000	0.744	1.000	0.990	0.996	0.672
mp3d	0.963	1.010	0.960	1.010	0.934	1.010	0.958	1.010	0.896
spice	0.996	1.003	0.996	1.003	0.876	1.003	0.995	1.003	0.794
pthor	0.994	1.031	0.994	1.029	0.982	1.039	0.993	0.863	0.823
ML	0.997	1.013	0.996	1.010	0.972	1.014	0.996	1.007	0.953
gcc	0.998	1.001	0.998	0.934	0.947	1.001	0.998	0.853	0.818
Wt. Avg.	1.020	1.120	1.017	1.098	0.871	1.136	1.016	1.070	0.809

Table I3-3b: Execution time speedups for superpage TLBs relative to set-associative single-page-size (4KB) TLBs of comparable chip area (512-block 4-way set-associative)

Workload	Single Page Size TLB	4KB/32KB Superpage TLB				4KB/64KB Superpage TLB			
	304-block fully-associative	293-block fully-associative		512-block set-associative		290-block fully-associative		512-block set-associative	
		with OS	no OS	with OS	no OS	with OS	no OS	with OS	no OS
coral	0.932	1.167	0.927	1.248	0.663	1.264	0.925	1.223	0.652
nasa7	0.755	1.003	0.748	1.000	0.593	1.003	0.753	0.986	0.567
compress	1.000	1.000	1.000	0.981	0.682	1.000	1.000	0.812	0.541
fftpde	1.435	1.438	1.435	1.437	1.431	1.438	1.435	1.435	1.431
wave5	1.000	1.000	1.000	1.000	0.751	1.000	1.000	0.996	0.674
mp3d	0.999	1.002	0.999	1.002	0.928	1.002	0.999	1.002	0.898
spice	0.999	1.000	0.999	1.000	0.876	1.000	0.999	1.000	0.792
pthor	0.984	1.020	0.983	1.014	0.976	1.020	0.983	0.848	0.811
ML	0.997	1.008	0.997	1.006	0.974	1.009	0.997	1.000	0.953
gcc	1.000	1.000	1.000	0.934	0.886	1.000	1.000	0.852	0.774
Wt. Avg.	1.000	1.065	0.998	1.061	0.836	1.074	0.998	1.006	0.765

Table I4-1a: Execution time speedups for complete-subblock TLBs relative to single-page-size (4KB) TLBs with same number of blocks (64-block fully-associative)

Workload	(NO preloading) subblock factor				(with preloading) subblock factor			
	2	4	8	16	2	4	8	16
coral	1.064	1.134	1.245	1.334	1.151	1.299	1.439	1.578
nasa7	1.009	1.115	1.648	1.649	1.207	1.442	1.649	1.649
compress	1.257	1.274	1.274	1.274	1.265	1.275	1.275	1.275
fftpde	1.000	1.000	1.000	1.003	1.001	1.001	1.002	1.114
wave5	1.043	1.151	1.151	1.152	1.088	1.152	1.152	1.152
mp3d	1.015	1.083	1.121	1.122	1.038	1.107	1.122	1.123
spice	1.047	1.068	1.072	1.073	1.054	1.071	1.072	1.073
pthor	1.003	1.007	1.011	1.018	1.017	1.031	1.041	1.048
ML	1.011	1.019	1.027	1.034	1.018	1.031	1.038	1.041
gcc	1.014	1.020	1.021	1.021	1.016	1.021	1.021	1.021
Wt. Avg.	1.044	1.089	1.157	1.170	1.091	1.150	1.185	1.214

Table I4-1b: Execution time speedups for complete-subblock TLBs relative to single-page-size (4KB) TLBs with same number of blocks (128-block fully-associative)

Workload	(NO preloading) subblock factor				(with preloading) subblock factor			
	2	4	8	16	2	4	8	16
coral	1.135	1.174	1.224	1.335	1.172	1.257	1.377	1.547
nasa7	1.100	1.631	1.632	1.632	1.267	1.632	1.632	1.632
compress	1.010	1.010	1.010	1.010	1.010	1.010	1.010	1.010
fftpde	1.000	1.000	1.000	1.257	1.001	1.001	1.002	1.258
wave5	1.090	1.090	1.090	1.090	1.090	1.090	1.090	1.090
mp3d	1.064	1.071	1.072	1.073	1.067	1.072	1.073	1.073
spice	1.011	1.012	1.013	1.013	1.012	1.013	1.013	1.013
pthor	1.003	1.008	1.020	1.039	1.015	1.028	1.039	1.047
ML	1.005	1.010	1.015	1.019	1.011	1.017	1.020	1.021
gcc	1.003	1.004	1.004	1.004	1.003	1.004	1.004	1.004
Wt. Avg.	1.048	1.101	1.109	1.152	1.073	1.114	1.129	1.173

Table I4-1c: Execution time speedups for complete-subblock TLBs relative to single-page-size (4KB) TLBs with same number of blocks (256-block 4-way set-associative)

Workload	(NO preloading) subblock factor				(with preloading) subblock factor			
	2	4	8	16	2	4	8	16
coral	1.036	1.078	1.170	1.385	1.070	1.150	1.287	1.418
nasa7	1.494	1.503	1.503	1.503	1.499	1.503	1.503	1.503
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	1.000	1.191	1.214	1.443	1.000	1.192	1.215	1.443
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.001
mp3d	1.008	1.009	1.010	1.010	1.009	1.010	1.010	1.010
spice	1.002	1.003	1.003	1.003	1.003	1.003	1.003	1.003
pthor	1.008	1.024	1.035	1.040	1.017	1.032	1.039	1.040
ML	1.004	1.009	1.013	1.014	1.008	1.013	1.014	1.015
gcc	1.001	1.001	1.001	1.001	1.001	1.001	1.001	1.001
Wt. Avg.	1.052	1.084	1.099	1.141	1.058	1.094	1.111	1.143

Table I4-3a: Effect of preloading in complete-subblock TLBs (64-block fully-associative)

Workload	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty
coral	1.082	1.20	1.145	1.43	1.156	1.61	1.183	1.96
nasa7	1.197	1.72	1.294	3.33	1.001	6.67	1.000	1.82
compress	1.006	1.75	1.000	1.79	1.000	1.75	1.000	1.89
fftpde	1.001	1.00	1.001	1.01	1.002	1.01	1.111	1.96
wave5	1.043	1.79	1.000	2.94	1.000	4.17	1.000	5.00
mp3d	1.023	1.30	1.022	2.63	1.002	3.23	1.001	10.00
spice	1.008	1.45	1.003	2.86	1.001	6.67	1.000	1.85
pthor	1.013	1.35	1.024	1.96	1.029	2.86	1.029	4.55
ML	1.007	1.32	1.011	2.08	1.011	3.45	1.007	6.67
gcc	1.002	1.43	1.001	2.27	1.000	1.69	1.000	1.75
Wt. Avg.	1.045	1.333	1.056	1.649	1.024	1.409	1.038	2.047

Table I4-3b: Effect of preloading in complete-subblock TLBs (128-block fully-associative)

Workload	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty
coral	1.033	1.11	1.071	1.28	1.125	1.67	1.159	2.78
nasa7	1.152	1.67	1.000	3.23	1.000	1.43	1.000	1.56
compress	1.000	1.43	1.000	1.61	1.000	1.72	1.000	1.79
fftpde	1.001	1.00	1.001	1.01	1.002	1.01	1.001	7.69
wave5	1.000	1.79	1.000	2.86	1.000	3.85	1.000	4.55
mp3d	1.003	1.54	1.001	2.56	1.001	5.56	1.000	1.08
spice	1.001	1.89	1.000	3.57	1.000	1.47	1.000	1.56
pthor	1.012	1.39	1.020	2.04	1.019	3.03	1.007	4.55
ML	1.005	1.52	1.006	2.44	1.005	5.00	1.002	10.00
gcc	1.000	1.49	1.000	1.52	1.000	1.67	1.000	1.72
Wt. Avg.	1.024	1.255	1.012	1.200	1.017	1.363	1.018	2.852

Table I4-3c: Effect of preloading in complete-subblock TLBs (256-block 4-way set-associative)

Workload	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty
coral	1.033	1.14	1.067	1.35	1.100	2.00	1.024	4.17
nasa7	1.003	1.96	1.000	1.37	1.000	1.47	1.000	1.52
compress	1.000	1.39	1.000	1.67	1.000	1.82	1.000	1.85
fftpde	1.000	1.00	1.001	1.01	1.001	1.01	1.000	1.06
wave5	1.000	1.75	1.000	2.78	1.000	3.70	1.000	1.27
mp3d	1.001	1.67	1.000	3.33	1.000	1.12	1.000	1.20
spice	1.000	1.85	1.000	2.70	1.000	1.47	1.000	1.54
pthor	1.009	1.39	1.008	2.00	1.004	3.45	1.000	1.92
ML	1.004	1.67	1.004	3.03	1.002	5.56	1.000	7.14
gcc	1.000	1.33	1.000	1.52	1.000	1.67	1.000	1.72
Wt. Avg.	1.006	1.076	1.009	1.211	1.011	1.393	1.002	4.026

Table I4-3d: Effect of preloading in complete-subblock TLBs (512-block 4-way set-associative)

Workload	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty
coral	1.025	1.16	1.040	1.43	1.012	2.38	1.000	1.15
nasa7	1.000	1.14	1.000	1.37	1.000	1.43	1.000	1.52
compress	1.000	1.37	1.000	1.61	1.000	1.75	1.000	1.82
fftpde	1.001	1.00	1.000	1.00	1.000	1.04	1.000	1.05
wave5	1.000	1.72	1.000	2.63	1.000	1.19	1.000	1.18
mp3d	1.000	1.85	1.000	1.06	1.000	1.09	1.000	1.11
spice	1.000	1.56	1.000	1.37	1.000	1.47	1.000	1.54
pthor	1.002	1.37	1.001	2.27	1.000	1.54	1.000	1.12
ML	1.002	1.82	1.001	3.33	1.000	4.35	1.000	1.27
gcc	1.000	1.28	1.000	1.49	1.000	1.64	1.000	1.72
Wt. Avg.	1.003	1.090	1.005	1.164	1.001	2.321	1.000	1.446

Table I4-5a: Execution time speedups relative to single-page-size (4KB) TLBs of equal area (64-block single-page-size TLB)

Workload	Superpage TLB (32KB)		Complete-subblock TLB (NO preloading)				Complete-subblock TLB (with preloading)			
	OS	NO OS	2	4	8	16	2	4	8	16
coral	1.422	0.993	1.032	1.025	0.959	0.874	1.123	1.227	1.283	1.217
nasa7	1.639	0.999	1.005	1.040	1.053	0.915	1.205	1.405	1.518	1.285
compress	1.274	0.984	1.183	1.255	1.243	0.850	1.217	1.268	1.267	1.126
fftpde	1.001	1.000	1.000	1.000	1.000	1.000	1.001	1.001	1.002	1.112
wave5	1.152	1.000	1.011	1.055	1.073	0.843	1.066	1.121	1.135	1.005
mp3d	1.122	0.999	0.999	0.991	0.977	0.907	1.020	1.030	1.024	0.950
spice	1.072	0.991	1.022	1.014	0.899	0.471	1.035	1.038	0.964	0.580
pthor	1.039	1.000	1.001	0.999	0.975	0.823	1.014	1.025	1.009	0.879
ML	1.034	0.998	1.001	0.995	0.968	0.819	1.010	1.011	0.997	0.880
gcc	1.020	0.998	1.006	1.009	0.978	0.807	1.010	1.015	1.010	0.939
Wt. Avg.	1.182	0.996	1.025	1.036	1.008	0.813	1.076	1.120	1.125	0.984

Table I4-5b: Execution time speedups relative to single-page-size (4KB) TLBs of equal area (128-block single-page-size TLB)

Workload	Superpage TLB (32KB)		Complete-subblock TLB (NO preloading)				Complete-subblock TLB (with preloading)			
	OS	NO OS	2	4	8	16	2	4	8	16
coral	1.361	0.988	1.013	1.007	0.983	0.913	1.078	1.146	1.205	1.222
nasa7	1.632	0.999	1.075	1.148	1.315	1.314	1.250	1.449	1.582	1.606
compress	1.010	0.999	1.010	1.010	1.010	1.004	1.010	1.010	1.010	1.010
fftpde	1.002	1.000	1.000	1.000	1.000	1.000	1.001	1.001	1.002	1.113
wave5	1.090	0.987	1.086	1.090	1.090	1.087	1.087	1.090	1.090	1.089
mp3d	1.073	0.997	1.024	1.067	1.069	1.042	1.041	1.070	1.071	1.067
spice	1.013	0.998	1.008	1.011	1.011	1.011	1.010	1.012	1.013	1.013
pthor	1.037	1.000	1.001	1.001	0.999	0.992	1.013	1.024	1.030	1.031
ML	1.019	0.999	1.002	1.002	0.998	0.980	1.008	1.012	1.011	1.003
gcc	1.004	1.000	1.002	1.003	1.003	1.000	1.003	1.004	1.004	1.004
Wt. Avg.	1.127	0.996	1.023	1.036	1.048	1.030	1.056	1.087	1.104	1.120

Table I4-5c: Execution time speedups relative to single-page-size (4KB) TLBs of equal area (256-block single-page-size TLB)

Workload	Superpage TLB (32KB)		Complete-subblock TLB (NO preloading)				Complete-subblock TLB (with preloading)			
	OS	NO OS	2	4	8	16	2	4	8	16
coral	1.252	0.994	1.003	0.987	0.965	0.921	1.033	1.056	1.090	1.102
nasa7	1.366	0.997	1.192	1.366	1.366	1.366	1.273	1.366	1.366	1.366
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	1.002	1.000	1.000	1.000	0.797	0.804	1.001	1.002	0.798	0.891
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	1.005	0.998	1.003	1.003	1.003	1.003	1.004	1.004	1.005	1.005
spice	1.001	1.000	1.001	1.001	1.001	1.001	1.001	1.001	1.001	1.001
pthor	1.040	0.999	1.002	1.003	1.002	0.997	1.013	1.022	1.027	1.031
ML	1.012	1.000	1.002	1.003	1.003	1.001	1.006	1.009	1.011	1.011
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	1.070	0.999	1.022	1.034	1.007	1.000	1.035	1.046	1.026	1.041

Table I4-5d: Execution time speedups relative to single-page-size (4KB) TLBs of equal area (512-block single-page-size TLB)

Workload	Superpage TLB (32KB)		Complete-subblock TLB (NO preloading)				Complete-subblock TLB (with preloading)			
	OS	NO OS	2	4	8	16	2	4	8	16
coral	1.231	0.994	0.996	0.977	0.962	0.939	1.023	1.036	1.074	1.103
nasa7	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	1.003	1.000	1.000	1.000	1.001	1.001	1.001	1.002	1.002	1.002
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	1.002	1.000	1.000	1.002	1.002	1.002	1.001	1.002	1.002	1.002
spice	1.001	1.000	1.001	1.001	1.001	1.001	1.001	1.001	1.001	1.001
pthor	1.021	0.998	1.010	1.014	1.007	1.000	1.013	1.018	1.016	1.016
ML	1.008	1.000	1.002	1.004	1.005	1.005	1.005	1.007	1.007	1.008
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	1.027	0.999	1.001	0.999	0.997	0.993	1.005	1.007	1.011	1.014

Table I4-6a: Execution time speedups for complete-subblock TLBs relative to single-page-size (4KB) TLBs with same TLB reach (256-block fully-associative)

Workload	Complete-subblock N blocks: s subblock factor				Complete-subblock with preloading N blocks: s subblock factor			
	128:2	64:4	32:8	16:16	128:2	64:4	32:8	16:16
coral	0.938	0.818	0.761	0.693	0.968	0.937	0.967	0.972
nasa7	0.921	0.923	0.934	0.929	1.061	1.194	1.277	1.318
compress	1.000	1.000	0.990	0.984	1.000	1.000	0.998	0.998
fftpde	0.797	0.797	0.797	0.797	0.797	0.798	0.798	0.887
wave5	1.000	1.000	0.999	0.967	1.000	1.000	1.000	0.978
mp3d	0.996	0.970	0.915	0.876	0.999	0.991	0.973	0.944
spice	1.000	0.997	0.990	0.832	1.001	1.000	0.997	0.914
pthor	0.995	0.992	0.987	0.957	1.008	1.015	1.019	0.997
ML	0.997	0.990	0.976	0.939	1.002	1.002	0.994	0.972
gcc	1.000	0.999	0.998	0.988	1.000	1.000	1.000	0.996
Wt. Avg.	0.958	0.937	0.920	0.878	0.981	0.989	0.998	0.995

Table I4-6b: Execution time speedups for complete-subblock TLBs relative to single-page-size (4KB) TLBs with same TLB reach (512-block fully-associative)

Workload	Complete-subblock N blocks: s subblock factor				Complete-subblock with preloading N blocks: s subblock factor			
	256:2	128:4	64:8	32:16	256:2	128:4	64:8	32:16
coral	0.921	0.864	0.800	0.710	0.949	0.925	0.925	0.927
nasa7	1.000	1.000	0.999	0.958	1.000	1.000	1.000	0.997
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	1.000	0.797	0.797	0.797	1.001	0.798	0.798	0.887
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	1.000	1.000	0.999	0.999	1.001	1.001	1.001	1.001
spice	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
pthor	0.988	0.981	0.977	0.973	0.997	1.001	1.005	1.008
ML	0.999	0.997	0.993	0.984	1.003	1.004	1.004	1.000
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	0.988	0.956	0.945	0.923	0.994	0.967	0.968	0.979

Table I4-6c: Execution time speedups for complete-subblock TLBs relative to single-page-size (4KB) TLBs with same TLB reach (256-block 4-way set-associative)

Workload	Complete-subblock N blocks: s subblock factor				Complete-subblock with preloading N blocks: s subblock factor			
	128:2	64:4	32:8	16:16	128:2	64:4	32:8	16:16
coral	0.920	0.837	0.766	0.699	0.965	0.958	0.975	0.965
nasa7	0.946	0.937	0.930	0.922	1.049	1.127	1.175	1.198
compress	1.000	0.998	0.992	0.982	1.000	1.000	0.998	0.998
fftpde	0.994	0.982	0.960	0.920	0.994	0.983	0.964	0.996
wave5	1.000	0.999	0.997	0.881	1.000	1.000	0.999	0.932
mp3d	0.991	0.969	0.931	0.882	0.997	0.989	0.973	0.939
spice	0.993	0.976	0.927	0.724	0.995	0.983	0.951	0.809
pthor	0.994	0.989	0.968	0.918	1.006	1.012	1.003	0.966
ML	0.995	0.984	0.962	0.903	1.001	0.998	0.983	0.945
gcc	0.999	0.996	0.988	0.966	1.000	0.999	0.994	0.982
Wt. Avg.	0.979	0.958	0.930	0.866	1.001	1.005	1.003	0.974

Table I4-6d: Execution time speedups for complete-subblock TLBs relative to single-page-size (4KB) TLBs with same TLB reach (512-block 4-way set-associative)

Workload	Complete-subblock N blocks: s subblock factor				Complete-subblock with preloading N blocks: s subblock factor			
	256:2	128:4	64:8	32:16	256:2	128:4	64:8	32:16
coral	0.922	0.852	0.779	0.703	0.952	0.930	0.935	0.936
nasa7	0.997	0.992	0.981	0.961	1.000	1.000	1.000	1.000
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	0.997	0.991	0.979	0.960	0.997	0.991	0.982	1.050
wave5	1.000	1.000	1.000	0.994	1.000	1.000	1.000	0.997
mp3d	1.000	0.999	0.997	0.987	1.001	1.001	1.000	0.997
spice	1.000	1.000	1.000	0.992	1.000	1.000	1.000	0.997
pthor	0.989	0.981	0.973	0.961	0.997	1.000	1.002	0.999
ML	0.998	0.996	0.990	0.976	1.002	1.003	1.002	0.995
gcc	1.000	1.000	0.999	0.998	1.000	1.000	1.000	1.000
Wt. Avg.	0.988	0.976	0.960	0.936	0.994	0.991	0.990	0.997

Table I4-9a: Execution time speedups for complete-subblock TLBs relative to superpage TLBs (64-block fully-associative)

Workload	subblock factor:superpage size				With preloading subblock factor:superpage size			
	2:8KB	4:16KB	8:32KB	16:64KB	2:8KB	4:16KB	8:32KB	16:64KB
coral	0.925	0.876	0.870	0.853	1.001	1.003	1.006	1.009
nasa7	0.836	0.774	1.002	1.000	1.000	1.001	1.002	1.000
compress	0.995	1.001	1.000	1.003	1.001	1.001	1.000	1.003
fftpde	0.999	0.999	0.999	0.897	1.000	1.000	1.001	0.997
wave5	0.959	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	0.981	0.985	0.999	0.999	1.003	1.007	1.000	1.000
spice	0.994	0.998	0.999	1.000	1.001	1.000	1.000	1.000
pthor	0.988	0.978	0.973	0.975	1.001	1.002	1.002	1.003
ML	0.994	0.991	0.992	0.995	1.002	1.003	1.002	1.002
gcc	0.999	1.000	1.000	1.000	1.001	1.001	1.000	1.000
Wt. Avg.	0.958	0.948	0.978	0.965	1.001	1.002	1.002	1.002

Table I4-9b: Execution time speedups for complete-subblock TLBs relative to superpage TLBs (128-block fully-associative)

Workload	subblock factor:superpage size				With preloading subblock factor:superpage size			
	2:8KB	4:16KB	8:32KB	16:64KB	2:8KB	4:16KB	8:32KB	16:64KB
coral	0.969	0.936	0.894	0.871	1.001	1.003	1.006	1.010
nasa7	0.869	1.000	1.000	1.000	1.001	1.000	1.000	1.000
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	0.999	0.999	0.999	0.999	1.000	1.000	1.001	1.000
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	0.998	0.999	0.999	1.000	1.001	1.000	1.000	1.000
spice	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000
pthor	0.988	0.981	0.983	0.995	1.001	1.001	1.001	1.003
ML	0.995	0.994	0.996	0.998	1.001	1.001	1.001	1.000
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	0.977	0.989	0.984	0.983	1.000	1.001	1.001	1.001

Table I4-9c: Execution time speedups for complete-subblock TLBs relative to superpage TLBs (256-block 4-way set-associative)

Workload	subblock factor:superpage size				With preloading subblock factor:superpage size			
	2:8KB	4:16KB	8:32KB	16:64KB	2:8KB	4:16KB	8:32KB	16:64KB
coral	0.969	0.940	0.918	1.020	1.001	1.003	1.010	1.044
nasa7	0.997	1.000	1.003	1.017	1.000	1.000	1.003	1.017
compress	1.000	1.000	1.019	1.231	1.000	1.000	1.020	1.231
fftpde	1.000	0.999	1.000	1.006	1.000	1.000	1.001	1.006
wave5	1.000	1.000	1.000	1.004	1.000	1.000	1.000	1.004
mp3d	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000
spice	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
pthor	0.992	1.002	1.006	1.204	1.001	1.010	1.010	1.204
ML	0.996	0.997	1.002	1.008	1.000	1.001	1.004	1.008
gcc	1.000	1.000	1.071	1.174	1.000	1.000	1.071	1.174
Wt. Avg.	0.995	0.992	1.000	1.066	1.000	1.001	1.012	1.069

Table I5-1a: Execution time speedup with partial-subblock TLBs relative to single-page-size (4KB) TLBs with same number of blocks (64-block fully-associative)

Workload	Partial-subblock subblock factor				Partial-subblock with preloading subblock factor			
	2	4	8	16	2	4	8	16
coral	1.064	1.132	1.239	1.324	1.150	1.297	1.435	1.570
nasa7	1.009	1.114	1.647	1.649	1.207	1.442	1.649	1.649
compress	1.256	1.273	1.274	1.274	1.264	1.274	1.275	1.275
fftpde	1.000	1.000	1.000	1.008	1.001	1.001	1.001	1.117
wave5	1.042	1.151	1.151	1.152	1.087	1.152	1.152	1.152
mp3d	1.012	1.068	1.120	1.121	1.036	1.100	1.122	1.123
spice	1.047	1.068	1.072	1.072	1.054	1.071	1.072	1.073
pthor	1.003	1.007	1.010	1.015	1.016	1.030	1.040	1.046
ML	1.010	1.018	1.024	1.031	1.017	1.029	1.036	1.040
gcc	1.013	1.019	1.021	1.021	1.015	1.020	1.021	1.021
Wt. Avg.	1.044	1.087	1.156	1.169	1.090	1.149	1.184	1.214

Table I5-1b: Execution time speedup with partial-subblock TLBs relative to single-page-size (4KB) TLBs with same number of blocks (128-block fully-associative)

Workload	Partial-subblock subblock factor				Partial-subblock with preloading subblock factor			
	2	4	8	16	2	4	8	16
coral	1.134	1.172	1.221	1.327	1.171	1.255	1.373	1.539
nasa7	1.100	1.631	1.632	1.632	1.266	1.632	1.632	1.632
compress	1.010	1.010	1.010	1.010	1.010	1.010	1.010	1.010
fftpde	1.000	1.000	1.000	1.257	1.001	1.001	1.001	1.258
wave5	1.090	1.090	1.090	1.090	1.090	1.090	1.090	1.090
mp3d	1.062	1.071	1.072	1.073	1.066	1.072	1.073	1.073
spice	1.011	1.012	1.013	1.013	1.012	1.013	1.013	1.013
pthor	1.003	1.007	1.019	1.030	1.015	1.027	1.038	1.045
ML	1.005	1.009	1.014	1.017	1.010	1.016	1.020	1.021
gcc	1.003	1.004	1.004	1.004	1.003	1.004	1.004	1.004
Wt. Avg.	1.047	1.101	1.109	1.150	1.072	1.114	1.128	1.172

Table I5-1c: Execution time speedup with partial-subblock TLBs relative to single-page-size (4KB) TLBs with same number of blocks (256-block 4-way set-associative)

Workload	Partial-subblock subblock factor				Partial-subblock with preloading subblock factor			
	2	4	8	16	2	4	8	16
coral	1.036	1.077	1.166	1.366	1.070	1.148	1.282	1.408
nasa7	1.494	1.503	1.503	1.503	1.498	1.503	1.503	1.503
compress	1.000	1.000	1.000	0.964	1.000	1.000	1.000	0.964
fftpde	1.000	1.191	1.214	1.441	1.000	1.192	1.214	1.441
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	1.008	1.009	1.010	1.009	1.009	1.010	1.010	1.010
spice	1.002	1.003	1.003	1.003	1.003	1.003	1.003	1.003
pthor	1.007	1.022	1.032	0.904	1.016	1.031	1.037	0.908
ML	1.004	1.008	1.009	1.007	1.008	1.012	1.011	1.009
gcc	1.001	1.001	0.998	0.985	1.001	1.001	0.999	0.987
Wt. Avg.	1.052	1.083	1.097	1.115	1.058	1.093	1.110	1.120

Table I5-3a: Effect of preloading in partial-subblock TLBs (64-block fully-associative)

Workload	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty
coral	1.081	1.20	1.145	1.43	1.158	1.61	1.186	1.96
nasa7	1.197	1.72	1.293	3.33	1.001	5.56	1.000	3.33
compress	1.006	1.69	1.000	1.59	1.000	1.39	1.000	1.35
fftpde	1.001	1.00	1.001	1.01	1.001	1.01	1.109	1.96
wave5	1.043	1.75	1.000	2.94	1.000	4.17	1.000	4.76
mp3d	1.024	1.32	1.030	2.50	1.002	2.94	1.001	9.09
spice	1.008	1.45	1.003	2.78	1.001	6.25	1.000	7.69
pthor	1.013	1.33	1.023	1.89	1.029	2.86	1.031	4.17
ML	1.008	1.32	1.011	1.85	1.011	2.63	1.009	4.35
gcc	1.002	1.37	1.001	2.22	1.000	2.04	1.000	1.72
Wt. Avg.	1.045	1.332	1.057	1.649	1.024	1.409	1.038	2.045

Table I5-3b: Effect of preloading in partial-subblock TLBs (128-block fully-associative)

Workload	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty
coral	1.033	1.10	1.071	1.27	1.124	1.64	1.159	2.70
nasa7	1.151	1.67	1.000	3.12	1.000	1.23	1.000	1.41
compress	1.000	1.23	1.000	1.23	1.000	1.27	1.000	1.27
fftpde	1.001	1.00	1.001	1.01	1.001	1.01	1.001	6.67
wave5	1.000	1.75	1.000	2.78	1.000	3.85	1.000	4.35
mp3d	1.004	1.56	1.001	2.50	1.001	5.56	1.000	1.15
spice	1.001	1.89	1.000	3.45	1.000	1.64	1.000	1.43
pthor	1.012	1.37	1.020	1.96	1.019	2.94	1.014	4.55
ML	1.005	1.49	1.007	2.50	1.006	4.35	1.003	6.67
gcc	1.000	1.45	1.000	1.43	1.000	1.56	1.000	1.56
Wt. Avg.	1.024	1.254	1.012	1.200	1.017	1.360	1.019	2.79

Table I5-3c: Effect of preloading in partial-subblock TLBs (256-block fully-associative)

Workload	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty
coral	1.030	1.12	1.062	1.33	1.097	1.96	1.032	3.85
nasa7	1.000	1.64	1.000	1.25	1.000	1.35	1.000	1.41
compress	1.000	1.23	1.000	1.19	1.000	1.22	1.000	1.23
fftpde	1.001	1.85	1.002	3.45	1.001	4.76	1.000	1.01
wave5	1.000	1.69	1.000	2.70	1.000	3.57	1.000	1.25
mp3d	1.001	1.69	1.001	3.33	1.000	1.05	1.000	1.14
spice	1.000	1.92	1.000	1.30	1.000	1.37	1.000	1.45
pthor	1.009	1.39	1.010	2.00	1.001	3.03	1.000	2.38
ML	1.004	1.69	1.004	3.03	1.002	5.00	1.001	7.14
gcc	1.000	1.22	1.000	1.39	1.000	1.56	1.000	1.56
Wt. Avg.	1.005	1.155	1.009	1.377	1.011	1.995	1.003	3.756

Table I5-3d: Effect of preloading in partial-subblock TLBs (256-block 4-way set-associative)

Workload	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty
coral	1.033	1.14	1.066	1.33	1.099	1.96	1.031	3.23
nasa7	1.003	1.92	1.000	2.13	1.000	1.75	1.000	1.04
compress	1.000	1.14	1.000	1.30	1.000	1.33	1.000	1.00
fftpde	1.000	1.00	1.001	1.01	1.000	1.00	1.000	1.05
wave5	1.000	1.72	1.000	2.70	1.000	3.57	1.000	1.56
mp3d	1.001	1.67	1.001	2.86	1.000	2.33	1.001	3.12
spice	1.000	1.85	1.000	2.63	1.000	1.22	1.000	1.30
pthor	1.009	1.37	1.009	1.92	1.005	2.86	1.005	1.04
ML	1.004	1.59	1.004	2.50	1.002	1.56	1.002	1.27
gcc	1.000	1.23	1.000	1.41	1.001	1.28	1.002	1.15
Wt. Avg.	1.006	1.076	1.009	1.209	1.012	1.383	1.004	1.190

Table I5-3e: Effect of preloading in partial-subblock TLBs (512-block 4-way set-associative)

Workload	subblock factor 2		subblock factor 4		subblock factor 8		subblock factor 16	
	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty	Speedup	critical miss penalty
coral	1.025	1.15	1.040	1.43	1.014	2.13	1.004	2.33
nasa7	1.000	1.18	1.000	1.30	1.000	1.32	1.000	1.04
compress	1.000	1.16	1.000	1.23	1.000	1.35	1.004	1.11
fftpde	1.001	1.00	1.000	1.00	1.000	1.03	1.000	1.06
wave5	1.000	1.72	1.000	2.56	1.000	1.61	1.000	1.25
mp3d	1.000	1.85	1.000	1.52	1.000	1.11	1.000	1.15
spice	1.000	1.59	1.000	1.30	1.000	1.35	1.000	1.28
pthor	1.002	1.33	1.002	2.08	1.001	2.63	1.004	1.03
ML	1.002	1.69	1.001	2.63	1.000	1.23	1.000	1.01
gcc	1.000	1.22	1.000	1.41	1.000	1.02	1.000	1.00
Wt. Avg.	1.003	1.088	1.005	1.165	1.002	1.809	1.001	1.059

Table I5-4a: Execution time speedups for partial-subblock TLBs (with preloading) relative to similar single-page-size (4KB) TLBs (64-block fully-associative)

Workload	Partial-subblock subblock factor With OS support				Partial-subblock subblock factor Without OS support			
	2	4	8	16	2	4	8	16
coral	1.150	1.297	1.435	1.570	1.000	1.000	1.000	1.000
nasa7	1.207	1.442	1.649	1.649	1.001	1.002	1.000	1.000
compress	1.264	1.274	1.275	1.275	0.999	0.999	1.000	0.999
fftpde	1.001	1.001	1.001	1.117	1.000	1.000	1.000	1.000
wave5	1.087	1.152	1.152	1.152	1.000	1.000	1.000	1.000
mp3d	1.036	1.100	1.122	1.123	1.001	1.001	1.000	1.000
spice	1.054	1.071	1.072	1.073	1.000	1.000	1.000	1.000
pthor	1.016	1.030	1.040	1.046	1.000	1.000	1.000	1.000
ML	1.017	1.029	1.036	1.040	1.000	1.000	1.000	1.000
gcc	1.015	1.020	1.021	1.021	1.000	1.000	1.000	1.000
Wt. Avg.	1.090	1.149	1.184	1.214	1.000	1.000	1.000	1.000

Table I5-4b: Execution time speedups for partial-subblock TLBs (with preloading) relative to similar single-page-size (4KB) TLBs (256-block 4-way set-associative)

Workload	Partial-subblock subblock factor With OS support				Partial-subblock subblock factor Without OS support			
	2	4	8	16	2	4	8	16
coral	1.070	1.148	1.282	1.408	0.975	0.906	0.710	0.688
nasa7	1.498	1.503	1.503	1.503	1.162	1.186	0.880	0.850
compress	1.000	1.000	1.000	0.964	1.000	0.993	0.683	0.542
fftpde	1.000	1.192	1.214	1.441	1.000	1.190	1.210	1.431
wave5	1.000	1.000	1.000	1.000	0.994	0.959	0.744	0.673
mp3d	1.009	1.010	1.010	1.010	0.998	0.996	0.935	0.896
spice	1.003	1.003	1.003	1.003	0.992	0.975	0.876	0.794
pthor	1.016	1.031	1.037	0.908	0.999	0.988	0.982	0.823
ML	1.008	1.012	1.011	1.009	0.998	0.991	0.972	0.954
gcc	1.001	1.001	0.999	0.987	0.999	0.993	0.886	0.774
Wt. Avg.	1.058	1.093	1.110	1.120	1.013	1.019	0.868	0.806

Table I5-7a: Execution time speedups relative to single-page-size (4KB) TLBs of equal area (64-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	1.422	1.064	1.130	1.211	1.296	1.032	1.025	0.959	0.874
nasa7	1.639	1.009	1.109	1.554	1.649	1.005	1.040	1.053	0.915
compress	1.274	1.256	1.273	1.274	1.274	1.183	1.255	1.243	0.850
fftpde	1.001	1.000	1.000	1.000	1.009	1.000	1.000	1.000	1.000
wave5	1.152	1.042	1.151	1.151	1.152	1.011	1.055	1.073	0.843
mp3d	1.122	1.012	1.062	1.120	1.121	0.999	0.991	0.977	0.907
spice	1.072	1.047	1.068	1.072	1.072	1.022	1.014	0.899	0.471
pthor	1.039	1.003	1.006	1.009	1.012	1.001	0.999	0.975	0.823
ML	1.034	1.010	1.017	1.023	1.028	1.001	0.995	0.968	0.819
gcc	1.020	1.013	1.019	1.020	1.021	1.006	1.009	0.978	0.807
Wt. Avg.	1.182	1.044	1.085	1.145	1.165	1.025	1.036	1.008	0.813

Table I5-7b: Execution time speedups relative to single-page-size (4KB) TLBs of equal area (128-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	1.361	1.133	1.170	1.213	1.288	1.013	1.007	0.983	0.913
nasa7	1.632	1.098	1.631	1.632	1.632	1.075	1.148	1.315	1.314
compress	1.010	1.010	1.010	1.010	1.010	1.010	1.010	1.010	1.004
fftpde	1.002	1.000	1.000	1.000	1.256	1.000	1.000	1.000	1.000
wave5	1.090	1.090	1.090	1.090	1.090	1.086	1.090	1.090	1.087
mp3d	1.073	1.061	1.071	1.072	1.073	1.024	1.067	1.069	1.042
spice	1.013	1.011	1.012	1.013	1.013	1.008	1.011	1.011	1.011
pthor	1.037	1.003	1.007	1.017	1.027	1.001	1.001	0.999	0.992
ML	1.019	1.005	1.009	1.014	1.017	1.002	1.002	0.998	0.980
gcc	1.004	1.003	1.004	1.004	1.004	1.002	1.003	1.003	1.000
Wt. Avg.	1.127	1.047	1.100	1.107	1.145	1.023	1.036	1.048	1.030

Table I5-7c: Execution time speedups relative to single-page-size (4KB) TLBs of equal area (256-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	1.252	1.034	1.065	1.139	1.308	1.003	0.987	0.965	0.921
nasa7	1.366	1.366	1.366	1.366	1.366	1.192	1.366	1.366	1.366
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	1.002	1.000	1.000	1.001	1.002	1.000	1.000	0.797	0.804
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	1.005	1.003	1.004	1.005	1.005	1.003	1.003	1.003	1.003
spice	1.001	1.001	1.001	1.001	1.001	1.001	1.001	1.001	1.001
pthor	1.040	1.007	1.019	1.040	1.040	1.002	1.003	1.002	0.997
ML	1.012	1.004	1.007	1.010	1.011	1.002	1.003	1.003	1.001
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	1.070	1.041	1.047	1.058	1.075	1.022	1.034	1.007	1.000

Table I5-7d: Execution time speedups relative to single-page-size (4KB) TLBs of equal area (512-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	1.231	1.035	1.087	1.210	1.249	0.996	0.977	0.962	0.939
nasa7	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	1.003	1.000	1.002	1.002	1.003	1.000	1.000	1.001	1.001
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	1.002	1.001	1.002	1.002	1.002	1.000	1.002	1.002	1.002
spice	1.001	1.001	1.001	1.001	1.001	1.001	1.001	1.001	1.001
pthor	1.021	1.018	1.020	1.021	1.021	1.010	1.014	1.007	1.000
ML	1.008	1.003	1.006	1.008	1.008	1.002	1.004	1.005	1.005
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	1.027	1.006	1.013	1.025	1.028	1.001	0.999	0.997	0.993

Table I5-8a: Execution time speedups using preloading in subblock TLBs relative to single-page-size (4KB) TLB of equal area (64-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	1.422	1.150	1.295	1.423	1.543	1.123	1.227	1.283	1.217
nasa7	1.639	1.207	1.439	1.639	1.649	1.205	1.405	1.518	1.285
compress	1.274	1.264	1.274	1.275	1.275	1.217	1.268	1.267	1.126
fftpde	1.001	1.001	1.001	1.001	1.118	1.001	1.001	1.002	1.112
wave5	1.152	1.087	1.152	1.152	1.152	1.066	1.121	1.135	1.005
mp3d	1.122	1.036	1.097	1.122	1.123	1.020	1.030	1.024	0.950
spice	1.072	1.054	1.071	1.072	1.072	1.035	1.038	0.964	0.580
pthor	1.039	1.016	1.030	1.039	1.046	1.014	1.025	1.009	0.879
ML	1.034	1.017	1.029	1.035	1.039	1.010	1.011	0.997	0.880
gcc	1.020	1.015	1.020	1.021	1.021	1.010	1.015	1.010	0.939
Wt. Avg.	1.182	1.090	1.148	1.182	1.211	1.076	1.120	1.125	0.984

Table I5-8b: Execution time speedups using preloading in subblock TLBs relative to single-page-size (4KB) TLB of equal area (128-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	1.361	1.170	1.253	1.364	1.508	1.078	1.146	1.205	1.222
nasa7	1.632	1.266	1.632	1.632	1.632	1.250	1.449	1.582	1.606
compress	1.010	1.010	1.010	1.010	1.010	1.010	1.010	1.010	1.010
fftpde	1.002	1.001	1.001	1.001	1.258	1.001	1.001	1.002	1.113
wave5	1.090	1.090	1.090	1.090	1.090	1.087	1.090	1.090	1.089
mp3d	1.073	1.065	1.072	1.073	1.073	1.041	1.070	1.071	1.067
spice	1.013	1.012	1.013	1.013	1.013	1.010	1.012	1.013	1.013
pthor	1.037	1.015	1.027	1.038	1.044	1.013	1.024	1.030	1.031
ML	1.019	1.010	1.016	1.019	1.021	1.008	1.012	1.011	1.003
gcc	1.004	1.003	1.004	1.004	1.004	1.003	1.004	1.004	1.004
Wt. Avg.	1.127	1.072	1.114	1.127	1.169	1.056	1.087	1.104	1.120

Table I5-8c: Execution time speedups using preloading in subblock TLBs relative to single-page-size (4KB) TLB of equal area (256-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	1.252	1.064	1.133	1.254	1.375	1.033	1.056	1.090	1.102
nasa7	1.366	1.366	1.366	1.366	1.366	1.273	1.366	1.366	1.366
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	1.002	1.001	1.002	1.002	1.003	1.001	1.002	0.798	0.891
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	1.005	1.004	1.005	1.005	1.005	1.004	1.004	1.005	1.005
spice	1.001	1.001	1.001	1.001	1.001	1.001	1.001	1.001	1.001
pthor	1.040	1.016	1.030	1.040	1.041	1.013	1.022	1.027	1.031
ML	1.012	1.007	1.011	1.012	1.013	1.006	1.009	1.011	1.011
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	1.070	1.046	1.057	1.070	1.081	1.035	1.046	1.026	1.041

Table I5-8d: Execution time speedups using preloading in subblock TLBs relative to single-page-size (4KB) TLB of equal area (512-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	1.231	1.060	1.131	1.231	1.249	1.023	1.036	1.074	1.103
nasa7	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	1.003	1.001	1.002	1.003	1.003	1.001	1.002	1.002	1.002
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	1.002	1.001	1.002	1.002	1.002	1.001	1.002	1.002	1.002
spice	1.001	1.001	1.001	1.001	1.001	1.001	1.001	1.001	1.001
pthor	1.021	1.019	1.021	1.021	1.021	1.013	1.018	1.016	1.016
ML	1.008	1.005	1.007	1.008	1.008	1.005	1.007	1.007	1.008
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	1.027	1.010	1.018	1.027	1.028	1.005	1.007	1.011	1.014

Table I5-9a: Execution time speedups using base pages relative to single-page-size (4KB) TLBs of equal area (without preloading) (64-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	0.993	0.999	0.997	0.990	0.984	1.032	1.025	0.959	0.874
nasa7	0.999	1.001	1.001	0.998	0.941	1.005	1.040	1.053	0.915
compress	0.984	1.000	0.992	0.976	0.946	1.183	1.255	1.243	0.850
fftpde	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
wave5	1.000	1.000	1.000	0.999	0.997	1.011	1.055	1.073	0.843
mp3d	0.999	1.000	0.999	0.998	0.995	0.999	0.991	0.977	0.907
spice	0.991	0.997	0.994	0.988	0.973	1.022	1.014	0.899	0.471
pthor	1.000	1.000	1.000	0.999	0.999	1.001	0.999	0.975	0.823
ML	0.998	1.000	1.000	0.997	0.993	1.001	0.995	0.968	0.819
gcc	0.998	1.000	0.999	0.997	0.994	1.006	1.009	0.978	0.807
Wt. Avg.	0.996	1.000	0.998	0.994	0.980	1.025	1.036	1.008	0.813

Table I5-9b: Execution time speedups using base pages relative to single-page-size (4KB) TLBs of equal area (without preloading) (128-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	0.988	0.997	0.995	0.988	0.981	1.013	1.007	0.983	0.913
nasa7	0.999	1.000	1.000	0.999	0.997	1.075	1.148	1.315	1.314
compress	0.999	1.000	0.999	0.999	0.997	1.010	1.010	1.010	1.004
fftpde	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
wave5	0.987	0.996	0.993	0.985	0.981	1.086	1.090	1.090	1.087
mp3d	0.997	0.999	0.999	0.996	0.990	1.024	1.067	1.069	1.042
spice	0.998	0.999	0.999	0.998	0.995	1.008	1.011	1.011	1.011
pthor	1.000	1.000	1.000	1.000	0.999	1.001	1.001	0.999	0.992
ML	0.999	1.000	0.999	0.999	0.998	1.002	1.002	0.998	0.980
gcc	1.000	1.000	1.000	1.000	0.999	1.002	1.003	1.003	1.000
Wt. Avg.	0.996	0.999	0.998	0.996	0.993	1.023	1.036	1.048	1.030

Table I5-9c: Execution time speedups using base pages relative to single-page-size (4KB) TLBs of equal area (without preloading) (256-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	0.994	0.999	0.998	0.991	0.981	1.003	0.987	0.965	0.921
nasa7	0.997	0.999	0.998	0.994	0.986	1.192	1.366	1.366	1.366
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.797	0.804
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	0.998	1.000	0.999	0.998	0.993	1.003	1.003	1.003	1.003
spice	1.000	1.000	1.000	1.000	0.999	1.001	1.001	1.001	1.001
pthor	0.999	1.000	1.000	0.999	0.998	1.002	1.003	1.002	0.997
ML	1.000	1.000	1.000	1.000	0.999	1.002	1.003	1.003	1.001
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	0.999	1.000	0.999	0.998	0.995	1.022	1.034	1.007	1.000

Table I5-9d: Execution time speedups using base pages relative to single-page-size (4KB) TLBs of equal area (without preloading) (512-block single-page-size TLB)

Workload	Superpage (32KB)	Partial-subblock				Complete-subblock			
		2	4	8	16	2	4	8	16
coral	0.994	0.999	0.997	0.992	0.980	0.996	0.977	0.962	0.939
nasa7	1.000	1.000	1.000	0.998	0.938	1.000	1.000	1.000	1.000
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.001	1.001
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	1.000	1.000	1.000	1.000	1.000	1.000	1.002	1.002	1.002
spice	1.000	1.000	1.000	1.000	1.000	1.001	1.001	1.001	1.001
pthor	0.998	1.000	0.999	0.997	0.994	1.010	1.014	1.007	1.000
ML	1.000	1.000	1.000	1.000	0.999	1.002	1.004	1.005	1.005
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	0.999	1.000	1.000	0.999	0.990	1.001	0.999	0.997	0.993

Table I5-11a: Execution time speedups for partial-subblock TLBs relative to single-page-size (4KB) TLBs with same TLB reach (256-block fully-associative)

Workload	Partial-subblock N blocks: s subblock factor				Partial-subblock with preloading N blocks: s subblock factor			
	128:2	64:4	32:8	16:16	128:2	64:4	32:8	16:16
coral	0.937	0.817	0.759	0.689	0.968	0.936	0.963	0.962
nasa7	0.921	0.923	0.929	0.918	1.060	1.194	1.274	1.312
compress	1.000	0.999	0.985	0.966	1.000	1.000	0.997	0.992
fftpde	0.797	0.797	0.797	0.797	0.797	0.798	0.798	0.886
wave5	1.000	1.000	0.997	0.965	1.000	1.000	0.999	0.977
mp3d	0.995	0.955	0.899	0.864	0.998	0.984	0.960	0.912
spice	1.000	0.997	0.988	0.797	1.001	1.000	0.996	0.889
pthor	0.995	0.991	0.984	0.874	1.007	1.014	1.016	0.948
ML	0.996	0.989	0.967	0.915	1.002	1.000	0.985	0.950
gcc	1.000	0.998	0.989	0.893	1.000	1.000	0.995	0.921
Wt. Avg.	0.957	0.935	0.915	0.853	0.980	0.988	0.994	0.972

Table I5-11b: Execution time speedups for partial-subblock TLBs relative to single-page-size (4KB) TLBs with same TLB reach (512-block fully-associative)

Workload	Partial-subblock N blocks: s subblock factor				Partial-subblock with preloading N blocks: s subblock factor			
	256:2	128:4	64:8	32:16	256:2	128:4	64:8	32:16
coral	0.921	0.863	0.796	0.706	0.948	0.924	0.922	0.923
nasa7	1.000	0.999	0.999	0.957	1.000	1.000	1.000	0.997
compress	1.000	1.000	1.000	0.998	1.000	1.000	1.000	0.999
fftpde	1.000	0.797	0.797	0.797	1.001	0.798	0.798	0.887
wave5	1.000	1.000	1.000	0.999	1.000	1.000	1.000	1.000
mp3d	1.000	1.000	0.999	0.998	1.001	1.001	1.001	1.000
spice	1.000	1.000	1.000	0.999	1.000	1.000	1.000	1.000
pthor	0.988	0.981	0.976	0.969	0.997	1.000	1.005	1.006
ML	0.999	0.996	0.991	0.973	1.003	1.003	1.001	0.991
gcc	1.000	1.000	1.000	0.992	1.000	1.000	1.000	0.996
Wt. Avg.	0.988	0.955	0.944	0.920	0.994	0.967	0.967	0.977

Table I5-11c: Execution time speedups for partial-subblock TLBs relative to single-page-size (4KB) TLBs with same TLB reach (256-block 4-way set-associative)

Workload	Partial-subblock N blocks: s subblock factor				Partial-subblock with preloading N blocks: s subblock factor			
	128:2	64:4	32:8	16:16	128:2	64:4	32:8	16:16
coral	0.920	0.835	0.763	0.698	0.964	0.955	0.970	0.961
nasa7	0.946	0.936	0.929	0.843	1.048	1.126	1.173	1.094
compress	0.999	0.997	0.987	0.936	1.000	0.999	0.996	0.948
fftpde	0.993	0.982	0.923	0.886	0.994	0.983	0.924	0.950
wave5	1.000	0.998	0.994	0.882	1.000	0.999	0.997	0.922
mp3d	0.989	0.960	0.917	0.777	0.994	0.981	0.960	0.807
spice	0.992	0.976	0.868	0.714	0.995	0.983	0.900	0.726
pthor	0.994	0.980	0.869	0.637	1.005	1.004	0.903	0.668
ML	0.994	0.980	0.944	0.860	1.000	0.991	0.966	0.894
gcc	0.998	0.968	0.916	0.808	0.999	0.973	0.930	0.867
Wt. Avg.	0.978	0.954	0.901	0.794	1.000	1.000	0.972	0.880

Table I5-11d: Execution time speedups for partial-subblock TLBs relative to single-page-size (4KB) TLBs with same TLB reach (512-block 4-way set-associative)

Workload	Partial-subblock N blocks: s subblock factor				Partial-subblock with preloading N blocks: s subblock factor			
	256:2	128:4	64:8	32:16	256:2	128:4	64:8	32:16
coral	0.922	0.851	0.777	0.701	0.952	0.928	0.931	0.927
nasa7	0.997	0.991	0.980	0.957	1.000	1.000	1.000	0.999
compress	1.000	1.000	0.997	0.942	1.000	1.000	0.999	0.955
fftpde	0.997	0.991	0.941	0.923	0.997	0.991	0.942	1.001
wave5	1.000	1.000	0.999	0.992	1.000	1.000	1.000	0.995
mp3d	1.000	0.994	0.988	0.935	1.001	0.998	0.996	0.955
spice	1.000	1.000	0.997	0.928	1.000	1.000	0.999	0.951
pthor	0.988	0.978	0.968	0.767	0.997	0.997	0.997	0.802
ML	0.998	0.994	0.979	0.946	1.002	1.001	0.993	0.968
gcc	1.000	1.000	0.989	0.951	1.000	1.000	0.993	0.961
Wt. Avg.	0.988	0.974	0.950	0.889	0.994	0.989	0.981	0.949

Table I5-13a: Execution time speedups for partial-subblock TLBs relative to superpage TLBs (64-block fully-associative)

Workload	subblock factor:superpage size				With preloading subblock factor:superpage size			
	2:8KB	4:16KB	8:32KB	16:64KB	2:8KB	4:16KB	8:32KB	16:64KB
coral	0.925	0.874	0.867	0.846	1.000	1.001	1.003	1.004
nasa7	0.836	0.773	1.001	1.000	1.000	1.000	1.002	1.000
compress	0.994	1.000	1.000	1.003	1.001	1.000	1.000	1.003
fftpde	0.999	0.999	0.999	0.902	1.000	1.000	1.000	1.000
wave5	0.959	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	0.978	0.971	0.998	0.999	1.001	1.000	1.000	1.000
spice	0.994	0.998	0.999	1.000	1.001	1.000	1.000	1.000
pthor	0.987	0.978	0.972	0.972	1.000	1.000	1.001	1.002
ML	0.993	0.990	0.989	0.992	1.001	1.001	1.000	1.001
gcc	0.998	0.999	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	0.958	0.946	0.977	0.964	1.000	1.000	1.001	1.001

Table I5-13b: Execution time speedups for partial-subblock TLBs relative to superpage TLBs (128-block fully-associative)

Workload	subblock factor:superpage size				With preloading subblock factor:superpage size			
	2:8KB	4:16KB	8:32KB	16:64KB	2:8KB	4:16KB	8:32KB	16:64KB
coral	0.969	0.935	0.892	0.866	1.000	1.001	1.003	1.004
nasa7	0.868	1.000	1.000	1.000	1.000	1.000	1.000	1.000
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	0.999	0.999	0.999	0.999	1.000	1.000	1.000	1.000
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	0.996	0.999	0.999	1.000	1.000	1.000	1.000	1.000
spice	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000
pthor	0.988	0.981	0.982	0.987	1.000	1.000	1.000	1.001
ML	0.995	0.993	0.995	0.997	1.000	1.000	1.000	1.000
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	0.976	0.988	0.983	0.982	1.000	1.000	1.000	1.001

Table I5-13c: Execution time speedups for partial-subblock TLBs relative to superpage TLBs (256-block 4-way set-associative)

Workload	subblock factor:superpage size				With preloading subblock factor:superpage size			
	2:8KB	4:16KB	8:32KB	16:64KB	2:8KB	4:16KB	8:32KB	16:64KB
coral	0.969	0.939	0.915	1.006	1.000	1.001	1.006	1.037
nasa7	0.997	1.000	1.003	1.017	1.000	1.000	1.003	1.017
compress	1.000	1.000	1.019	1.187	1.000	1.000	1.019	1.186
fftpde	1.000	0.999	1.000	1.005	1.000	1.000	1.000	1.005
wave5	1.000	1.000	1.000	1.004	1.000	1.000	1.000	1.004
mp3d	0.999	0.999	1.000	0.999	1.000	1.000	1.000	1.000
spice	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
pthor	0.991	1.000	1.003	1.046	1.000	1.009	1.008	1.052
ML	0.996	0.996	0.999	1.000	1.000	1.000	1.001	1.002
gcc	1.000	1.000	1.068	1.155	1.000	1.000	1.069	1.158
Wt. Avg.	0.994	0.992	0.999	1.043	1.000	1.001	1.010	1.047

Table I5-15a: Execution time speedups for partial-subblock TLBs relative to complete-subblock TLBs with same subblock factor, number of blocks and associativity (64-block fully-associative)

Workload	subblock factor				With preloading subblock factor			
	2	4	8	16	2	4	8	16
coral	1.000	0.998	0.996	0.992	0.999	0.999	0.997	0.995
nasa7	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
compress	0.999	0.999	1.000	1.000	0.999	0.999	1.000	1.000
fftpde	1.000	1.000	1.000	1.005	1.000	1.000	1.000	1.002
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	0.997	0.985	1.000	1.000	0.998	0.993	1.000	1.000
spice	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
pthor	0.999	0.999	0.999	0.997	0.999	0.999	0.999	0.999
ML	0.999	0.998	0.998	0.997	0.999	0.998	0.998	0.999
gcc	0.999	0.999	1.000	1.000	0.999	1.000	1.000	1.000
Wt. Avg.	0.999	0.998	0.999	0.999	0.999	0.999	0.999	0.999

Table I5-15b: Execution time speedups for partial-subblock TLBs relative to complete-subblock TLBs with same subblock factor, number of blocks and associativity (128-block fully-associative)

Workload	subblock factor				With preloading subblock factor			
	2	4	8	16	2	4	8	16
coral	0.999	0.999	0.998	0.994	0.999	0.998	0.997	0.994
nasa7	1.000	1.000	1.000	1.000	0.999	1.000	1.000	1.000
compress	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
fftpde	1.000	1.000	1.000	1.000	1.000	1.000	0.999	1.000
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	0.998	1.000	1.000	1.000	0.999	1.000	1.000	1.000
spice	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
pthor	1.000	0.999	0.999	0.992	0.999	0.999	0.999	0.998
ML	0.999	0.999	0.999	0.999	0.999	1.000	1.000	1.000
gcc	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Wt. Avg.	1.000	1.000	0.999	0.998	1.000	1.000	0.999	0.999

Table I5-15c: Execution time speedups for partial-subblock TLBs relative to complete-subblock TLBs with same subblock factor, number of blocks and associativity (256-block 4-way set-associative)

Workload	subblock factor				With preloading subblock factor			
	2	4	8	16	2	4	8	16
coral	1.000	0.999	0.997	0.986	1.000	0.998	0.996	0.994
nasa7	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
compress	1.000	1.000	1.000	0.964	1.000	1.000	1.000	0.964
fftpde	1.000	1.000	1.000	0.999	1.000	1.000	0.999	0.999
wave5	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
mp3d	1.000	1.000	1.000	0.999	1.000	1.000	1.000	1.000
spice	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
pthor	1.000	0.998	0.997	0.869	0.999	0.999	0.999	0.873
ML	1.000	0.999	0.997	0.993	1.000	0.999	0.997	0.994
gcc	1.000	1.000	0.998	0.984	1.000	1.000	0.998	0.986
Wt. Avg.	1.000	1.000	0.999	0.978	1.000	1.000	0.999	0.979

Appendix J: Tables with absolute number of TLB misses

This appendix shows the absolute number of TLB misses for each workload and TLB configuration that I consider in this thesis. All the numbers are rounded to the nearest thousand. These tables are useful for recalculating the execution times for different TLB miss penalty or recalculating the weights used in calculating the normalized speedups or comparing the number of TLB misses for different TLBs. Most of the tables group data for TLBs of similar type that differ in the number of TLB blocks. For example, Table J-1 lists the number of TLB misses for all fully-associative single-page-size TLBs.

Table J-1: Fully-associative single-page-size (4KB) TLBs

Workload	Number of fully-associative TLB blocks					
	64	128	162	256	304	512
coral	85975	63525	48969	36666	33085	22712
nasa7	152357	148312	126002	85896	77234	18
compress	21348	818	250	29	28	27
fftpde	11281	11280	117	110	108	107
wave5	14511	8652	1053	46	45	37
mp3d	4050	2396	1599	160	88	55
spice	41923	7442	3789	818	664	323
pthor	2581	2217	2117	1862	1692	961
ML	38424	19303	16119	11610	10240	7334
gcc	2441	449	304	60	43	34

Table J-2: Set-associative (4-way) single-page-size (4KB) TLBs

Workload	Number of 4-way set-associative TLB blocks		
	128	256	512
coral	59431	38980	24648
nasa7	212513	118016	695
compress	842	41	28
fftpde	19739	19348	19156
wave5	6179	48	38
mp3d	2401	331	56
spice	16241	1663	238
pthor	2284	1830	930
ML	24565	13259	7829
gcc	1012	101	37

Table J-3: 64-block fully-associative single-page-size (4KB) TLBs (varying replacement policy)

Workload	Replacement policy			
	GODS	Clock	Random	FIFO
coral	85975	87267	93883	97137
nasa7	152357	151797	157625	158655
compress	21348	22157	26608	25967
fftpde	11281	11279	10117	11998
wave5	14511	14629	11978	13969
mp3d	4050	4059	4908	4846
spice	41923	47624	75532	65518
pthor	2581	2617	2993	2898
ML	38424	39915	52502	49408
gcc	2441	2688	4167	3859

Table J-4: Fully-associative superpage TLBs (superpage sizes 8KB and 16KB)

Workload	#blocks in Superpage TLB (4KB/8KB)			#blocks in Superpage TLB (4KB/16KB)		
	64	128	256	64	128	256
coral	62829	40942	28911	45652	32244	21606
nasa7	85957	67749	43	33965	65	3
compress	709	25	22	76	25	24
fftpde	11241	11237	59	11221	11215	30
wave5	5687	27	21	17	14	12
mp3d	2814	221	34	696	26	9
spice	10504	560	181	1021	107	3
pthor	1834	1532	1106	1213	960	478
ML	22816	10023	4985	12534	4499	1797
gcc	649	70	29	147	28	25

Table J-5: Fully-associative superpage TLBs (superpage size 32KB)

Workload	#blocks in Superpage TLB (4KB/32KB)								
	62	64	123	128	156	247	256	293	494
coral	33434	32762	22505	21889	18718	10947	10361	8123	1397
nasa7	1457	601	4	4	3	3	3	3	3
compress	64	50	25	25	24	24	24	23	22
fftpde	11207	11206	11187	11197	17	11	10	7	5
wave5	11	11	9	9	9	9	9	8	6
mp3d	32	29	8	7	4	2	2	2	2
spice	103	98	5	5	3	3	3	3	3
pthor	801	790	508	481	403	20	18	14	7
ML	6941	6130	1734	1658	1196	578	516	413	105
gcc	75	62	25	25	24	23	23	23	23

Table J-6: Fully-associative superpage TLBs (superpage size 64KB)

Workload	#blocks in Superpage TLB (4KB/64KB)				
	64	128	154	256	290
coral	22119	9811	6744	1252	490
nasa7	22	3	3	3	3
compress	264	25	25	24	24
fftpde	5502	9	6	6	5
wave5	8	8	7	7	6
mp3d	8	2	2	2	2
spice	28	3	3	3	3
pthor	554	221	50	7	7
ML	2827	599	435	177	109
gcc	53	25	24	23	23

Table J-7: 256-block 4-way set-associative superpage TLBs

Workload	Superpage size in exact-index TLB				Superpage size in superpage-index TLB			
	8KB	16KB	32KB	64KB	8KB	16KB	32KB	64KB
coral	30555	22514	10620	1279	30542	22344	10999	4712
nasa7	964	3	3	3	739	13	756	3997
compress	25	25	24	25	24	26	1527	17982
fftpde	18677	8242	7004	12	19342	9188	8232	251
wave5	24	12	9	7	23	13	50	415
mp3d	38	7	2	2	44	11	7	15
spice	147	17	3	3	165	16	36	123
pthor	1080	416	94	15	1081	810	512	9303
ML	5879	2116	626	199	5935	2412	3750	7092
gcc	40	26	24	24	32	55	8215	20110

Table J-8: Fully-associative superpage TLBs (superpage sizes 32KB and 64KB) using base pages

Workload	#blocks in Superpage TLB (4KB/32KB) using base pages only						#blocks in Superpage TLB (4KB/64KB) using base pages only	
	62	123	156	247	293	494	154	290
coral	87200	65336	49851	37417	33801	23423	50141	34023
nasa7	152755	148807	130573	86991	80092	22	132180	78026
compress	22960	878	343	29	29	27	363	28
fftpde	11280	11268	116	113	108	107	119	108
wave5	14555	10044	1055	47	45	37	1056	45
mp3d	4105	2522	1733	211	93	56	1778	94
spice	47734	8595	4164	882	687	330	4357	687
pthor	2597	2237	2134	1890	1729	1066	2139	1737
ML	40222	20184	16904	11912	10629	7607	17129	10831
gcc	2636	483	327	67	46	34	335	47

Table J-9: 4-way Set-associative superpage TLBs with superpage index (superpage sizes 32KB and 64KB) using base pages only

Workload	256-block set-associative Superpage TLB using base pages only				512-block set-associative Superpage TLB using base pages only	
	8KB	16KB	32KB	64KB	32KB	64KB
coral	42299	52751	95075	98138	83438	86404
nasa7	69508	63955	166338	180153	162444	180155
compress	146	647	36221	66226	36245	65895
fftpde	19379	9295	8388	359	228	218
wave5	538	4088	33005	46577	31638	46187
mp3d	414	454	2675	4190	2630	3803
spice	6422	16441	83467	152467	82312	152458
pthor	1892	2419	2709	11959	2082	11737
ML	15356	21096	40136	59033	31804	53043
gcc	184	942	6517	25868	14941	33855

Table J-10: Fully-associative complete-subblock TLBs (subblock factor 2) - no preloading

Workload	#blocks in complete-subblock TLB (subblock factor 2)						
	51	64	102	128	206	256	413
coral	80450	75252	61506	45151	36343	32400	23145
nasa7	150306	148932	121493	113464	34296	18	4
compress	6072	1100	48	30	27	27	25
fftpde	11279	11279	11280	11278	111	109	103
wave5	13324	10013	412	46	40	37	33
mp3d	4068	3517	1565	282	71	58	40
spice	28796	14328	2880	1000	456	323	4
pthor	2553	2419	2182	2077	1754	1520	521
ML	37452	28114	17776	14368	9903	8377	5572
gcc	1738	785	237	88	36	34	34

Table J-11: Fully-associative complete-subblock TLBs (subblock factor 2) - with preloading

Workload	#blocks in complete-subblock TLB (subblock factor 2)						
	51	64	102	128	206	256	413
coral	66524	62715	52379	40847	32597	28834	20155
nasa7	86637	85870	71825	67591	17242	12	3
compress	3666	623	33	21	20	19	18
fftpde	11240	11239	11240	11237	61	59	55
wave5	7709	5653	308	26	23	21	19
mp3d	3328	2704	1012	183	45	34	21
spice	20723	9915	1713	530	241	172	3
pthor	1906	1789	1590	1505	1261	1088	369
ML	28904	21278	11906	9539	6040	4779	3104
gcc	1316	553	156	59	28	27	26

Table J-12: Fully-associative complete-subblock TLBs (subblock factor 4) - no preloading

Workload	#blocks in complete-subblock TLB (subblock factor 4)					
	35	64	72	128	147	297
coral	81637	65055	62427	40627	38306	25424
nasa7	137416	112547	99065	121	15	3
compress	1226	36	32	28	27	25
fftpde	11280	11279	11279	11274	111	95
wave5	8754	47	47	38	37	31
mp3d	4382	1205	183	63	56	3
spice	33337	2423	947	323	287	3
pthor	2620	2239	2183	1842	1710	337
ML	43003	20429	17891	9773	8728	3853
gcc	1336	125	77	35	34	34

Table J-13: Fully-associative complete-subblock TLBs (subblock factor 4) - with preloading

Workload	#blocks in complete-subblock TLB (subblock factor 4)					
	35	64	72	128	147	297
coral	53269	45247	43790	31893	29907	18769
nasa7	40876	33766	29579	38	6	3
compress	396	20	19	17	17	16
fftpde	11222	11219	11220	11214	35	28
wave5	2659	16	16	13	13	12
mp3d	2961	462	99	25	20	2
spice	19334	860	275	91	79	3
pthor	1427	1145	1110	908	840	158
ML	27766	9902	8257	3997	3126	1214
gcc	679	56	38	23	23	23

Table J-14: Fully-associative complete-subblock TLBs (subblock factor 8) - no preloading

Workload	#blocks in complete-subblock TLB (subblock factor 8)						
	20	32	44	64	92	128	188
coral	93477	76671	66249	51195	41272	35245	27260
nasa7	132846	108599	56674	216	10	3	3
compress	2002	812	38	29	27	27	26
fftpde	11282	11279	11280	11276	11264	11258	80
wave5	7061	139	46	39	35	31	30
mp3d	4933	3248	113	74	51	26	2
spice	111680	6715	870	396	58	3	3
pthor	3813	2504	2256	2053	1783	1273	652
ML	70097	34052	21579	13624	8570	5731	3184
gcc	5125	329	69	36	34	34	34

Table J-15: Fully-associative complete-subblock TLBs (subblock factor 8) - with preloading

Workload	#blocks in complete-subblock TLB (subblock factor 8)						
	20	32	44	64	92	128	188
coral	46897	41064	37284	31983	26161	21175	14900
nasa7	20281	16461	7367	34	4	3	2
compress	495	153	18	17	16	16	15
fftpde	11194	11188	11188	11184	11177	11173	14
wave5	1389	63	11	10	9	8	8
mp3d	3167	1086	61	24	10	5	2
spice	65097	2432	144	61	11	3	2
pthor	2153	968	818	710	598	423	213
ML	41532	17132	9273	3965	1965	1128	562
gcc	1241	104	29	22	21	21	20

Table J-16: Fully-associative complete-subblock TLBs (subblock factor 16) - no preloading

Workload	#blocks in complete-subblock TLB (subblock factor 16)							
	9	16	23	32	51	64	107	128
coral	111589	93184	78236	69193	47665	41637	30153	24706
nasa7	188088	110246	56837	10408	11	4	3	3
compress	38772	1296	534	35	29	28	27	26
fftpde	11287	11280	11278	11278	10821	11131	72	52
wave5	35009	3268	332	42	34	32	29	28
mp3d	7865	4853	991	91	50	39	2	2
spice	738567	117707	1327	563	63	4	3	3
pthor	12910	3984	2599	2265	1979	1738	937	445
ML	246857	71705	37760	22605	10598	7103	3081	2353
gcc	30591	1492	428	40	35	34	34	34

Table J-17: Fully-associative complete-subblock TLBs (subblock factor 16) - with preloading

Workload	#blocks in complete-subblock TLB (subblock factor 16)							
	9	16	23	32	51	64	107	128
coral	54348	40381	35397	31626	24829	21159	12104	8854
nasa7	66542	8637	3763	678	3	2	2	2
compress	10238	183	73	17	16	15	14	15
fftpde	5727	5710	5709	5704	5472	5629	9	7
wave5	13932	2157	140	8	7	7	7	6
mp3d	6001	2113	171	31	7	4	2	2
spice	491353	55210	167	48	21	2	2	2
pthor	9188	2005	769	574	458	390	208	97
ML	167066	38062	16398	7622	1892	1042	342	226
gcc	10106	561	41	21	20	20	19	20

Table J-18: 4-way set-associative complete-subblock TLBs (subblock factor 2)

Workload	without preloading			with preloading		
	128	256	512	128	256	512
coral	50232	34427	19946	43729	30453	17247
nasa7	138182	1368	3	101631	693	3
compress	74	28	26	32	21	19
fftpde	19760	19370	8608	19729	19337	8573
wave5	66	39	32	42	23	19
mp3d	638	62	23	419	37	12
spice	6041	283	11	4664	152	7
pthor	2100	1458	367	1533	1052	269
ML	17586	9222	4478	12535	5572	2475
gcc	197	38	34	136	29	27

Table J-19: 4-way set-associative complete-subblock TLBs (subblock factor 4)

Workload	without preloading				with preloading			
	64	128	256	512	64	128	256	512
coral	64365	44670	29555	13590	44683	33354	21984	9520
nasa7	141920	2664	3	3	78394	678	3	3
compress	167	30	27	25	70	18	16	16
fftpde	20507	19757	9239	8062	20461	19710	9184	8050
wave5	140	44	32	28	74	16	12	11
mp3d	1383	84	23	2	706	35	7	2
spice	15924	361	18	3	11735	108	7	3
pthor	2370	1824	744	94	1259	925	368	42
ML	27816	11767	5101	1527	15268	4733	1696	466
gcc	571	40	34	34	165	25	23	23

Table J-20: 4-way set-associative complete-subblock TLBs (subblock factor 8) - no preloading

Workload	#blocks in complete-subblock TLB (subblock factor 8)				
	32	64	128	256	512
coral	78790	57433	37893	20058	1941
nasa7	144691	5176	4	3	3
compress	651	32	28	26	25
fftpde	21976	20506	10421	8232	5
wave5	347	65	33	28	6
mp3d	2780	147	26	2	2
spice	47427	477	28	3	3
pthor	3381	2197	1254	247	9
ML	49542	16829	6424	1872	181
gcc	1453	120	35	34	34

Table J-21: 4-way set-associative complete-subblock TLBs (subblock factor 8) - with preloading

Workload	#blocks in complete-subblock TLB (subblock factor 8)				
	32	64	128	256	512
coral	42302	32641	21906	9990	819
nasa7	65575	659	3	2	2
compress	171	22	15	14	14
fftpde	21702	20331	10261	8190	4
wave5	181	21	9	8	5
mp3d	1243	47	5	2	2
spice	31677	88	6	2	2
pthor	1708	840	427	72	6
ML	29013	6416	1490	339	41
gcc	775	23	21	21	21

Table J-22: 4-way set-associative complete-subblock TLBs (subblock factor 16) - no preloading

Workload	#blocks in complete-subblock TLB (subblock factor 16)					
	16	32	64	128	256	512
coral	95012	73534	49184	25367	2875	7
nasa7	147647	10361	4	3	3	3
compress	1441	59	29	27	25	25
fftpde	24820	21770	11367	8550	5	5
wave5	12930	638	36	28	6	6
mp3d	4771	498	31	2	2	2
spice	222483	4812	101	3	3	3
pthor	6061	2797	1706	465	11	6
ML	111782	30352	8960	2373	252	19
gcc	4187	232	62	34	34	34

Table J-23: 4-way set-associative complete-subblock TLBs (subblock factor 16) - with preloading

Workload	#blocks in complete-subblock TLB (subblock factor 16)				
	16	32	128	256	512
coral	43729	32581	9228	682	6
nasa7	59724	666	2	2	2
compress	203	24	14	14	14
fftpde	19585	16192	8475	4	4
wave5	7003	367	7	5	5
mp3d	2502	148	2	2	2
spice	138139	1919	2	2	2
pthor	3484	994	89	6	5
ML	67032	12683	275	36	15
gcc	2265	24	20	20	20

Table J-24: Fully-associative partial-subblock TLBs (subblock factor 2) - no preloading

Workload	#blocks in partial-subblock TLB (subblock factor 2)					
	64	127	128	255	256	509
coral	75333	45389	45232	32527	32447	18846
nasa7	148958	114110	113542	71	66	3
compress	1148	32	31	28	27	25
fftpde	11279	11260	11277	109	109	101
wave5	10057	47	47	37	36	31
mp3d	3620	381	345	58	58	27
spice	14336	1041	1003	346	338	3
pthor	2449	2096	2092	1538	1531	126
ML	29427	14909	15008	8395	8383	4480
gcc	889	102	99	35	35	34

Table J-25: Fully-associative partial-subblock TLBs (subblock factor 2) - with preloading

Workload	#blocks in partial-subblock TLB (subblock factor 2)					
	64	127	128	255	256	509
coral	62822	41088	40942	28983	28910	16277
nasa7	85910	67886	67749	39	40	3
compress	673	27	25	23	22	22
fftpde	11240	11221	11237	59	59	54
wave5	5688	27	27	22	22	18
mp3d	2763	241	220	34	34	15
spice	9924	557	533	180	177	3
pthor	1833	1533	1532	1108	1105	89
ML	22314	9844	10006	4962	4949	2462
gcc	648	71	69	29	29	28

Table J-26: Fully-associative partial-subblock TLBs (subblock factor 4) - no preloading

Workload	#blocks in partial-subblock TLB (subblock factor 4)						
	63	64	126	128	252	256	504
coral	65552	65287	41105	40810	28833	28503	13597
nasa7	114361	112620	162	156	4	3	3
compress	111	90	29	29	27	27	25
fftpde	11280	11279	11276	11274	102	100	35
wave5	48	48	38	38	32	31	29
mp3d	1878	1714	66	65	28	27	2
spice	2712	2467	367	344	3	3	3
pthor	2281	2271	1898	1882	971	955	38
ML	22708	21850	11057	10709	5340	5199	1734
gcc	248	236	37	37	34	34	34

Table J-27: Fully-associative partial-subblock TLBs (subblock factor 4) - with preloading

Workload	#blocks in partial-subblock TLB (subblock factor 4)						
	63	64	126	128	252	256	504
coral	45661	45455	32344	32101	21677	21468	9554
nasa7	34344	33827	53	50	3	3	3
compress	68	56	23	23	23	23	21
fftpde	11221	11220	11216	11214	30	30	12
wave5	17	17	14	14	12	12	11
mp3d	775	694	27	26	9	8	2
spice	1010	878	104	101	3	3	3
pthor	1205	1197	960	951	482	474	17
ML	12220	11713	4470	4315	1759	1702	569
gcc	112	107	26	26	25	25	25

Table J-28: Fully-associative partial-subblock TLBs (subblock factor 8) - no preloading

Workload	#blocks in partial-subblock TLB (subblock factor 8)							
	32	61	64	122	128	244	256	489
coral	77163	55157	51814	36422	35552	21105	19804	2938
nasa7	110512	14460	304	7	3	3	3	3
compress	1191	34	32	28	28	27	26	25
fftpde	11280	11279	11276	11266	11265	54	45	7
wave5	296	40	40	33	32	28	28	7
mp3d	3870	84	79	38	29	2	2	2
spice	7959	540	448	7	4	3	3	3
pthor	2639	2131	2103	1443	1340	55	49	12
ML	42640	17351	15904	6929	6611	2627	2367	485
gcc	1348	73	61	35	35	34	34	34

Table J-29: Fully-associative partial-subblock TLBs (subblock factor 8) - with preloading

Workload	#blocks in partial-subblock TLB (subblock factor 8)							
	32	61	64	122	128	244	256	489
coral	41546	33345	32346	22267	21540	10834	10031	1326
nasa7	17055	1436	56	3	3	3	2	2
compress	305	24	23	22	22	22	22	21
fftpde	11208	11207	11204	11198	11197	11	10	5
wave5	173	10	10	9	9	8	8	6
mp3d	1549	31	27	6	5	2	2	2
spice	3055	87	73	3	3	3	3	3
pthor	1105	776	746	490	456	18	16	7
ML	25264	6754	5975	1681	1520	530	468	99
gcc	665	35	30	23	22	22	22	22

Table J-30: Fully-associative partial-subblock TLBs (subblock factor 16) - no preloading

Workload	#blocks in partial-subblock TLB (subblock factor 16)								
	16	32	57	64	114	128	228	256	456
coral	94424	70067	45581	42647	28983	25434	6630	3987	7
nasa7	114520	10549	15	9	3	3	3	3	3
compress	2775	160	31	30	27	27	27	26	25
fftpde	11281	11279	10776	10864	69	46	18	5	5
wave5	3503	89	35	33	29	28	21	7	6
mp3d	5359	108	51	46	3	2	2	2	2
spice	148595	662	80	18	3	3	3	3	3
pthor	8646	2420	2000	1879	960	812	20	15	7
ML	97375	32979	12293	10196	4199	3497	1188	882	117
gcc	13895	917	64	46	34	34	34	34	34

Table J-31: Fully-associative partial-subblock TLBs (subblock factor 16) - with preloading

Workload	#blocks in partial-subblock TLB (subblock factor 16)								
	16	32	57	64	114	128	228	256	456
coral	41776	32260	23650	21687	11495	9407	1815	1030	6
nasa7	9602	708	4	3	2	2	2	2	2
compress	671	120	22	22	22	22	21	21	20
fftpde	5731	5724	5465	5515	9	8	5	5	4
wave5	2308	35	7	7	7	7	6	6	5
mp3d	3349	49	7	6	2	2	2	2	2
spice	73293	66	8	3	3	2	2	2	2
pthor	4448	697	494	450	220	183	7	7	6
ML	60194	15522	3201	2372	629	524	194	120	24
gcc	9942	536	34	27	22	22	22	22	22

Table J-32: 4-way set-associative partial-subblock TLBs (subblock factor 2)

Workload	without preloading			with preloading		
	128	256	512	128	256	512
coral	50363	34493	20002	43879	30494	17327
nasa7	138202	1426	4	101755	738	3
compress	89	30	27	68	26	23
fftpde	19766	19372	8610	19735	19340	8575
wave5	88	39	32	62	23	19
mp3d	704	72	23	531	44	13
spice	6069	294	12	4682	160	8
pthor	2123	1479	386	1579	1080	291
ML	18655	9562	4505	13363	5982	2641
gcc	333	39	34	256	31	28

Table J-33: 4-way set-associative partial-subblock TLBs (subblock factor 4)

Workload	without preloading				with preloading			
	64	128	256	512	64	128	256	512
coral	64672	44870	29698	13761	45046	33577	22181	9697
nasa7	141992	2835	25	4	78502	763	12	3
compress	289	42	30	28	148	31	23	23
fftpde	20520	19764	9241	8064	20473	19717	9187	8051
wave5	199	50	33	28	120	21	13	11
mp3d	1701	255	29	5	985	138	11	3
spice	15975	386	20	3	11768	121	8	3
pthor	2787	1979	812	161	1649	1068	419	78
ML	32511	13614	5933	2036	21883	6883	2388	774
gcc	3871	86	36	34	3361	53	26	25

Table J-34: 4-way set-associative partial-subblock TLBs (subblock factor 8) - no preloading

Workload	#blocks in partial-subblock TLB (subblock factor 8)				
	32	64	128	256	512
coral	79408	57915	38801	20436	2411
nasa7	145137	5525	56	5	3
compress	1065	228	148	32	33
fftpde	24612	23115	12743	8245	48
wave5	600	94	35	29	9
mp3d	3351	462	48	10	2
spice	89489	2247	56	4	3
pthor	8937	2474	1351	368	65
ML	67709	27004	10297	4991	2253
gcc	10739	1272	325	309	313

Table J-35: 4-way set-associative partial-subblock TLBs (subblock factor 8) - with preloading

Workload	#blocks in partial-subblock TLB (subblock factor 8)				
	32	64	128	256	512
coral	43016	33179	22404	10384	1141
nasa7	65938	770	15	3	3
compress	389	92	91	24	25
fftpde	24554	23057	12673	8228	46
wave5	346	34	10	8	6
mp3d	1733	198	13	5	2
spice	65747	1003	13	3	3
pthor	6910	1089	505	130	25
ML	45703	14228	5356	3208	1823
gcc	8855	833	307	240	307

Table J-36: 4-way set-associative partial-subblock TLBs (subblock factor 16) - no preloading

Workload	#blocks in partial-subblock TLB (subblock factor 16)					
	16	32	64	128	256	512
coral	95154	73894	50608	26164	4167	596
nasa7	183738	11146	109	52	22	22
compress	5333	4775	3127	3113	2943	3100
fftpde	27451	24409	13925	8575	49	5
wave5	12868	801	90	32	10	7
mp3d	9868	2372	238	92	26	6
spice	234301	44963	134	15	5	5
pthor	28786	15016	10006	7790	6869	6773
ML	163489	60306	22242	11592	6951	6186
gcc	27639	6010	3366	2983	1874	2992

Table J-37: 4-way set-associative partial-subblock TLBs (subblock factor 16) - with preloading

Workload	#blocks in partial-subblock TLB (subblock factor 16)					
	16	32	64	128	256	512
coral	44323	33785	22539	9908	1283	257
nasa7	87826	899	32	26	21	21
compress	4304	3721	2950	2939	2946	2783
fftpde	22636	19070	12965	8549	46	5
wave5	8117	525	34	9	7	6
mp3d	8289	1602	105	40	8	5
spice	220069	30277	28	9	4	4
pthor	25326	12391	8390	7101	6607	6556
ML	122028	37577	12966	8887	5487	6142
gcc	17776	4724	3365	3009	1633	2982

Table J-38: Partial-subblock TLBs with preloading and no OS support

Workload	64-block fully-associative				256-block 4-way set-associative			
	2	4	8	16	2	4	8	16
coral	85987	85987	85987	85955	42300	52440	92215	98035
nasa7	152043	151558	152377	152379	68763	62763	165964	180172
compress	21403	21401	21382	21406	71	603	36038	65789
fftpde	11282	11282	11282	11282	19379	9295	8389	359
wave5	14491	14513	14531	14469	619	4158	33007	46559
mp3d	4035	4035	4051	4051	414	454	2651	4189
spice	41936	41894	41960	41941	6434	16406	83435	152477
pthor	2581	2581	2577	2580	1893	2420	2709	11962
ML	38337	38767	38494	38110	15338	21422	40055	58018
gcc	2441	2439	2440	2440	184	936	14911	33791

Table J-39: Fully-associative partial-subblock TLBs without preloading and no OS support (subblock factors 2 and 4)

Workload	#blocks in partial-subblock TLB subblock factor 2				#blocks in partial-subblock TLB subblock factor 4			
	64	127	255	509	63	126	252	504
coral	86095	63997	36747	22834	86575	64366	36933	23016
nasa7	152155	148193	86257	18	152094	148201	86701	18
compress	21347	853	29	27	22171	877	29	27
fftpde	11281	11281	111	107	11280	11275	111	107
wave5	14511	9029	47	37	14536	9423	47	37
mp3d	4050	2423	165	55	4078	2448	179	55
spice	43589	7772	830	321	45696	7981	833	334
pthor	2581	2222	1866	981	2588	2227	1874	1013
ML	38531	19425	11548	7418	38884	19777	11741	7496
gcc	2443	452	61	34	2536	456	63	34

Table J-40: Fully-associative partial-subblock TLBs without preloading and no OS support (subblock factors 8 and 16)

Workload	#blocks in partial-subblock TLB subblock factor 8				#blocks in partial-subblock TLB subblock factor 16			
	61	122	244	489	57	114	228	456
coral	87753	65462	37830	23631	88757	66501	39157	25003
nasa7	153111	148734	87764	435	176535	149646	90532	15576
compress	23754	916	29	27	26953	1066	30	27
fftpde	11281	11278	114	107	11280	11279	118	107
wave5	14576	10216	47	37	14793	10711	47	38
mp3d	4133	2546	233	57	4242	2742	393	59
spice	49629	8627	880	337	59453	10372	1194	369
pthor	2606	2240	1899	1088	2647	2272	1943	1260
ML	40810	20113	12047	7574	44629	21625	12939	7928
gcc	2750	489	70	34	3187	552	85	35

Bibliography

[Abra81] D. Abramson. Hardware Management of a Large Virtual Memory. In *Proc. of the 4th Australian Computer Science Conference*, 1981.

[Acce86] M. Accetta, Robert V. Baron, William Bolosky, David B. Golub, and Richard F. Rashid. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of the Summer USENIX Conference*, Atlanta, Summer 1986.

[Adva93] Advanced RISC Machines. ARM610 RISC Processor, 1993. Document #: ARM DDI 0004C.

[Agar88] A. Agarwal, M. Horowitz, and J. Hennessy. Cache Performance of Operating Systems and Multiprogramming Workloads. *ACM Trans. on Computer Systems*, 6(4):393–431, November 1988.

[Alex85] C. A. Alexander, W. M. Keshlear, and F. Briggs. Translation Buffer Performance in a UNIX Environment. *Computer Architecture News*, pages 2–14, December 1985.

[Alex86] C. Alexander, W. Keshlear, F. Cooper, and F. Briggs. Cache Memory Performance in a UNIX Environment. *Computer Architecture News*, 14(3):14–70, June 1986.

[Ande92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[Appe91a] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–106, Palo Alto, April 1991.

[Appe91b] Andrew W. Appel and David B. McQueen. Standard ML of New Jersey. In *Proc. Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, August 1991.

[Aspr93] Tom Asprey, Gregory S. Averill, Eric DeLano, Russ Mason, Bill Weiner, and Jeff Yetter. Performance Features of the PA7100 Microprocessor. *IEEE Micro*, 13(3):22–35, June 1993.

[Aust95] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi. Streamlining Data Cache Access with Fast Address Calculation. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 369–380, June 1995.

[Bach86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[Baer88] Jean-Loup Baer and Wen-Hann Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *Proc. of the 15th Annual International Symposium on Computer Architecture*, pages 73–80, Honolulu Hawaii, June 1988.

[Bail91] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. *Intl. Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[Bala92] Ramesh Balan and Kurt Gollhardt. A Scalable Implementation of Virtual Memory HAT layer for Shared Memory Multiprocessor. In *Proc. of the Summer USENIX Conference*, pages 107–116, June 1992.

[Bala94] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proc. First Symposium on Operating System Design and Implementation (OSDI)*, pages 243–253, Monterey, CA, November 1994.

[Bann95] Peter Bannon and Jim Keller. Internal Architecture of Alpha 21164 Microprocessor. *Compcon Digest of Papers*, pages 79–87, March 1995.

[Bark89] R. E. Barkley and T. Paul Lee. A Lazy Buddy System Bounded by Two Coalescing Delays per Class. In *Proc. of the 12th Symposium on Operating System Principles*, pages 167–176, December 1989.

[Barr93] David A. Barrett and Benjamin G. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 187–196, June 1993.

[Baye77] R. Bayer and M. Schkolnick. Concurrency of Operations on Btrees. *Acta Informatica*, 9(1), 1977. Also published as IBM, San Jose Research Lab, Research Report RJ 1791, May 1976.

[Beck93] Michael C. Becker, Michael S. Allen, Charles R. Moore, John S. Muhich, and David P. Tuttle. The PowerPC 601 Microprocessor. *IEEE Micro*, 13(5):54–68, October 1993.

[Bela66] L. A. Belady. A Study of Replacement Algorithms for a Virtual Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[Bell74] J. Bell, D. Casasent, and C. G. Bell. An investigation of Alternative Cache Organizations. *IEEE Trans. on Computers*, C-23(4):346–351, April 1974.

[Blac89] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron. Translation Lookaside Buffer Consistency: A Software Approach. In *Proc. of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 113–122, Boston, April 1989.

[Blan92] Greg Blanck and Steve Krueger. The SuperSPARC Microprocessor. *Compcon Digest of Papers*, pages 136–141, February 1992.

[Blum94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathon Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. 21st Annual Symposium on Computer Architecture, Computer Architecture N ews*, pages 142–153, April 1994.

[Bur61] *A definition of the B5000 Information Processing System*. Burrough Corp, 1961.

[Camp91] M. Campbell and et al. The Parallelization of UNIX System V Release 4.0. In *Proceedings of the Winter 1991 USENIX Conference*, 1991.

[Cao94] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *Proc. First Symposium on Operating System Design and Implementation (OSDI)*, page 165 177, Monterey, CA, November 1994.

[Cart94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-Based Addressing. In *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 328–337, October 1994.

[Chan88] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Trans. on Computer Systems*, 6(1):28–50, February 1988.

[Chan90] A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter. Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors. *IBM Journal of Research and Development*, 34(1):105–110, January 1990.

[Chan95] David Chih-Wei Chang and et al. Microarchitecture of HaL's Memory Management Unit. *Compcon Digest of Papers*, pages 272–279, March 1995.

[Chas94] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[Chen92] J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A Simulation Based Study of TLB Performance. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 114–123, May 1992.

[Chen93a] J. Bradley Chen. Software Methods for System Address Tracing. In *Proc. of the Fourth Workshop on Workstation Operating Systems*, pages 178–185, Napa CA, October 1993.

[Chen93b] J. Bradley Chen and Brian N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proc. of the 14th Symposium on Operating System Principles*, pages 120–133, December 1993.

[Chiu92] Tzicker Chiueh and Randy H. Katz. Eliminating the Address Translation Bottleneck for Physical Address Cache. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 137–148, Boston MA, October 1992.

[Clar85] 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):31–62, February 1985.

[Clar95] Ron Clark, Jack O' Quinn, and Tom Weaver. Symmetric Multiprocessing for the AIX Operating System. *Compcon Digest of Papers*, pages 110–115, March 1995.

[Come79] D. Comer. The Ubiquitous Btree. *ACM Surveys*, 11(2), June 1979.

[Cust93] Helen Custer. *Inside Windows NT*. Microsoft Press, 1993.

[Dall92] William J. Dally. A Fast Translation Method for Paging on top of Segmentation. *IEEE Transactions on Computers*, 41(2), February 1992.

[Dekk87] G. J. Dekker and A. J. van de Goor. AMORE, Address Mapping with Overlapped Rotating Entries. *IEEE Micro*, 7(3):22–34, June 1987.

[DeMo86] M. DeMoney, J. Moore, and J. Mashey. Operating System Support on a RISC. In *Proceedings 1986 COMPCON*, San Francisco, CA, March 4-6 1986. IEEE.

[Denn68] Peter J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[Denn70] Peter J. Denning. Virtual Memory. *Computing Surveys*, 2(3):153–189, September 1970.

[Denn75] Peter J. Denning and Kevin Kahn. A Study of Program Locality & Lifetime Functions. In *Proc. of the 5th Symposium on Operating System Principles*, pages 207–216, November 1975.

[Devi92] Yannick Deville and Jean Gobert. A class of replacement policies for medium and high associativity structures. *Computer Architecture News*, 20(1):55–64, March 1992.

[Dubn92] Czarek Dubnicki and Thomas J. LeBlanc. Adjustable Block Size Coherent Caches. In *Proc. 19th Annual International Symposium on Computer Architecture*, May 1992.

[East79] M. C. Easton and P. A. Franasek. Use of Bit Scanning in Replacement Decisions. *IEEE Transactions on Computing*, 28(2):133–141, February 1979.

[Eden90] Robin W. Edenfield, Michael G. Gallup, William B. Ledbetter, Jr., Ralph C. McGarity, Eric E. Quintana, and Russell A. Reininger. The 68040 Processor: Part 2, Memory Design & Chip Verification. *IEEE Micro*, 10(3):22–35, June 1990.

[Elli87] Carla S. Ellis. Concurrency in Linear Hashing. *ACM Transactions on Database Systems*, 12(2), June 1987. Also published as ACM SIGACT-SIGMOD Symposium on Principles of Database Systems 4, Mar.1985.

[ETA 86] ETA Systems, Inc. Mainframe Subsystem Instruction Specification for the ETA10, Rev: B, March 1986.

[Eykh92] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond Multiprocessing: Multithreading the SunOS Kernel. In *Proc. of the Summer USENIX Conference*, pages 11–18, June 1992.

[Fabr74] R. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[Fagi79] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible Hashing — A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3), September 1979. Also published as IBM Research Report RJ2305, July 1978.

[Fran74] Mark A. Franklin, G. Scott Graham, and R. K. Gupta. Anomalies with Variable Partition Paging Algorithms. *Communications of the ACM*, 21(3):232–236, March 1974.

[Gels89] P. P. Gelsinger, P. A. Gargini, G. H. Parker, and A. Y. C. Yu. Microprocessors circa 2000. *IEEE Spectrum*, 26(10):43–47, October 1989.

[Ging87a] Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mary S. Weeks. Shared Libraries in SunOS. In *Proc. of the Summer USENIX Conference*, pages 81–94, Phoenix, Summer 1987.

[Ging87b] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual Memory Architecture in SunOS. In *Proc. of the Summer USENIX Conference*, pages 131–146, Phoenix, Summer 1987.

[Glas65] E. L. Glaser, J. F. Couleur, and G. A. Oliver. System Design of a computer for time sharing applications. In *Proc. of AFIPS*, volume 27, pages 197–202, 1965.

[Good83] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proc. of the Tenth Annual International Symposium on Computer Architecture*, pages 124–131, Stockholm Sweden, June 1983.

[Gutt84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOD Conference*, page 47, Boston, MA, June 1984. Reprinted in M. Stonebraker, Readings in Database Systems, Morgan Kaufmann, San Mateo, CA, 1988.

[Hart92] Kieran Harty and David R. Cheriton. Application-Controlled Physical Memory using Extern Page-Cache Management. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, Boston MA Boston MA, October 1992.

[Henn90] John L Hennessy and David A Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[Hewl93] Hewlett Packard. Hewlett-Packard's 7100: A High-speed Superscalar PA-RISC Processor, 1993. White paper.

[Hill84] Mark D. Hill and Alan Jay Smith. Experimental Evaluation of On-Chip Microprocessor Cache Memories. In *Proc. of the 11th Annual International Symposium on Computer Architecture*, pages 158–166, Ann Arbor MI, June 1984.

[Hill86] Mark D. Hill, Susan J. Eggers, James R. Larus, George S. Taylor, G. Adams, B. K. Bose, Garth A. Gibson, P. M. Hansen, J. Keller, Shing I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, David A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, Randy H. Katz, John Ousterhout, and David A. Patterson. Design Decisions in SPUR. *IEEE Computer*, 19(11):8–22, November 1986.

[Hill87] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. Ph.D. thesis, University of California, Berkeley, November 1987.

[Hill88] Mark D. Hill. A Case for Direct-Mapped Caches. *IEEE Computer*, 21(12):25–40, December 1988. Also available as Computer Sciences Technical Report #778, Univ. of Wisconsin, June 1988.

[Hill89] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *IEEE Trans. on Computers*, C-38(12):1612–1630, December 1989.

[Hirs73] Daniel S. Hirschberg. A Class of dynamic Memory Allocation Algorithms. *Communications of the ACM*, 16(10):615–618, October 1973.

[Houd68] M. E. Houdek and G. R. Mitchell. Translating a large virtual address. *IBM System/38 Tech. Developments*, pages 22–24, 1968.

[Hsu86] M-C. Hsu and M-P. Yang. Concurrent Operations in Extendible Hashing. In *Proceedings of the 12th Conference on Very Large Databases*, August 1986.

[Huck93] Jerry Huck and Jim Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 39–50, May 1993.

[Hunt95] Doug Hunt. Advanced Performance Features of the 64-bit PA-8000. *Compcon Digest of Papers*, pages 123–115, March 1995.

[IBM78] *IBM System/38 technical developments*. IBM, 1978. Order no G580-0237.

[Inte91] Intel Corporation. i860 Microprocessor Family Programmer's Reference Manual, 1991.

[John61] L. R. Johnson. Indirect chaining method for addressing on secondary keys. *Communications of the ACM*, pages 218–222, May 1961.

[John87] Mike Johnson. System Consideration in the Design of the Am29000. *IEEE Micro*, 7(4):28–41, August 1987.

[Joup89] Norman P. Jouppi and David W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proc. of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, April 1989.

[Joup94] Norman P. Jouppi and Steven J. E. Wilson. Tradeoffs in Two-Level On-Chip Caching. In *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 34–45, April 1994. (Also as) WRL Research Report 93/3.

[Kagi91] Toyohiko Kagimasa, Kikuo Takahashi, and Toshiaki Mori. Adaptive Storage Management for Very Large Virtual/Real Storage Systems. In *Proc. of the 18th Annual International Symposium on Computer Architecture*, pages 372–379, May 1991.

[Kane89] Gerry Kane. *MIPS RISC Architecture*. Prentice Hall, 1989.

[Kane92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[Karl88] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive Snoop Caching. *Algorithmica*, 3(1):70–119, 1988.

[Karl91] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for Shared Memory Multiprocessors. In *Proc. of the Thirteenth ACM Symposium on Operating System Principles*, 1991.

[Kess89] R. E. Kessler and Miron Livny. An Analysis of Distributed Shared Memory Algorithms. Computer Sciences Technical Report #825, Univ. of Wisconsin, February 1989.

[Kess91] Richard Eugene Kessler. Analysis of Multi-Megabyte Secondary CPU Cache Memories. Computer Sciences Technical Report #1032, Univ. of Wisconsin, July 1991.

[Kess92] R. E. Kessler and Mark D. Hill. Page Placement Algorithms for Large Real-Index Caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.

[Khal93a] Yousef A. Khalidi, Glen R. Anderson, Stephen A. Chessin, Shing Ip Kong, Charles E. Narad, and Madhusudhan Talluri. Virtual Address To Physical Address Translation Cache that Supports Multiple Page Sizes. Patent application filed, Serial No. 08/118,398, Sun Microsystems, September 1993. (Accepted March 1995).

[Khal93b] Yousef A. Khalidi, Madhusudhan Talluri, Michael N. Nelson, and Dock Williams. Virtual Memory Support for Multiple Page Sizes. In *Proc. of the Fourth Workshop on Workstation Operating Systems*, pages 104–109, Napa CA, October 1993.

[Khal94] Yousef A. Khalidi, Vikram P. Joshi, and Dock Williams. A Study of the Structure and Performance of MMU Handling Software. Technical Report TR-94-28, Sun Microsystems Laboratories, 1994.

[Khal95a] Yousef Khalidi, Vikram Joshi, Madhusudhan Talluri, Adrian Caceras, and Dock Williams. De-

sign Rationale of the UltraSPARC Hardware Address Translation Layer. In *SunSoft Technical Conference*, April 1995.

[Khal95b] Yousef A. Khalidi and Madhusudhan Talluri. Improving the Address Translation Performance of Widely Shared Pages. Technical Report TR-95-38, Sun Microsystems Laboratories, February 1995.

[Kim91] Yul H. Kim, Mark D. Hill, and David A. Wood. Implementing Stack Simulation for Highly-Associative Memories. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 212–213, May 1991. Also available as University of Wisconsin-Madsion, Computer Sciences Technical Report #997.

[Klei86] Steve R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proc. of the Summer USENIX Conference*, Atlanta, Summer 1986.

[Knig81] J. Knight and P. Rosenfield. Segmented Virtual to Real Translation Assist. *IBM Technical Disclosure Bulletin*, 23(11):5186–5187, April 1981.

[Know65] Kenneth C. Knowlton. A Fast Storage Allocator. *Communications of the ACM*, 8(10):623–625, October 1965.

[Knut68a] Donald E. Knuth. *The Art of Computer Programming, Volume 1*. Addison Wesley, 1968. Second Printing.

[Knut68b] Donald E. Knuth. *The Art of Computer Programming, Volume 3*. Addison Wesley, 1968. Second Printing.

[Koga88] M. S. Kogan and F. L. Rawson, III. The design of Operating System/2. *IBM Systems Journal*, 27(2):90–104, 1988.

[Kold92] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural Support for Single Address Space Operating Systems. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–186, Boston MA, October 1992.

[Kong92] Shing Kong. *Sparc V9 Memory Management Unit Design Rationale*. Sun Microsystems Inc., October 1992.

[Kuma90] Vijay Kumar. Concurrent Operations on Extendible Hashing and its Performance. *Communications of the ACM, ; ACM CR 9012-0959*, 33(6), June 1990.

[Lebe95] Alvin R. Lebeck and David A. Wood. Active Memory: A New Abstraction for Memory-System Simulation. In *Proc. of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1995.

[Lee69] Francis F. Lee. Study of "Look-Aside" Memory. *IEEE Transactions on Computers*, 18(11):1062–1064, November 1969.

[Lee89a] David D. Lee, Shing I. Kong, Mark D. Hill, George S. Taylor, David A. Hodges, Randy H. Katz, and David A. Patterson. VLSI chip set for a multiprocessor workstation - Part I: An RISC microprocessor with coprocessor interface and support for symbolic processing. *IEEE Journal of Solid-State Circuits*, pages 1688–1698, December 1989.

[Lee89b] Ruby B. Lee. Precision Architecture. *IEEE Computer*, 22(1):78–91, January 1989.

[Lee89c] T. Paul Lee and Ronald E. Barkley. A Watermark-Based Lazy Buddy System for Kernel Memory Allocation. In *Proc. Summer 89 USENIX Conference*, pages 1–14, June 1989.

[Leff90] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1990.

[Leon82] T. Leonard, editor. *VAX-11 Architecture Reference Manual*. Digital Press, May 1982. Revision 6.1.

[Levi95] David Levitan, Thomas Thomas, and Paul Tu. The PowerPC 620 Microprocessor: A High-Performance Superscalar RISC Processor. *Compcon Digest of Papers*, pages 285–291, March 1995.

[Levy82] H. M. Levy and P. H. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, 15(3):35–41, March 1982.

[Lied95] Jochen Liedtke. Address Space Sparsity and Fine Granularity. *Operating Systems Review*, 29(1):87–90, January 1995.

[Lipt68] J. S. Liptay. Structural aspects of the System/360 Model 85, Part II: the cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[Litw93] W. Litwin, M. Neimat, and D. Schneider. LH -Linear Hashing for Distributed Files. *19 ACM SIGMOD Conf. on the Management of Data*, May 1993.

[Mack94] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and Frans Kaashoek. FUGU: Implementing Translation and Protection in a Multiuser, Multimodel Multiprocessor. Technical Memo MIT/LCS/TM-503, October 1994.

[Matt70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[May94] Cathay May, Ed Silha, Rick Simpson, and Hank Warren. *The PowerPC Architecture*. Morgan Kaufman Publishers, May 1994.

[McKu84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):191–197, August 1984.

[Mile90] Milan Milenkovic. Microprocessor Memory Management Units. *IEEE Micro*, 10(2):70–85, April 1990.

[MIPS93] MIPS Technologies, Inc. TFP Microprocessor Chip Set: Preliminary Product Information, October 1993.

[Mogu93] Jeffrey C. Mogul. Big Memories on the Desktop. In *Proc. of the Fourth Workshop on Workstation Operating Systems*, pages 110–115, Napa CA, October 1993.

[Mogu95] Jeffrey Mogul, Joel Bartlett, Robert Mayo, and Amitabh Srivastava. Performance Implications of Multiple Pointer Sizes. *USENIX*, 1995.

[Mora88] Joseph P. Moran. SunOS Virtual Memory Implementation. In *Proc. of Europoean UNIX Users Group Conference*, Spring 1988.

[Morr68] R. Morris. Scatter Storage Techniques. *Communications of the ACM*, 11(1):38–43, January 1968.

[Moto86] Motorola Inc. MC68851 Paged Memory Management Unit User's Manual, 1986.

[Muld91] Johannes M. Mulder, Nhon T. Quach, and Michael J. Flynn. An Area Model for On-Chip Memories and its Applications. *IEEE Journal of Solid State Circuits*, 26(2):98–106, February 1991.

[Nagl92] David Nagle, Richard Uhlig, and Trevor Mudge. Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architecture. University of michigan technical report, University of Michigan, May 1992.

[Nagl94a] David Nagle, Richard Uhlig, Trevor Mudge, and Stuart Sechrest. Optimal Allocation of On-Chip Memory for Multiple-API Operating Systems. In *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 358–369, April 1994.

[Nagl94b] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design Tradeoffs for Software-Managed TLBs. *ACM Trans. on Computer Systems*, 12(3):175–205, August 1994.

[Ogde95] Deene Ogden, Belli Kuttanna, Albert J. Loper, Soummya Mallick, and Michael Putrino. A New PowerPC Microprocessor for Low Power Computing Systems. *Compcon Digest of Papers*, pages

281–284, March 1995.

[Orga72] E.J. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, 1972.

[Pete77] J. L. Peterson and N. Theodore. Buddy Systems. *Communications of the ACM*, 20(6):421–431, June 1977.

[Prie76] B. G. Prieve and R. S. Fabry. VMIN- AN Optimal Variable Space Page Replacement algorithm. *Communications of the ACM*, 19(6):295–297, May 1976.

[Purd70] P. W. Purdom and S. M. Stigler. Statistical Properties of the Buddy System. *JACM*, 17(4):683–697, October 1970.

[Puza85] T. R. Puzak. *Analysis of Cache Replacement Algorithms*. Ph.D. dissertation, Dept. of Electrical and Computer Engineering, University of Massachusetts, February 1985.

[Radi82] G. Radin. The 801 Minicomputer. In *Proc. of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, Palo Alto CA, March 1982.

[Rama81] K. Ramamohanarao and R. Sacks-Davis. Hardware address translation for machines with a large virtual memory. *Information Processing Letters*, 13(1):23–29, 1981.

[Rama93] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the CORAL Deductive Database System. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993.

[Rash88] Richard F. Rashid, Avadis Tevanian, Michael Young, David B. Golub, Robert V. Baron, David L. Black, William Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[Redd75] Y. V. Reddy. Optimal Segment Size for Storage Allocation in a Multiprogrammed Computer System. In *Proc. of IEEE Computer Society Conference*, pages 303–305, September 1975.

[Rein93] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[Rein94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[Repp94] John H. Reppy. A High-performance Garbage Collector for Standard ML, 1994. AT&T Bell Labs Technical Memo.

[Rome95] Ted Romer, Wayne Ohlrich, Anna Karlin, and Brian Bershad. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 176–187, June 1995.

[Rose85] J. Rosenberg and D. A. Abramson. MONAD PC: A Capability Based Workstation to Support Software Engineering. In *Proc. of the 18th Hawaii International Conference on System Sciences*, pages 222–231, 1985.

[Rose92] J. Rosenberg, J. L. Keedy, and D. Abramson. Addressing Large Virtual Memories. *The Computer Journal*, 35(4):369–376, 1992.

[Saty81] M. Satyanarayanan and D. Bhandarkar. Design Trade-offs in VAX-11 Translation Buffer Organization. *IEEE Computer*, 14(12):103–111, December 1981.

[Silh93] Ed Silha. *The PowerPC Architecture, IBM RISC System/6000 Technology, Volume II*. IBM Corp., 1993.

[Sing92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[Site92] Richard L. Sites. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[Site93] Richard L. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

[Slea85] D. D. Sleator and R. E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, pages 202–208, February 1985.

[Smit78a] A. 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.

[Smit78b] Alan J. Smith. Sequential Program Prefetching in Memory Hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[Smit78c] Alan Jay Smith. Bibliography on Paging and Related Topics. *Operating Systems Review*, October 1978.

[Smit82] Alan Jay Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[Smit86] Alan Jay Smith. Bibliography and Readings on Cache Memories. *Computer Architecture News*, 11(1):22–42, January 1986.

[Smit87] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon. The ZS-1 Central Processor, 1987.

[Smit88] J. E. Smith and A. R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Trans. on Computers*, C-37(5):562–573, May 1988.

[Smit91] Alan Jay Smith. Second Bibliography on Cache Memories. *Computer Architecture News*, 19(4):154–182, June 1991.

[So88] Kimming So and Rudolph N. Rechtschaffen. Cache Operations by MRU Change. *IEEE Trans. on Computers*, C-37(6), June 1988.

[SPAR91] SPARC International Inc. The SPARC Architecture Manual, Version 8, 1991.

[SPAR94] SPARC International Inc. The SPARC Architecture Manual, Version 9, 1994.

[SPEC91] SPEC. (entire issue). *SPEC Newsletter*, 3(4), December 1991.

[Sriv94] Amitabh Srivastava and Alan Eustace. ATOM A System for Building Customized Program Analysis Tools. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[Tall92] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in Supporting Two Page Sizes. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 415–424, May 1992.

[Tall93] Madhusudhan Talluri, Yousef A. Khalidi, Dock Williams, and Vikram Joshi. Virtual Memory Computer System Address Translation Mechanism that Supports Multiple Page Sizes. Patent application filed, Serial No. 08/139,549, Sun Microsystems, October 1993. (Accepted 1995).

[Tall94a] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of Superpages with Less Operating System Support. In *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, San Jose, CA, October 1994.

[Tall94b] Madhusudhan Talluri and Yousef A. Khalidi. Apparatus and Method for Efficient Sharing of Virtual Memory Translations. Patent application filed, Serial No. 08/333,487, Sun Microsystems, November 1994.

[Tall95] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. A New Page Table for 64-bit Address Spaces. In *(To appear) Proceedings of 15th ACM Symposium on Operating System Principles*, December 1995.

[Tayl81] Mitchell B. Taylor. *Efficient Memory allocation with the buddy algorithm*. Motorola, November 1981.

[Tayl90] George Taylor, Peter Davies, and Michael Farmwald. The TLB Slice - A Low-Cost High-Speed Address Translation Mechanism. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 355–363, June 1990.

[Tell90] Patricia J. Teller. Translation-Lookaside Buffer Consistency. *IEEE Computer*, 23(6):26–36, June 1990.

[Thak86] Shreekant S. Thakkar and Alan E. Knowles. A High-Performance Memory Management Scheme. *IEEE Computer*, pages 8–22, May 1986.

[Thom74] K. Thompson and D. M. Ritchie. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[Turn81] R. Turner and H. Levy. Segmented FIFO Page Replacement. In *Proc. of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–51, 1981.

[Uhli94] Richard Uhlig, David Nagle, Trevor Mudge, and Stuart Sechrest. Tapeworm II: A New Method for Measuring OS Effects on Memory Architecture Performance. In *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 132–144, October 1994.

[Wada92] Tomohisa Wada, Suresh Rajan, and Steven A. Przyblski. An Analytical Access Time Model for On-Chip Cache Memories. *IEEE Journal of Solid State Circuits*, 27(8):1147–1156, August 1992.

[Wang93] Chia-Jiu Wang and Frank Emnett. Implementing Precise Interruptions in Pipelines RISC Processors. *IEEE Micro*, 13(4):36–43, August 1993.

[West88] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design*. Addison Wesley, 1988.

[Whee92] Bob Wheeler and Brian N. Bershad. Consistency Management for Virtually Indexed Caches. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 124–136, Boston MA, October 1992.

[Wilk92] J. Wilkes and B. Sears. A comparison of protection lookaside buffers and the PA-RISC protection architecture. Technical Report HPL-92-55, Hewlett-Packard Laboratories, March 1992.

[Wilt93] Steven J. E. Wilton and Norman P. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches. WRL Research Report 93/5, DEC Western Research Lab, 1993.

[Wood86] David A. Wood, S. J. Eggers, G. Gibson, Mark D. Hill, J. Pendleton, S. A. Ritchie, Randy H. Katz, and David A. Patterson. An In-Cache Address Translation Mechanism. In *Proc. of the 13th Annual International Symposium on Computer Architecture*, pages 158–166, Tokyo Japan, June 1986.

[Yoo93] Hyuck Yoo and Tom Rogers. UNIX Kernel Support for OLTP Performance. In *1993 Winter USENIX Conference*, pages 241–247, January 1993.

[Youn89] Michael W. Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University, November 1989.

[Yung94] Robert Yung and Leslie Kohn. *UltraSPARC Programmer's Reference Manual*. Sun Microsystems Inc., 1994.

[Yung95] Robert Yung. UltraSPARC-I (Spitfire) Architecture. Technical report, Sun Microsystems, April 1995.