Surpassing the TLB Performance of Superpages with Less Operating System Support

Madhusudhan Talluri and Mark D. Hill.
Computer Sciences Department, University of Wisconsin, Madison WI 53706
{talluri,markhill}@cs.wisc.edu

Abstract

Many commercial microprocessor architectures have added *translation lookaside buffer* (TLB) support for *superpages*. Superpages differ from segments because their size must be a power of two multiple of the base page size and they must be aligned in both virtual and physical address spaces. Very large superpages (*e.g.*, 1MB) are clearly useful for mapping special structures, such as kernel data or frame buffers. This paper considers the architectural and operating system support required to exploit medium-sized superpages (*e.g.*, 64KB, *i.e.*, sixteen times a 4KB base page size). First, we show that superpages improve TLB performance only after invasive operating system modifications that introduce considerable overhead.

We then propose two subblock TLB designs as alternate ways to improve TLB performance. Analogous to a subblock cache, a *complete-subblock TLB* associates a tag with a superpage-sized region but has valid bits, physical page number, attributes, etc., for each possible base page mapping. A *partial-subblock TLB* entry is much smaller than a complete-subblock TLB entry, because it shares physical page number and attribute fields across base page mappings. A drawback of a partial-subblock TLB is that base page mappings can share a TLB entry only if they map to consecutive physical pages and have the same attributes. We propose a physical memory allocation algorithm, *page reservation*, that makes this sharing more likely. When page reservation is used, experimental results show partial-subblock TLBs perform better than superpage TLBs, while requiring simpler operating system changes. If operating system changes are inappropriate, however, complete-subblock TLBs perform best.

This work was primarily supported by a National Science Foundation Presidential Young Investigator Award (MIPS-8957278) and a Sun Microsystems External Research Grant. The experiments were performed on equipment donated by Sun Microsystems.

This paper, without Appendix B (Sensitivity Analysis), appears in the Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI), October 1994.

1. Introduction

Most architectures that support paged virtual memory [Denn70] accelerate address translation with a *translation lookaside buffer*[1] (TLB). A TLB is a cache whose tags are virtual page numbers (VPN) and data are physical page numbers (PPN), page attributes (*e.g.*, protection, cacheability), and optional reference and modified bits [Mile90, Henn90, Smit82]. TLBs must be studied again, because of current workload and processor trends.

Future workloads will demand greater *TLB reach*—the maximum size of memory mapped by a TLB—than today. Typical physical memory sizes continue to follow their historical exponential growth curve with 100MB+ memories likely to be common when the microprocessors being designed today are deployed in systems. It seems unlikely that this demand for physical memory is occurring without a commensurate increase in memory use (*e.g.*, larger working sets). Furthermore, the growing importance of nontraditional computation, such as multimedia, is likely to increase memory usage and change locality patterns. TLBs must be designed for larger TLB reach to support future applications.

Furthermore, processor trends require that the increased TLB reach be provided with a fast TLB access time. Dramatic reductions in processor cycles-per-instruction (CPI), from ten to less than one, have increased the relative importance of TLBs. In addition, the continued use of physically-tagged level-one caches places TLB access times on the cache-access critical path. Furthermore, the trend toward supporting multiple cache accesses per cycle (*e.g.*, Intel Pentium and SGI R8000 (TFP)) also means that the TLB must support multiple translations per cycle through multi-porting or replication. Multi-porting increases TLB complexity and access time, while replication increases cost. Both suggest that the brute force solution of increasing TLB reach by making larger TLBs may be unattractive. Finally, current and future TLBs are on microprocessor chips, so TLB design is a part of chip design instead of system design, as in the past. Prudent designers will seek TLBs that serve many

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

workloads to avoid condemning their chips to limited markets.

For these and other reasons, several recent microprocessor architectures support one or more sizes of *superpages*. 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. A superpage of size B is aligned if it begins at an address that is a multiple of B. Supporting superpages is easier than supporting segments, which use a two-dimensional address space, may be arbitrarily long, and may start at arbitrary virtual and physical addresses [Orga72]. Architectures that support superpages include MIPS R4x00, DEC Alpha, SPARC, PowerPC, Intel, ARM, Motorola 68K and HP PA-RISC. The MIPS R4000 [Kane92], for example, supports a 4KB base page size and superpages of 16KB, 64KB, 256KB, 1MB, 4MB and 16MB.

The clear motivation for supporting superpages is that using them appears to increase TLB reach *for free*. This is certainly true for very large superpages (*e.g.*, 1MB) as they are very effective in mapping large objects such as kernel data, frame buffers and large arrays. Some architectures specify two TLBs—one for base pages and another for superpages (*e.g.*, PowerPC)—and allow for restricted use of superpages with special operating system support. We assume that TLBs will include special support for large superpages and operating systems will use them.

In this study, we concentrate only on the benefits and costs of supporting medium-sized superpages (*e.g.*, 64KB). Thus, in the rest of this paper, when we say *superpages* we mean *medium-sized superpages*.

The impact of supporting (medium-sized) superpages in TLBs is twofold. First, it appears that the TLB must be fully-associative, because selecting a set with the least significant bits of the virtual page number is difficult when the page size in not known [Tall92]. The SGI R8000 (TFP), for example, implements a set-associative TLB, but restricts a process to a single page size [MIPS93]. Second, the complexity or time needed to handle TLB misses is likely to be larger for superpages. We expect this to be offset easily by the reduction in the number of TLB misses.

The impact of supporting superpages, however , is not limited to the TLB. Ef fective paged virtual memory requires considerable operating system support, and superpages are no exception. T able 1 motivates the rest of this paper by giving an example which shows that superpage TLBs significantly reduce the number of user TLB misses only after substantial operating system modifications. The SPEC benchmark **gcc** [SPEC91] requires page table changes and superpage support in the file system for mapped files. A uniprocessor version of the SPLASH benchmark **mp3d** [Sing92] requires superpage support for mapped files and heaps. In both cases TLB performance is important, because the benchmarks spend 3% and 11% of user time in TLB miss handling respectively; other benchmarks spend up to 50% (T able 5 in Section 5.5)!

Section 2 elaborates on why operating system support for superpages has an invasive ef fect on operating system data structures and interfaces, including how increased operating system overheads for superpages may *increase* execution time in some cases. This causes us to ask: (1) *Can we surpass the TLB benefit of superpages while adding less over head to existing operating systems?* and (2) *What is the best TLB design if operating system changes are inappropriate?*

Section 3 answers both questions by proposing subblock TLBs, analogous to subblock caches. Subblock caches associate with each address tag several data subblocks that each have their own valid bit so that they may be loaded independently. With each tag, a subblock TLB associates valid bits and other information for several base pages. With 4KB base pages and *subblock factor* 16, for example, each tag identifies an aligned 64KB virtual address region (like a superpage) while the data portion has sixteen independent PPNs, attributes, etc. We call this design *complete-subblocking*. Alternatively, one can make the TLB entry much smaller by having only one copy of the PPN and attributes and allow base pages to share a TLB entry only if they are *properly placed* in a superpage-sized region of physical memory and have identical attributes. Base pages that do not meet these conditions are allowed, but their translations will be cached in different TLB entries. We call this design *partial-subblocking.*

Table 1: Effect of Operating System Support on a 64-entry fully-associative TLB

Level of Operating System Support (Each row adds substantial modifications to the OS)	TLB	# User TLB misses (thousands)	
		gcc	mp3d
Base system	4KB single-page-size	3335	4050
Base system	4KB/64KB Superpage	3335	4050
+ TLB and Page table support		3335	4050
+ Superpage support for mapped files		994	4049
+ Superpage support for heaps		495	13
% of user time spent in TLB miss handling in base system	either	3%	11%

Section 4 introduces a physical memory allocation algorithm that makes partial-subblock TLBs effective by often mapping consecutive base virtual pages to consecutive aligned base physical pages. On a page fault, it tries to map the base page into an unused superpage-aligned region of memory and tries to place the other base pages in the region into a **reserved** state. A **reserved** physical page is **free** but can be reused when absolutely necessary. The same algorithm can be used to efficiently support superpages.

Section 5 discusses our evaluation methodology. We first describe, *Foxtrot*, a version of a commercial operating system (Solaris 2.1) we modified to do these studies. Next we describe our trap-based simulation and metrics. Finally, we discuss the workloads we selected to pressure TLBs.

Section 6 gives our experimental results. When operating system changes are allowed, we recommend partial-subblock TLBs. They perform better than superpage TLBs and require much less chip area than complete-subblock TLBs. In particular, a partial subblock TLB entry can map multiple base pages in situations where the guarantees needed to use superpages are not met (e.g., for unaligned segments, small objects, and non-uniform attributes). When operating system changes are inappropriate, complete-subblock TLBs perform best.

While many commercial architectures support medium-sized superpages in their TLBs, there are few published studies on their impact on TLBs or operating systems. Kagimasa *et al*. [Kagi91] describe a system using two page sizes in a partitioned address space. Chen *et al.* [Chen92] and Talluri *et al.* [Tall92] present data that supports the use of superpages. Khalidi *et al.* [Khal93] and Mogul [Mogu93] raise some operating system issues researchers should address to support superpages.

2. Operating System Support for Superpages

Effective paged virtual memory [Denn70] requires coordinated support from a computer's operating system and hardware architecture. Operating system support for virtual memory with a single fixed page size is substantial but well-understood (*e.g.,* UNIX, VMS, NT, MACH). It includes a virtual memory manager that allocates virtual addresses, enforces protection, initiates I/O and loads/unloads mappings from a page table; file systems that manage and maintain structure/coherence of objects on disk/network; a physical memory manager that manages/allocates pages for file systems; a page replacement process; and a hardware-dependent layer that manages TLBs and page tables.

Most facets of paged virtual memory operating system policies and mechanisms require modifications to support superpages effectively. We first describe a new policy—*page-size assignment*—and two new mechanisms—*page promotion* and *page demotion*. We then briefly discuss the impact of supporting superpages on existing operating system policies and mechanisms.

A *page-size assignment policy* decides the page size to use for each virtual address. The policy may change over time, differ between objects and differ between processes. A policy must balance the costs and benefits of using superpages. A *static page-size assignment policy* will make the decision once and fix the page size over the life of the mapping (*e.g.*, for frame buffers). Often the operating system does not know, in advance, enough about the costs and characteristics of accesses to the object to make an informed static decision. The operating system will then have to use a *dynamic page-size assignment policy* guessing a page size to use and modifying it, if the guess was incorrect. Implementation of the policy will span the virtual memory manager and the file systems.

Two additional operating system mechanisms support a dynamic page-size assignment policy. *Page promotion* is the mechanism by which a set of pages are coalesced to a larger superpage. *Page demotion* is the reverse process. The operating system uses these mechanisms when it decides to switch page sizes for a virtual address range. *Page demotion* involves unloading the superpage mapping, and possibly replacing it with base page mappings. *Page promotion* may involve verifying that the base pages are compatible for promotion, unloading any existing base page mappings from the page tables and TLBs, allocating contiguous physical memory, copying the base pages to contiguous memory—a *gather* operation—doing additional I/O and updating page tables and TLBs. A *gather* operation is very expensive and may more than offset any TLB performance improvement due to use of superpages.

The impact of adding superpage support to operating systems is twofold. First, it adds significant overhead (time spent in the operating system) and makes superpages less attractive. These overheads are fundamental to the use of superpages and independent of the operating system. For example, using superpages increases the amount of I/O, page initialization overhead, and page fault latency. If the operating system is efficient and reduces the cost of these overheads, superpages can be used more often to improve TLB performance. For example, intelligent physical memory allocation can remove the need for a gather operation during page promotion. Also, TLB misses incur a higher average miss penalty since the page tables are expected to be more complicated when using superpages.

Second, the changes required for efficient superpage support are invasive and affect large portions of existing operating systems. Physical memory management, for example, must be overhauled to handle variable sizes and external fragmentation [Knut68, Pete77]. Many key data structures (*e.g.*, page tables) and interfaces need to be redesigned. Use of superpages often conflicts with file system read-ahead and requires coordination on what would otherwise have been local policy decisions. Many

Table 2: Operating system overheads for superpage support

OS Mechanism/Policy	Overhead
Page Replacement & File System Writes	Modified/Referenced information is available at a coarse granularity. Results in increased disk/network write traffic and may increase page fault rate.
Page Fault Handling	Superpages increase pagefault latency and program execution time. Operating systems would otherwise overlap some of the I/O with execution.
Physical Memory Management & Page promotion: Allocate	Physical memory cannot be treated as equal-sized pages. Requires an algorithm to efficiently allocate memory in variable-sized chunks [Knut68].
Physical Memory Management	During periods of high memory demand, external fragmentation prevents use of superpages. Many of the operating system modifications for superpages continue to add overhead, even though there is no further TLB benefit.
Data Structures	Linear arrays and hash tables do not scale efficiently to include superpages. *e.g.*, page tables and most hash tables. Algorithms traversing more complicated data structures take longer, *e.g.*, TLB miss penalty increases.
Page promotion: Populate	Superpages increase internal fragmentation and memory demand. Significant time is spent in I/O and initializing memory that the program never references.
Page promotion: Check	Operating system has to *guarantee* that the constraints for superpages are satisfied. Adds a check, sometimes of information not easily accessible.
Page promotion: Gather	If the base pages involved in the promotion are not contiguous in physical memory, the contents must be copied to a superpage. Adds significant overhead.

Table 3: Modifications required for OS mechanisms and policies to support superpages

OS Mechanism/Policy	Modifications
Page-size assignment	New policy. It is difficult to balance the costs and benefits of page promotion as both are often not easily estimated or known in advance.
Page Replacement	Replacement policies, such as CLOCK, give equal weight to all pages. Superpages have a higher cost. Requires re-evaluation of page replacement policies.
File System read-ahead/ Page Clustering & Page-size assignment	File systems and device drivers read-ahead and cluster I/O into efficient large requests. Superpages already include this benefit. File systems and the virtual memory manager must coordinate to avoid making locally-optimal decisions.
Physical Memory Management & Page Coloring	Superpages already include some of the benefits of page coloring—a superpage consists of one base page from each physical equivalence class. But large physically-tagged caches will require page coloring with superpages too.
Aliases and Synonyms	Aliases could use different page sizes and the page sizes for a virtual address and corresponding physical address may differ. Complicates data structures.

Table 4: Cases where superpages are inadequate

OS Mechanism/Policy	Issues
Page-size assignment: Small objects	Objects smaller than a superpage cannot use superpages. *e.g.*, a 60KB object has to use 15 base pages while a 64KB object could use one superpage.
Page-size assignment: non-uniform attributes	Applications using fine-grain protection (*e.g.*, copy-on-write) have to forgo the benefits of superpages if even a singe base page has different attributes.
Virtual Address Allocation	Objects mapped into an address space may not start or end at superpage-aligned addresses. Restricts use of superpages.
Interfaces	Many interfaces assume a single page size. *e.g.*, external pager interfaces. Superpages are hard to use efficiently with existing interfaces.

of these changes also adversely affect the performance of programs that do not use superpages.

Table 2 lists a sample of the overheads that operating systems incur when using superpages. For superpages to be useful, the TLB benefit of using superpages must be greater than the costs due to these overheads. Table 3 lists some modifications to important operating system policies and mechanisms to support superpages efficiently. Table 4 lists some situations where superpages are inadequate. A detailed discussion is beyond the scope of this paper. In the next section, we describe subblock TLBs, which reduce the burden on the operating system, and, also deliver better TLB performance.

3. Subblock TLBs

Subblocking[2], borrowed from cache design, makes TLBs more effective than superpages while requiring simpler operating system support. Subblock TLBs have the TLB reach advantages of superpages and, in addition, can exploit these advantages more often than superpages, (*e.g.*, for objects smaller than superpage size). However, subblocking requires larger TLB entries and additional control logic.

Figure 1 illustrates the structure of a single entry for the four different types of TLBs we will consider. Entries may be combined to build fully-associative or set-associative TLBs. The first entry illustrates a non-subblocked TLB entry that maps a single base page and consists of Tag and Data fields. The Tag consists a virtual page number (VPN), and Data contains a physical page number (PPN), attributes (Attr., *e.g.*, protection, cacheability), modified (Mod) and valid (V) bits.

The next entry in Figure 1 illustrates a superpage TLB entry. The Tag includes a Size field that masks bits during tag compare and physical address generation.

Next, Figure 1 illustrates a *complete-subblock TLB* entry with subblock factor 4. A complete-subblock TLB entry with a subblock factor **n** has an **n** times larger data portion but a $\log_2(\mathbf{n})$ bits smaller tag than a non-subblocked TLB entry. The MIPS R4x00 has, for example, a complete-subblock TLB with a subblock factor of 2 [Kane92]. On a TLB miss, before attempting a replacement, the tags and valid bits are checked to see if an empty subblock can hold the mapping. Alternatively, all subblocks can be loaded on a TLB miss. The IBM RS/6000 and ARM 6x0, for example, support subblock attributes and require all subblocks to be valid.

There are at least six advantages to using complete-subblocking over superpages, even though subblock and superpage TLBs have the same TLB reach (with subblock factor **n** and superpage size **n** times the base page size). First, complete-subblock TLBs allow applications to get all the benefits of using superpages with no operating system modifications beyond the TLB management code. Second, complete-subblock entries can map multiple base pages in situations where superpages cannot be used, such as, for unaligned segments, small objects, nonuniform attributes. Third, set-associative subblock TLBs are straightforward, while superpage ones are not. Fourth, subblocking does not increase internal fragmentation or require additional I/O as superpages do (subblock caches, similarly, reduce bus bandwidth usage). Fifth, subblocking allows all but the referenced base page to be loaded asynchronously, thereby not incurring the larger page fault latency of superpages (subblock instruction caches often use a similar technique to reduce cache miss penalty). Sixth, subblocking maintains reference and modified information at the finer granularity of the base page size.

Complete-subblocking, however, has one major disadvantage compared to superpages, namely, that a complete-subblock TLB entry occupies a larger area than a superpage TLB entry. For example, using the area model described in Appendix A, a complete-subblock TLB with subblock factor 16 is about 4.5 times larger than a superpage TLB with the same number of entries. Also the increased area can translate into increased access time, but the tradeoffs are complex. Complete-subblock TLBs, for example, can be set-associative and avoid slower CAM (content addressable memory) cells required in fully-associative designs. Rather than quantify the precise effect, we next propose partial-subblocking which substantially mitigates the area and speed disadvantage of complete-subblocking.

The final entry of Figure 1 shows a *partial-subblock TLB* entry (with subblock factor 4) where we coalesce the four pairs of PPN and Attr fields into a single pair. The good news is that the entry is now not much larger in area than a superpage TLB entry, yet by maintaining individual valid and modified bits we retain many advantages of complete subblocking. The bad news is that base pages can share a partial-subblock TLB entry only if they have identical attributes and are properly placed in a superpage region.

With subblock factor **n**, base pages `x` and `y` are *properly placed* only if they are placed in the same superpage region (`PPN(x) div n = PPN(y) div n` and `VPN(x) div n = VPN(y) div n` where `div` is integer division) and are superpage aligned (`VPN(x) mod n = PPN(x) mod n` and `VPN(y) mod n = PPN(y) mod n`, where `mod` is the modulus operator). Translations that do not meet these conditions are allowed but are cached in separate TLB entries. TLB entries cache unaligned base page mappings (`VPN(x) mod n ≠ PPN(x) mod n`) by disabling subblocking (SB=0) for those entries.

Figure 2 shows how different fully-associative TLBs cache mappings to four consecutive virtual pages that are backed by noncontiguous physical pages. A single-page-size TLB will require all four TLB entries. A superpage TLB also will require four TLB entries[3] as *all* the physical pages were not contiguous. A complete-subblock TLB

2. Subblocking [Hill84] is also called sectoring [Lipt68] and address/transfer blocks [Good83].

Figure 1: Single TLB entry

	Tag					Data						
Single-page-size TLB entry	VPN					PPN	Attr	Mod	V			
Superpage TLB entry	VPN	Size				PPN	Attr	Mod	V			
Complete-subblock TLB entry (subblock factor = 4)	VPN(-2bits)					PPN0	Attr	M0	V0			
						PPN1	Attr	M1	V1			
						PPN2	Attr	M2	V2			
						PPN3	Attr	M3	V3			
Partial-subblock TLB entry (subblock factor = 4)	VPN(-2bits)	V0	V1	V2	V3	PPN	Attr	SB	M0	M1	M2	M3

Figure 2: TLB entry examples

Page 1: VPN = 110100	PPN = 10000	Attr = α
Page 2: VPN = 110101	PPN = 11011	Attr = α
Page 3: VPN = 110110	PPN = 00010	Attr = α
Page 4: VPN = 110111	PPN = 00011	Attr = α

Single-Page-Size TLB

110100	10000	α	Mod	✓
110101	11011	α	Mod	✓
110111	00011	α	Mod	✓
110110	00010	α	Mod	✓

✓ => Valid

× => Invalid

4K/16K Superpage TLB

110100	4KB	10000	α	Mod	✓
110101	4KB	11011	α	Mod	✓
110111	4KB	00011	α	Mod	✓
110110	4KB	00010	α	Mod	✓

Complete-subblock TLB (subblock factor = 4)

1101	10000	α	M0	✓	11011	α	M1	✓	00010	α	M2	✓	00011	α	M3	✓
Unused				×				×				×				×
Unused				×				×				×				×
Unused				×				×				×				×

Partial-subblock TLB (subblock factor = 4)

1101	✓	×	×	×	10000	α	0	M0			
1101	×	×	✓	✓	00000	α	0			M2	M3
1101	×	✓	×	×	11011	α	1		M1		
Unused	×	×	×	×							

Figure 3: Partial-subblock TLB examples.

Page 1: VPN = 110100	PPN = 000000	Attr = α
Page 2: VPN = 110101	PPN = 000001	Attr = α
Page 3: VPN = 110110	PPN = 111000	Attr = α
Page 4: VPN = 110111	PPN = 000011	Attr = α

1101	✓	✓	×	✓	000000	α	0	M0	M1		M3
1101	×	×	✓	×	111000	α	1			M2	

Example (a) Noncontiguous Physical Memory

Page 1: VPN = 110100	PPN = 000000	Attr = α
Page 2: VPN = 110101	PPN = 000001	Attr = α
Page 3: VPN = 110110	PPN = 000010	Attr = α

1101	✓	✓	✓	×	000000	α	0	M0	M1	M2	

Example (b) Small object

Page 1: VPN = 110101	PPN = 000001	Attr = α
Page 2: VPN = 110110	PPN = 000010	Attr = α
Page 3: VPN = 110111	PPN = 000011	Attr = α

1101	×	✓	✓	✓	000000	α	0		M1	M2	M3

Example (c) Unaligned start address

Page 1: VPN = 110100	PPN = 000000	Attr = α
Page 2: VPN = 110101	PPN = 000001	Attr = α
Page 3: VPN = 110110	PPN = 000010	Attr = γ
Page 4: VPN = 110111	PPN = 000011	Attr = α

1101	✓	✓	×	✓	000000	α	0	M0	M1		M3
1101	×	×	✓	×	000000	γ	0			M2	

Example (d) Copy-on-write/different attributes will use a single TLB entry. A partial-subblock TLB will use three TLB entries as two pages (VPNs 110110 & 110111) are aligned and share an entry.

One implementation difficulty for partial-subblock TLBs is that the same tag value could be loaded into more than one TLB entry, as happened in Figure 2 when the PPNs were not aligned. Such a multiple match condition could cause electrical problems for some implementations. A straightforward solution is to combine the valid bits with the tag so that, at most, one TLB entry can match on a lookup. This change may slightly increase TLB area and access time.

Finally, some operating system support is necessary to make partial-subblock TLBs effective. If the operating system succeeds in allocating aligned physical pages for two out of four base pages, a partial-subblock TLB can use just three TLB entries to hold the four mappings (Figure 2). The operating system could use a best-effort allocation algorithm and does not have to guarantee contiguity as in superpages. If all four pages were allocated contiguous pages, a single TLB entry could be used or, alternately, a superpage could be used. The operating system support for partial-subblocking has two advantages over superpages. First, while partial-subblocking requires superpage-like support in physical memory management and file systems, it is more efficient and less intrusive than superpage support. Second, partial-subblocking can be used to exploit the TLB reach advantages in more situations than superpages. For example, Figure 3 shows how partial-subblock TLBs can increase their effective TLB reach when mapping small objects, objects with unaligned starting addresses and copy-on-write cases, situations where superpages cannot be used.

A superpage mapping requires only one TLB miss to be loaded into the TLB, but a subblock TLB will require multiple TLB misses for all the subblocks to be loaded. By paying a higher miss penalty, the TLB miss handler could preload the mappings corresponding to all the subblocks that will be cached in the same TLB entry or all mappings in a superpage region even if they must be cached in different TLB entries. In either case, TLB hardware support can help reduce a much higher TLB miss penalty (*e.g.*, to check whether page attributes and PPNs are compatible).

Superpages and partial-subblocking can be integrated into a single TLB. Partial-subblock support is very effective for medium-sized objects for which the operating system finds page-size assignment hard, for example, program text, shared libraries, and data and many heap segments. Superpages are easily the choice for very large objects for which the user or operating system can do static page-size assignment, for example, kernel data, frame buffers and large heaps.

This section explained subblock TLBs and their advantages. Partial-subblock TLBs are most effective with operating system support for aligned physical memory allocation. In the next section we describe an efficient algorithm for such physical memory allocation.

4. Physical Memory Allocation—Page reservation

The effectiveness of superpage and partial-subblock TLBs depends on the ability of the operating system to allocate aligned physical memory. In this section we describe a physical memory allocation algorithm, *page reservation*, which attempts to allocate memory in a way that helps TLB performance. While superpages require the operating system to *guarantee* contiguity and require page promotions that may include gather operations, partial-subblocking only requires a *best-effort*.

Physical memory is usually divided into equal-sized pages that are 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 id, offset>) and vice versa. When a new page is required, the physical memory allocator searches the index structure to avoid duplicate allocations. Then it chooses a **free** page and updates the index structures. More than one process may map the same physical page using different virtual addresses. Hence, the physical memory manager uses the unique object page identity instead of virtual addresses in the index structures.

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 does not help the performance of either superpage or partial-subblock TLBs.[4] *Page reservation*, described next, allocates physical pages in aligned superpage-sized regions, effectively treating physical memory as a fully-associative subblocked-cache with superpage-sized blocks and base-page-size subblocks.

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

Page reservation works as follows. On an initial base page fault (or during read-ahead by the file system), the

3. A 8KB superpage could be used for pages 3 and 4, if supported. Further, if the page promotion costs were justified, a superpage of 16KB could be used after copying the pages into a superpage.

4. Page coloring [Tayl90, Kess92] also carefully selects physical pages for virtual addresses but for a different purpose and in a different way than page reservation. Page coloring seeks to reduce physical cache conflict misses by partitioning virtual and physical pages into equivalence classes and reducing the probability of virtual pages from different VPN equivalence classes being allocated to the same PPN equivalence class. Page coloring, however, makes no attempt to place consecutive virtual pages into consecutive physical pages, as page reservation does.

physical memory manager allocates a superpage-sized region of base pages. With superpage size 64KB, for example, a page fault to address 0x41034 allocates sixteen base pages: the object pages corresponding to virtual addresses 0x40000, 0x41000, 0x42000,..., 0x4f000. The accessed base page (0x41000) is loaded as normal and marked **busy**. Other base pages are marked **reserved** and added to the end of the reserved list. If the physical memory manager runs out of **free** pages, it frees pages by moving them from the head of the reserved list to the end of the free list. Subsequent page faults may find previously reserved base physical pages **reserved** or not. If **reserved**, the **reserved** physical page will be allocated and marked **busy**; if not, a physical page from the free list will be allocated.

Page reservation provides a natural feedback mechanism for improving the effectiveness of partial-subblock and superpage TLBs without unduly increasing memory demand. In periods of low memory demand, pages will be allocated from **reserved** physical pages, allowing multiple base pages to share a partial-subblock TLB entry. Superpage TLBs benefit, because page promotion can be done without the cost of gathering base pages together. 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 should be no significant change in the page fault rate from the non-superpage implementation.

Page reservation for both superpages and partial-subblocking requires the physical memory manager to find free superpages. External fragmentation, where memory was allocated such that no free superpages are available but there are still sufficient free base pages, will cause page reservation to degenerate to standard page allocation. Memory management techniques for variable-sized objects have been studied extensively. There are well-known techniques to minimize external fragmentation [Knut68, Pete77]. Some file systems also use similar techniques to reserve disk space [McKu84].

Page reservation significantly improves the performance of partial-subblock TLBs and reduces page promotion cost if using superpages. However, we did not study the effect of page reservation on cache behavior.

5. TLB Simulation Methodology

In this section, we describe the operating system support, simulation technique, metrics, and workloads used to compare the performance of single-page-size, superpage, and subblock TLBs.

5.1. Foxtrot: Operating System Prototype

A study of superpage and partial-subblock TLBs requires appropriate operating system support. We are not aware of such support in any operating system. We built *Foxtrot*, based on Solaris 2.1, to serve as a test-bed for our on-going operating system research. It includes modified or new mechanisms for virtual address allocation, physical memory management, page fault handling, page reservation, and page promotion/demotion. Instead of requiring the virtual memory system to unload base page mappings and reload superpage mappings during page promotion, the Foxtrot only loads base page mappings in the page table. The page table manager coalesces neighboring PTEs into superpage mappings if they are compatible.

Foxtrot supports partial-subblock TLBs using page reservation and file system prefetching. For page reservation, Foxtrot uses a superpage-sized region that corresponds to the TLB type, *e.g.*, a TLB with subblock factor 16 will use a superpage of 64KB. When objects are smaller than a superpage-sized region, Foxtrot only reserves base pages up to the object size. Sometimes, the object must also be reserved (*e.g.*, heaps require swap space allocation). For disk files, Foxtrot also initiates asynchronous I/O for the region. File system clustering makes the I/O more efficient. Foxtrot does not prefetch for **nfs** and heap objects as it is more expensive.

When supporting superpages, Foxtrot uses a dynamic page-size assignment policy which does page reservation and prefetching as in the partial-subblock TLB case, and, in addition, makes policy decisions on when to promote base pages to superpages as follows:

- For every superpage region, the virtual memory manager maintains a count of the base pages within the region that have mappings in the page table. Page promotion occurs when the count exceeds the *page promotion threshold*. The page promotion threshold depends on the cost of populating the pages—heap (100%), disk files (50%), **nfs** files (75%).
- During page promotion, the attributes and physical page numbers for the base pages within the superpage region are checked to see if a superpage mapping can indeed be used. Foxtrot does not implement the gather operation, so page promotion fails when it requires a gather (if the PPNs are not contiguous).
- Foxtrot’s does not do page reservation or prefetches for regions smaller than a superpage.

While this may not be the optimal policy or the most efficient implementation, it is “a” policy. Superpage and partial-subblock TLB simulations without operating system support are unrealistic. While *Foxtrot* can support many page-size assignment policies, this paper focuses on TLB performance by fixing the operating system mechanisms and policies.

5.2. Trap-based simulation

We use *trap-based simulation* to compare the performance of superpage and subblock TLBs. Trap-based simulation for TLBs manipulates the valid bits in the page table to invoke a TLB simulator on page faults. The simulator maintains a data structure corresponding to the

TLB under study, the *target TLB*, and marks valid only those page table entries that reflect the contents of the target TLB. This technique invokes the simulator only on target TLB misses and *never* on hits [Uhli94]. The kernel is modified to account for operating system effects (*e.g.*, TLB invalidations) and superpage support.

Trap-based simulation has significant advantages over trace-driven simulation. First, TLB simulation requires information that is hard to encapsulate in a trace, such as page-size assignment, physical page numbers, and attributes. The simulator has access to such information in the kernel. Second, trap-based simulation incurs overhead only on very infrequent TLB misses, allowing hits to proceed at hardware speed. Our simulator runs three to four orders of magnitude faster than a trace-driven simulation. Third, trap-based simulation naturally extends to multi-program workloads.

The key disadvantage of trap-based simulation is the inability to calculate the number of TLB hits without hardware support such as profiling counters [Site93] or external probes [Nagl92]. This makes it difficult to use normalized metrics (*e.g.*, TLB miss ratio).

Foxtrot implements trap-based simulation for SPARC V8 processors [SPAR91]. The cost of a target TLB miss—including trap cost, TLB simulator complexity and wrappers, much of which is written in C—is 1500 to 4000 cycles, comparable to the overhead seen by others [Uhli94, Rein93]. Our implementation, however, does not account for kernel references.

5.3. Metrics

While the ultimate measure of TLB performance is the fraction of execution time spent in servicing TLB misses, the TLB miss ratio is often used instead. As explained above, our simulator lacks the capability to count the number of TLB hits and we use the unnormalized number of TLB misses as our metric for comparing different TLBs. We also normalize the number of TLB misses by dividing by the number of TLB misses in an equivalent single-page-size TLB. In Table 5, we also include the cache miss counts, obtained from profiling counters on the machine.

5.4. TLB replacement algorithm

We use a pseudo-LRU TLB replacement algorithm for fully-associative TLBs. The algorithm is similar to the "Go Down Stack (GODS)" algorithm described by Deville *et al*. [Devi92]. We associate an *used* bit with every TLB entry that is set on hits to that entry. On a miss: (a) if there are any unfilled (invalid) TLB entries, we choose the first one for replacement; (b) if there are no unfilled TLB entries, we choose the first one with the used bit clear, and (c) if there are no unused TLB entries, we clear all the used bits and retry the algorithm.

5.5. Workloads

Many programs have negligible TLB miss ratios and do not justify the overhead of page promotion required to use superpages. We concentrate on benchmarks where TLB miss handling is a significant part of the execution time, because we expect it to be true for future workloads. **Nasa7, compress, wave5, spice,** and **gcc** are from the SPEC92 suite [SPEC91]; **fftpde** is a NAS benchmark [Bail91] and operates 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** [Appe91] is executing a program that does a stress test on the garbage collector [Repp94].

Table 5 displays benchmark data, with the benchmarks sorted from most to least percent of user time spent on TLB miss handling. Columns two and three give total and user execution time, showing that these benchmarks spend most of their time in user mode. Columns four and five give the number of user TLB misses (for a 64-entry fully-associative single-page-size TLB) and the percent of user time spent servicing these misses (assuming a 40 cycle TLB miss penalty). The data show that user TLB miss handling time is significant. Column six also supports this

Table 5: Workloads

benchmark	total time (seconds)	user time (seconds)	#User TLB misses (thousands) 4KB Single-page-size	% user time for TLB misses (40 cycle penalty)	#(User+Kernel) cache misses (thousands)	Peak memory usage (MB)
coral	177	172	85974	50%	76516	20.5
nasa7	387	385	152357	40%	65356	3.8
compress	104	82	21347	26%	22963	3.4
fftpde	55	53	11280	21%	14472	14.8
wave5	110	107	14511	14%	5082	14.8
mp3d	36	36	4050	11%	5457	5.4
spice	620	617	41922	7%	81949	4.2
pthor	48	35	2580	7%	7456	15.8
ML	950	919	38423	4%	369771	33.6
gcc	159	133	3335	3%	19662	12.5

Table 6: Comparison of 64-entry fully-associative unified TLBs

benchmark	Number of user TLB misses in thousands (Normalized to single-page-size TLB)				
	4KB single - page-size	4KB/64KB Superpage	4KB/64KB Partial-subblock	4KB/64KB Complete-subblock	4KB 256-entry 4-way set-assoc
coral	85974 (100%)	54277 (64.3%)	42647 (49.6%)	41636 (48.4%)	46304 (53.9%)
nasa7	152357 (100%)	14264 (9.4%)	9 (0.0%)	3 (0.0%)	75916 (49.8%)
compress	21347 (100%)	714 (3.3%)	29 (0.1%)	27 (0.1%)	65 (0.3%)
fftpde	11280 (100%)	11201 (99.3%)	10863(96.3%)*	11130(98.7%)*	14923(132.3%)
wave5	14511 (100%)	14 (0.1%)	33 (0.2%)	31 (0.2%)	60 (0.4%)
mp3d	4050 (100%)	13 (0.3%)	46 (1.1%)	38 (0.9%)	551 (13.6%)
spice	41922 (100%)	492 (1.2%)	5 (0.0%)	4 (0.0%)	2429 (5.8%)
pthor	2580 (100%)	2466 (95.6%)	1879 (72.8%)	1737 (67.3%)	1676 (65.0%)
ML	38423 (100%)	21304 (55.4%)	10206 (26.6%)	7103 (18.5%)	15952 (41.5%)
gcc	3335 (100%)	495 (14.8%)	74 (2.2%)	59 (1.8%)	308 (9.2%)

conclusion, showing that some benchmarks have more user TLB misses than user plus system caches misses (with a 1MB direct-mapped cache with 32-byte blocks). TLB misses may be even more important then cache misses, because, in many systems, the TLB miss penalty is larger than the cache miss penalty. Finally, column seven displays peak memory usage, showing that none of these benchmarks paged on our 96MB machine.

6. TLB Performance Study

In this section, we present simulation studies of superpage and subblock TLBs using operating system support from *Foxtrot*. Both types of TLBs use Foxtrot's page reservation (Section 4), while superpage TLBs also require page promotion (Section 5.1). Table 6 shows the number of user TLB misses for the benchmarks using 64-entry fully-associative unified TLBs with a single page size of 4KB, two page sizes of 4KB and 64KB, and partial- and complete-subblocking with a subblock factor of 16. The TLB replacement algorithm is described in Section 5.4. We also include the results for a single-page-size 256-entry 4-way set-associative TLB using random replacement. In parenthesis we normalize the TLB misses with respect to the single-page-size TLB.

6.1. Comparing TLB Misses

The second column of Table 6 demonstrates that using superpages can reduce TLB misses significantly. The SPEC benchmarks and **mp3d** see an order of magnitude reduction in the number of TLB misses. Not shown in this table is that the improvement comes from a few large mappings since only 10%-20% of misses were to superpages. The **ML** and **coral** results show that superpages are effective with very large data sets too. The data also shows that the operating system can implement a good page-size assignment—we did not modify the applications. **Fftpde** and **pthor**, however, show little improvement due to sparse access patterns.

The third column demonstrates that partial-subblock TLBs usually perform significantly better than superpage TLBs for reasons given in Section 3. However, subblock TLBs can have more TLB misses than superpage TLBs, as illustrated by **mp3d** and **wave5**, since it takes multiple TLB misses to load what a superpage can in a single miss.

The fourth column shows that the performance of complete-subblock TLBs is not much better than that of partial-subblock TLBs. This shows that the operating system was very successful at allocating physical memory to support partial-subblock TLBs. Copy-on-write situations, which Foxtrot does not optimize, account for most of the difference between the performance of complete- and partial-subblock TLBs. Thus one can choose between the large TLB size for complete-subblocking and the operating system support for partial-subblocking. We have not yet come up with a convincing explanation for why **fftpde** (flagged with asterisks) performs slightly worse with complete-subblocking than with partial-subblocking.

The brute force method of increasing TLB reach is to build a much larger TLB that supports only a single page size. The key advantage of this approach is that no operating system changes are needed. The disadvantage is that the larger TLB may have an unacceptably large access time and/or chip area. The final column explores this possibility with a 256-entry TLB that uses four -way set-associativity instead of a fully-associative design. Results show that the larger TLB suffers more misses than a 64-entry fully-associative partial-subblock TLB. However , in the absence of operating system support or in the presence of excessive external fragmentation, superpage and partial-subblock TLBs degenerate to a single-page-size TLB. Under these conditions set-associative single-page-size or complete-subblock TLBs should be favored.

The data, so far, assume 64-entry fully-associative TLBs with a superpage size of 64KB or partial-subblocking with subblock factor of 16. Appendix B includes results of sensitivity analysis which show that

Table 7: Fully-associative TLBs with comparable number of user misses

benchmark	4KB Single-page-size		4KB/64KB Superpage		4KB/64KB Partial-Subblock		4KB/64KB Complete-Subblock	
	# entries	area ratio	# entries	area ratio	# entries	area ratio	# entries	area ratio
coral	141	2.0	64	1.0	46	0.9	45	3.2
nasa7	460	6.4	64	1.0	31	0.6	31	2.3
compress	135	1.9	64	1.0	29	0.6	21	1.7
fftpde	16-133	0.3-1.9	64	1.0	48	0.9	50	3.5
wave5	3505	48.3	64	1.0	244	4.1	227	14.7
mp3d	1128	15.6	64	1.0	86	1.5	76	5.2
spice	371	5.2	64	1.0	39	0.7	34	2.6
pthor	78	1.2	64	1.0	31	0.6	28	2.1
ML	115	1.7	64	1.0	42	0.8	33	2.5
gcc	159	2.3	64	1.0	37	0.7	22	1.8

varying the superpage size from 16KB to 64KB (Figure 4), the subblock factor from 2 to 16 (Figure 5), and TLB size from 32 to 256 entries (Table 9) does not qualitatively alter the conclusions.

6.2. Comparing TLB chip areas

The results above assume TLBs with the same number of entries, but require different chip area per entry. Here we size TLBs to get comparable number of TLB misses to see which TLB minimizes chip area [Joup94, Nagl94]. We estimate the chip area required to implement a single-ported TLB using the on-chip cache area model proposed by Mulder *et al*. [Muld91] with the assumptions given in Appendix A. Table 7 gives the number of single-page-size, partial- and complete-subblock TLB entries required to get comparable number of misses to a 64-entry superpage TLB and the corresponding area normalized with respect to the area for the 64-entry superpage TLB. We obtained these numbers by iteratively rerunning our simulation varying the TLB size until the TLB miss counts were comparable. This analysis ignores, however, that operating system overheads and TLB miss penalties can differ significantly.

Table 7 illustrates four results. First, 4KB/64KB superpage TLBs require much less area than 4KB single-page-size TLBs, by ratios of 1.2 to as much as 5 and 48. Second, 4KB/64KB partial-subblock TLBs require less area than superpage TLBs. Further, to their advantage, partial-subblock TLBs have a smaller TLB miss penalty, less operating system overhead and do less I/O. Third, fully-associative complete-subblock TLBs often required smaller area than even the single-page-size TLBs. Since complete-subblock TLBs have a much smaller number of tags, access time advantages may make them an attractive option. Fourth, partial-subblock TLBs occupy a much smaller area than complete-subblock TLBs but require some operating system support.

For a range of single-page-size TLB sizes (16-133 entries) **fftpde** has nearly identical TLB performance. The partial-subblock TLB for **mp3d** and **wave5**, while larger than the superpage TLB, due to multiple TLB misses required to load a superpage, is still significantly smaller than the single-page-size TLB required. Results of sensitivity analysis (T able 10 in Appendix B) show that complete-subblock TLBs with a small subblock factor (*e.g.*, two) are more attractive than a single page size TLB with more entries.

7. Conclusions

Many recent microprocessor architectures specify TLBs that support superpages. Very large superpages (*e.g.*, 1MB) are clearly useful for mapping special structures, such as kernel data and frame buffers. This paper considers medium-sized superpages (*e.g.*, 64KB), but proposes subblock TLBs as an alternative way to improve TLB performance. A *complete-subblock* TLB associates a tag with a superpage-sized region but has valid bits, physical page number, attributes, etc., for each possible base page mapping. A *partial-subblock* TLB entry is much smaller than a complete-subblock TLB entry, because it shares physical page number and attribute fields across base page mappings. Our results show:

- All newer alternatives yield better TLB performance than a conventional TLB supporting a single page size (unless the implementation technology allows all TLBs to be made very large).
- Complete-subblock TLBs yield better performance than conventional TLBs without requiring any operating system changes.
- Partial-subblock TLBs require that the operating system make a *best-effort* to place together the physical pages of most superpage-sized regions. We propose *page reservation* as an algorithm that does this by treating physical memory as a fully-associative subblocked-cache of pages, except in periods of high memory demand. With page reservation, partial subblock TLBs perform comparable to complete subblock TLBs for equal number of TLB entries, but much better for equal TLB areas.
- Finally, superpage TLBs require support that has

an invasive effect on operating system data structures and interfaces that can add considerable overhead. With these changes and using page reservation, superpage TLBs perform well, but not as good as partial subblock TLBs. The reason is that a partial subblock TLB entry can map multiple base pages in situations where the guarantees needed to use superpages are not met (e.g., for unaligned segments, small objects, and non-uniform attributes).

Table 8: Key Results (Oversimplified)

TLB Type	Additiona l OS support	TLB performance with fixed	
		# of entries	chip area
Single-page-size	None	Worst	Worst
Complete-subblock	None	Best	Medium
Partial-subblock	Best-effort	Almost Best	Best
Superpage	Invasive	Good	Good

Table 8 summarizes the key results of this paper. There are three factors that determine TLB performance: operating system support, number of TLB entries (often a function of cycle time), 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 can be modified to support an algorithm like page reservation, partial-subblock TLBs will reduce TLB area or perform better than complete-subblock TLBs. While very large superpages are useful, our 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.

Acknowledgments

We would like to thank David Wood for suggesting trap-based simulation; Steve Barton, Howard Chartock, Joe Eykholt, Yousef Khalidi, Michael Nelson, Bill Shannon, Anil Shivalingiah, Dock Williams, Robert Yung, Deepak Bhagat, Karl Danz, John Entenmann, Don Jaworski, Robert Roessler, Emil Sarpa and others at Sun Microsystems Inc. for their comments, help in the implementation of Foxtrot and help in funding the project; Praveen Seshadri and John Reppy for providing us with the coral and ML benchmarks; Doug Burger, Krishna Kunchithapadam, James Larus, John Mashey, Subbarao Palacharla, Anne Rogers, Guri Sohi, and Guhan Viswanathan for their comments.

Bibliography

[Appe91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Proc. Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, August 1991.

[Bail91] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Report RNR-91-002 Revision 2, Ames Research Center, August 1991.

[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.

[Denn70] Peter J. Denning. Virtual Memory. *Computing Surveys*, 2(3):153–189, September 1970.

[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.

[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.

[Henn90] John L Hennessy and David A Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[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.

[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*, 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.

[Kane92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[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.

[Khal93] 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.

[Knut68] Donald E. Knuth. *The Art of Computer Programming, Volume 1*. Addison Wesley, 1968. Second Printing.

[Lipt68] J. S. Liptay. Structural aspects of the System/360 Model 85, Part II: the cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[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.

[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.

[Nagl94] 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*, April 1994.

[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.

[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.

[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.

[Repp94] John H. Reppy. A High-performance Garbage Collector for Standard ML. AT&T Bell Laboratories Technical Memo, 1994.

[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.

[Site93] Richard L. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

[Smit82] Alan Jay Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[SPAR91] SPARC International Inc. The SPARC Architecture Manual, Version 8, 1991.

[SPEC91] SPEC. (entire issue). *SPEC Newsletter*, 3(4), December 1991.

[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.

[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.

[Uhli94] Richard Uhlig, David Nagle, Trevor Mudge, and Stuart Sechrest. Tapeworm II: A New Method for Measuring OS Effects on Memory Architecture Performance. In *(To appear in) Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

Appendix A: Area Model Assumptions

We made the following assumptions while using the area model suggested for on-chip fully-associative caches by Mulder *et. al.* [Muld91]. The units are *register bit equivalents* (**rbe**).

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

The tag bits include a 12-bit PID and a 52-bit VPN (64-bit virtual address - 12-bit base page offset). In subblock TLBs the VPN is $\log_2$(subblock factor) bits smaller.

The data bits include a 36-bit PPN (48-bit physical address - 12-bit base page offset) and 8 bits of attributes. They also include the modified and valid bits that are one bit each. In partial-subblock TLBs we count the valid bits as tag bits, though they are not true CAM cells. Partial-subblock TLBs have one additional attribute bit (SB).

There is one status bit per TLB entry—the **used** bit (for LRU replacement).

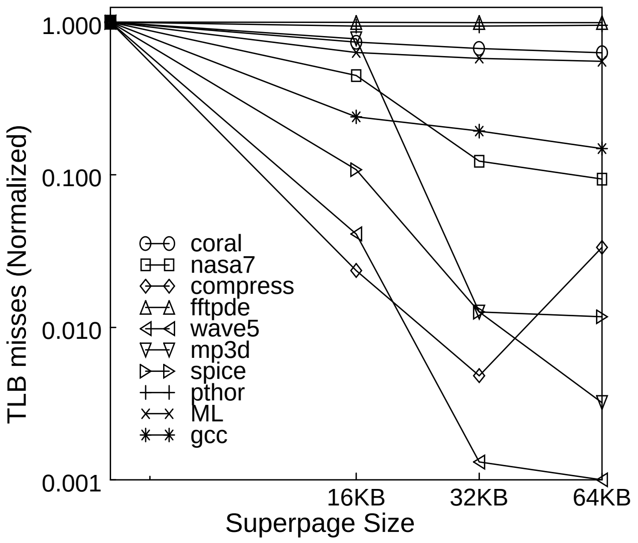
For superpage TLBs, we assume a 4-bit size field in CAM. In implementations, the size field is neither completely in CAM or RAM. It functions as a mask in tag compare and controls physical address generation too.

We use the model's assumptions "as is" about the size of drivers (6), sense amps (6), PLA (130), RAM cells (0.6 rbe), CAM cells (1.2 rbe) and CAM aspect ratio (1:1).

Appendix B: Sensitivity Analysis

Figure 4 shows the effect of varying the superpage size from 16KB to 64KB in a 64-entry fully-associative superpage TLB. We use a log-log scale to accommodate the orders of magnitude reduction in the number of TLB misses. As expected, TLB performance improves as the superpage size is increased. Much of the improvement is due to the heap segments. **Pthor** and **fftpde** thrash all the TLBs, however, all are slightly better than the single-page-size TLB. The large benchmarks, **coral** and **ML**, show a steady improvement in TLB performance with larger superpages but still incur a large number of misses due to insufficient TLB reach in even the superpage TLBs.

Figure 4: Effect of Increasing Superpage Size (64-entry fully-associative superpage TLB)



For **compress** the number of TLB misses decreases at first but increases as the superpage size is increased further. The page-size assignment policy we use causes this anomaly. Four 32KB regions that used 32KB superpages could not be promoted as 64KB superpages because the usage was less than the page promotion threshold for 64KB superpages. The start and end of heap, a part of **libc** and **ld.so** used base pages instead. The degradation is small (the log scale makes it dramatic) with the TLB misses increasing from 103,000 to 714,000 that still compares favorably to the 21 million misses for a single-page-size TLB.

Figure 5 shows the effect of varying the subblock factor from 2 to 16 in a 64-entry fully-associative partial-subblock TLB. We again use a log-log scale. As expected, TLB performance improves with increasing subblock factor. Significant gains occur with a subblock factor above 8. Often, a partial-subblock TLB is better than a superpage TLB with the equivalent superpage size. **Fftpde** shows a small improvement with a subblock factor of 16. **Pthor, coral** and **ML** show a steady

Table 9: Effect of fully-associative TLB size. Number of user TLB misses (in thousands)

bench-mark	4KB Single-page-size				4KB/64KB Superpage				4KB/64KB Partial-subblock			
	32	64	128	256	32	64	128	256	32	64	128	256
coral	112763	85974	63524	36666	67787	54277	38002	23979	70067	42647	25433	3986
nasa7	179903	152357	148312	85895	24304	14264	3	2	10549	9	3	3
compress	52786	21347	818	29	4550	714	28	25	161	29	27	26
fftpde	11288	11280	11279	110	11211	11201	2157	7	11278	10863	46	4
wave5	34463	14511	8680	46	3270	14	9	8	89	33	27	6
mp3d	4943	4050	2395	159	4115	13	2	1	108	46	2	1
spice	203201	41922	7441	818	44987	492	4	3	662	5	3	3
pthor	3775	2580	2216	1862	3627	2466	2055	1245	2420	1879	811	15
ML	83843	38423	19303	11609	59352	21304	7163	1867	32979	10206	3530	951
gcc	18060	3335	663	174	7956	495	62	47	917	74	54	53

improvement in TLB performance with increasing subblock factor but still incur a significant number of TLB misses. **Compress** does not show any anomaly as partial-subblock TLBs do not depend on page promotions.

Figure 5: Effect of increasing subblock factor (64-entry fully-associative partial-subblock TLBs)

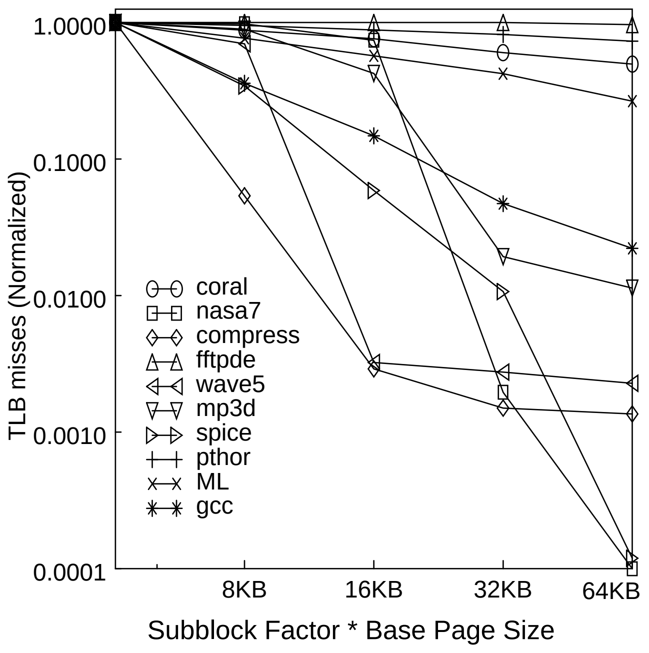


Table 9 shows the effect of varying TLB size. As pointed out in previous studies [Tall92, Chen92], TLB reach is an important factor governing TLB performance. TLB misses occur less often in larger TLBs. The TLB reach of the superpage and partial-subblock TLBs was sufficient to hold the working set of some benchmarks—**nasa7, compress, wave5, mp3d, spice,** and **gcc**—while even a large 256-entry single-page-size TLB was not (except for **compress**). For our benchmarks, a 64-entry partial-subblock TLB often incurs fewer misses than a 256-entry single-page-size TLB. Also, a partial-subblock TLB often performs better than a superpage TLB with the same number of entries. However, superpage and partial-subblock TLBs always are better than an equivalent single-page-size TLB.

Table 10: Comparison of fully-associative TLBs with equal chip areas (Single-page-size and Complete-subblock TLBs)

benchmark	Number of user TLB misses in thousands		
	4KB single - page-size (80-entry)	4KB/8KB Complete-subblock (64-entry)	4KB/16KB Complete-subblock (45-entry)
coral	77651	75251	76402
nasa7	151267	148931	122510
compress	10507	1100	812
fftpde	11278	11279	11280
wave5	14076	10012	875
mp3d	3620	3517	3669
spice	25331	14328	9813
pthor	2453	2418	2439
ML	30292	28114	30721
gcc	1921	1023	793
subblock-factor	0	2	4

In Table 10 we compare the performance of complete-subblock TLBs which occupy comparable chip area to a single-page-size TLB. (as per the area model described in Appendix A) We consider a 80-entry single-page-size TLB, a 64-entry complete-subblock TLB (subblock factor of 2), and a 45-entry complete-subblock TLB (subblock factor of 4). For these benchmarks, the complete-subblock TLB with subblock factor of 2 always performs better than the single-page-size TLB. The complete-subblock with subblock factor of 4 is also often better than the single-page-size TLB. Since the complete-subblock TLBs have a smaller number of tags, access time restrictions may make complete-subblock TLBs even more attractive.