Design Tradeoffs for Software-Managed TLBs

RICHARD UHLIG, DAVID NAGLE, TIM STANLEY, TREVOR MUDGE, STUART SECHREST, and RICHARD BROWN University of Michigan

An increasing number of architectures provide virtual memory support through softwaremanaged TLBs. However, software management can impose considerable penalties that are highly dependent on the operating system's structure and its use of virtual memory. This work explores software-managed TLB design tradeoffs and their interaction with a range of monolithic and microkernel operating systems. Through hardware monitoring and simulation, we explore TLB performance for benchmarks running on a MIPS R2000-based workstation running Ultrix, OSF/1, and three versions of Mach 3.0.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles—associative memories; cache memories; virtual memory; B.3.3. [Memory structures]: Performance Analysis and Design Aids—simulation; C.4 [Computer Systems Organization]: Performance of Systems—measurement techniques; D.4.2 [Operating Systems]: Storage Management—virtual memory; D.4.8 [Operating Systems]: Performance—measurements

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Hardware monitoring, simulation, translation lookaside buffer (TLB), trap-driven simulation

1. INTRODUCTION

Many computers support virtual memory by providing hardware-managed translation lookaside buffers (TLBs). However, beginning with the ZS-1 in 1988 [Smith et al. 1988], an increasing number of computer architectures, including the AMD 29050 [Advanced Micro Devices 1991], the HP-PA [Hewlett-Packard 1990], and the MIPS RISC [Kane and Heinrich 1992], have shifted TLB management responsibility into the operating system. These software-managed TLBs simplify hardware design and provide greater flexi-

© 0734-2071/94/0800-0175\$03.50

ACM Transactions on Computer Systems, Vol. 12, No 3, August 1994, Pages 175-205.

A shorter version of this article appeared in the *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, May 1993.

This work was supported by Defense Advanced Research Projects Agency under DARPA/ARO contract DAAL03-90-C-0028, by National Science Foundation CISE Research Instrumentation grant CDA-9121887, and a National Science Foundation Graduate Fellowship.

Authors' address: Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

bility in page table structure, but typically have slower refill times than hardware-managed TLBs [DeMoney et al. 1986].

At the same time, operating systems such as Mach 3.0 [Accetta et al. 1986] are moving functionality into user processes and making greater use of virtual memory for mapping kernel data structures held within the kernel. These and related operating system trends can increase TLB miss rates and, hence, decrease overall system performance.

This article is a case study, exploring the impact of TLBs on the performance of application and operating system code running on a MIPS R2000based workstation. In particular, we examine the differences in performance seen when running the same applications under different operating systems, showing how these performance differences reflect the construction of the operating system code. Through simulation, we explore alternative TLB configurations. The results illustrate the design tradeoffs for a particular software-managed TLB in conjunction with particular operating system implementations. As a case study, this work seeks to illuminate the broader problems of interaction between operating system software and architectural features.

This work is based on measurement and simulation of running systems. To examine issues that cannot be adequately modeled with simulation, we have developed a system analysis tool called Monster, which enables us to monitor running machines. We have also developed a novel TLB simulator called Tapeworm, which is compiled directly into the operating system so that it can intercept all TLB misses caused by both user process and OS kernel memory references. The information that Tapeworm extracts from the running system is used to obtain TLB miss counts and to simulate different TLB configurations.

The remainder of this article is organized as follows: Section 2 examines previous TLB and OS research related to this work. Section 3 describes our analysis tools, Monster and Tapeworm. The MIPS R2000 TLB structure and its performance under Ultrix, OSF/1, and Mach 3.0 are explored in Section 4. Hardware- and software-based performance improvements are presented in Section 5. Section 6 summarizes our conclusions.

2. RELATED WORK

By caching page table entries, TLBs greatly speed up virtual-to-physical address translations. However, memory references that require mappings not in the TLB result in misses that must be serviced either by hardware or by software. Clark and Emer [1985] examined the cost of hardware TLB management by monitoring a VAX-11/780. For their workloads, 5 to 8% of a user program's run-time was spent handling TLB misses.

More recent papers have investigated the TLB's impact on user program performance. Using traces generated from the SPEC benchmarks, Chen, et al. [1992] showed that, for a reasonable range of page sizes, the amount of the address space that could be mapped was more important in determining the TLB miss rate than was the specific choice of page size. Talluri et al.

[1992] showed that while older TLBs (such as that in the VAX 11/780) mapped large regions of physical memory, TLBs in newer architectures, including MIPS RISC, do not. They showed also that increasing the page size from 4 to 32 Kbytes could significantly decrease the TLB's contribution to CPI.

Operating system references also have a strong impact on TLB miss rates. Clark and Emer [1985] showed that although only 18% of all memory references in the system they examined were made by the operating system, these references were responsible for 70% of the TLB misses. Several recent papers [Anderson et al. 1991; Ousterhout 1989; Welch 1991] have noted that changes in the structure of operating systems are altering the utilization of the TLB. For example, Anderson et al. compared an old-style monolithic implementation of an operating system (Mach 2.5) and a newer microkernel implementation of that operating system (Mach 3.0) on a MIPS R3000-based machine, which requires software management of TLBs. Anderson et al. found a 600% increase in the incidence of the most expensive category of TLB misses. Moreover, the handling of these expensive misses was far and away the most frequently invoked system primitive for the Mach 3.0 kernel.

This article differs from previous work through its focus on softwaremanaged TLBs and its examination of the impact of changing operating system technology on TLB design. We trace the specific sources of performance difficulties and the options for eliminating them. The design tradeoffs for software-managed TLBs can be complex, when, as in the systems examined here, there is significant variance in the refill penalty. While hardwaremanaged TLBs have relatively small refill penalties, with low variance, our measurements show that the cost of handling a single TLB miss on a DECstation 3100 running Mach 3.0 can vary from 20 to more than 400 cycles. The different service times reflect the varying lengths of the code paths that handle different types of misses. The particular mix of TLB miss types is highly dependent on the construction of the operating system. We, therefore, focus on the operating system in our analysis and discussion.

3. ANALYSIS TOOLS AND EXPERIMENTAL ENVIRONMENT

To monitor and analyze TLB behavior for benchmark programs running on a variety of operating systems, we have developed a hardware-monitoring system called Monster and a TLB simulator called Tapeworm. The remainder of this section describes these tools and the experimental environment in which they are used.

3.1 System Monitoring with Monster

The Monster monitoring system (Figure 1) enables comprehensive analyses of the interaction between operating systems and architectures. Monster is comprised of a monitored DECstation 3100, an attached logic analyzer, and a controlling workstation. The logic analyzer component of Monster contains a programmable hardware state machine and a 128K-entry trace buffer. The state machine includes pattern recognition hardware that can sense the



Fig 1. The Monster monitoring system. Monster is a hardware-monitoring system consisting of a Tektronix DAS 9200 Logic Analyzer and a DECstation 3100 running three operating systems: Ultrix, OSF/1, and Mach 3.0. The DECstation motherboard has been modified to provide access to the CPU pins, which he between the processor and the cache. This allows the logic analyzer to monitor virtually all system activity. To enable the logic analyzer to tragger on certain operating systems events, such as the servicing of a TLB miss, each operating system has been instrumented with special marker NOP instructions that indicate the entry and exit points of various routines.

processor's address, data, and control signals on every clock cycle. This state machine can be programmed to trigger on predefined patterns appearing at the CPU bus and then store these signals and a timestamp (with 1 ns resolution) into the trace buffer. Monster's capabilities are described more completely in Nagle et al. [1992].

In this article, we used Monster to measure TLB miss-handling costs by instrumenting each OS kernel with marker instructions denoting entry and exit points of various code segments. The instrumented kernels were then monitored with the logic analyzer, whose state machine detected and stored the marker instructions and a nanosecond resolution timestamp into the logic analyzer's trace buffer. Once filled, the trace buffer was postprocessed to obtain a histogram of time spent in the different invocations of the TLB miss handlers. This technique allowed us to time single executions of code paths with far greater accuracy than can be obtained using the coarser-resolution system clock. It also avoids the problems inherent in the common method of improving system clock resolution by taking averages over repeated invocations [Clapp et al. 1986]. Note that this measurement technique is not limited to processors with off-chip caches. Externally observable markers can be implemented on a processor with on-chip caches through noncached memory accesses.

3.2 TLB Simulation with Tapeworm

Many previous TLB studies have used trace-driven simulation to explore design tradeoffs [Alexander et al. 1985; Clark and Emer 1985; Talluri et al. 1992]. However, there are a number of difficulties with trace-driven TLB simulation. First, it is difficult to obtain accurate traces. Code annotation tools like pixie [Smith 1991] or AE [Larus 1990] generate user-level address

traces for a single task. However, more complex tools, such as hardware monitors [Agarwal et al. 1988; Clark and Emer 1985] or annotated kernels [Chen 1993], are required to obtain realistic system-wide address traces that account for both multiprocess workloads and the operating system itself. Second, trace-driven simulation can consume considerable processing and storage resources. Some researchers have overcome the storage resource problem by consuming traces on-the-fly [Agarwal et al. 1988; Chen et al. 1992]. This technique requires that system operation be suspended for extended periods of time while the trace is processed, thus introducing distortion at regular intervals. Third, trace-driven simulation assumes that address traces are invariant to changes in the structural parameters or management policies of a simulated TLB. While this may be true for cache simulation (where misses are serviced by hardware state machines), it is not true for software-managed TLBs where a miss (or absence thereof) directly changes the stream of instruction and data addresses flowing through the processor. Because the code that services a TLB miss can itself induce a TLB miss, the interaction between a change in TLB structure and the resulting system address trace can be quite complex.

We have overcome these problems by compiling our TLB simulator, Tapeworm, directly into the OSF/1 and Mach 3.0 operating system kernels. Tapeworm relies on the fact that all TLB misses in an R2000-based DECstation 3100 are handled by software. We modified the operating systems' TLB miss handlers to call the Tapeworm code via procedural "hooks" after every miss. This mechanism passes the relevant information about all user and kernel TLB misses directly to the Tapeworm simulator. Tapeworm uses this information to maintain its own data structures and to simulate other possible TLB configurations.

A simulated TLB can be either larger or smaller than the actual TLB. Tapeworm ensures that the actual TLB only holds entries available in the simulated TLB. For example, to simulate a TLB with 128 slots using only 64 actual TLB slots (Figure 2), Tapeworm maintains an array of 128 virtual-tophysical address mappings and checks each memory reference that misses the actual TLB to determine if it would have also missed the larger, simulated one. Thus, Tapeworm maintains a strict inclusion property between the actual and simulated TLBs. Tapeworm controls the actual TLB management policies by supplying placement and replacement functions called by the operating system miss handlers. It can simulate TLBs with fewer entries than the actual TLB by providing a placement function that never utilizes certain slots in the actual TLB. Tapeworm uses this same technique to restrict the associativity of the actual TLB.1 By combining these policy functions with adherence to the inclusion property, Tapeworm can simulate the performance of a wide range of different-sized TLBs with different degrees of associativity and a variety of placement and replacement policies.

¹The actual R2000 TLB is fully associative, but varying degrees of associativity can be emulated by using certain bits of a mapping's virtual page number to restrict the slot (or set of slots) into which the mapping may be placed.

ACM Transactions on Computer Systems, Vol. 12, No. 3, August 1994



Fig. 2. Tapeworm. The Tapeworm TLB simulator is built into the operating system and is invoked whenever there is a real TLB miss. The simulator uses the real TLB misses to simulate its own TLB configuration(s). Because the simulator resides in the operating system, Tapeworm captures the dynamic nature of the system and avoids the problems associated with simulators driven by static traces.

The Tapeworm design avoids many of the problems with trace-driven TLB simulation cited above. Because Tapeworm is driven by procedure calls within the OS kernel, it does not require address traces at all; the various difficulties with extracting, storing, and processing large address traces are completely avoided. Because Tapeworm is invoked by the machine's actual TLB miss-handling code, it considers the impact of all TLB misses whether they are caused by user-level tasks or the kernel itself. The Tapeworm code and data structures are placed in unmapped memory and therefore do not distort simulation results by causing additional TLB misses. Finally, because Tapeworm changes the structural parameters and management policies of the actual TLB, the behavior of the system itself changes automatically, thus avoiding the distortion inherent in fixed traces.

3.3 Experimental Environment

All experiments were performed on a DEC station 3100^2 running three different base operating systems (Table I): Ultrix, OSF/1, Mach 3.0. Each of these systems includes a standard Unix file system (UFS) [McKusick et al. 1984]. Two additional versions of Mach 3.0 include the Andrew file system (AFS) cache manager [Satyanarayanan 1990]. One version places the AFS cache manager in the Mach Unix Server (AFSin) while the other migrates the AFS cache manager into a separate server task (AFSout).

To obtain measurements, all of the operating systems were instrumented with counters and markers. For TLB simulation, Tapeworm was embedded in the OSF/1 and Mach 3.0 kernels.

 $^{^2\}mathrm{The}$ DEC station 3100 contains an R2000 microprocessor (16.67 MHz) and 16 Megabytes of memory.

ACM Transactions on Computer Systems, Vol 12, No. 3, August 1994

Benchmark	Description
compress	Uncompresses and compresses a 7.7 Megabyte video clip.
IOzone	A sequential file I/O benchmark that writes and then reads a 10 Megabyte file. Written by Bill Norcott.
jpeg_play	The xloadimage program written by Jim Frost. Displays four JPEG images.
mab	John Ousterhout's Modified Andrew Benchmark [18].
mpeg_play	mpeg_play V2.0 from the Berkeley Plateau Research Group. Displays 610 frames from a compressed video file [19].
ousterhout	John Ousterhout's benchmark suite from [18].
video_play	A modified version of mpeg_play that displays 610 frames from an uncompressed video file.
Operating System	Description
Ultrix	Version 3.1 from Digital Equipment Corporation.
OSF/1	OSF/1 1 0 is the Open Software Foundation's version of Mach 2.5.
Mach 3.0	Carnagie Mellon University's version mk77 of the kernel and uk38 of the UNIX server.
Mach3+AFSin	Same as Mach 3.0, but with the AFS cache manager (CM) run- ning in the UNIX server.
Mach3+AFSout	Same as Mach 3.0, but with the AFS cache manager running as a separate task outside of the UNIX server. Not all of the CM func- tionality has been moved into this server task.

Table I. Benchmarks and Operating Systems.

Benchmarks were compiled with the Ultrix C compiler version 2.1 (level-2 optimization). Inputs were tuned so that each benchmark takes approximately the same amount of time to run (100–200 seconds under Mach 3.0) All measurements cited are the average of three runs.

Throughout this article we use the benchmarks listed in Table I. The same benchmark binaries were used on all the operating systems. Each measurement cited in this article is the average of three trials.

4. OS IMPACT ON SOFTWARE-MANAGED TLBS

Operating system references have a strong influence on TLB performance. Yet, few studies have examined these effects, with most confined to a single operating system [Clark and Emer 1985]. However, differences between operating systems can be substantial. To illustrate this point, we ran our benchmark suite on each of the operating systems listed in Table I. The results (Table II) show that although the same application binaries were run on each system, there is significant variance in the number of TLB misses and total TLB service time. Some of these increases are due to differences in the functionality between operating systems (i.e., UFS vs. AFS). Other increases are due to the structure of the operating systems. For example, the monolithic Ultrix kernel spends only 11.82 seconds handling TLB misses, while the microkernel-based Mach 3.0 spends 80.01 seconds.

182 • Richard Uhlig et al

Operating System	Run Time (sec)	Total Number of TLB Misses	Total TLB Service Time (sec)*	Ratio to Ultrix TLB Service Time
Ultrix 3.1	583	9,177,401	11.82	1.0
OSF/1	892	11,691,398	51.85	4.39
Mach 3.0	975	24,349,121	80.01	6.77
Mach3+AFSin	1,371	33,933,413	106.56	9.02
Mach3+AFSout	1,517	36,649,834	134.71	11.40

Table II. Total TLB Misses across the Benchmarks.

*Time based on measured median time to service TLB miss.

The total run-time and number of TLB misses incurred by the seven benchmark programs Although the same application binaries were run on each of the operating systems, there is a substantial difference in the number of TLB misses and their corresponding service times.

Notice that while the total number of TLB misses increases four fold (from 9,177,401 to 36,639,834 for AFSout), the total time spent servicing TLB misses increases 11.4 times. This is due to the fact that there are different types of software-managed TLB misses, each with its own associated cost. For this reason, it is important to understand page table structure, its relationship to TLB miss handling, and the frequencies and costs of different types of misses.

4.1 Page Tables and Translation Hardware

OSF/1 and Mach 3.0 both implement linear page table structures (Figure 3). Each task has its own level-1 (L1) page table, which is maintained by machine-independent pmap code [Rashid et al. 1988]. Because the user page tables can require several megabytes of space, they are themselves stored in the virtual address space. This is supported through level-2 (L2 or kernel) page tables, which also map other kernel data. Because kernel data is relatively large and sparse, the L2 page tables are also mapped. This gives rise to a 3-level page table hierarchy and four different page table entry (PTE) types.

The R2000 processor contains a 64-slot, fully associative TLB, which is used to cache recently used PTEs. When the R2000 translates a virtual address to a physical address, the relevant PTE must be held by the TLB. If the PTE is absent, the hardware invokes a trap to a software TLB miss-handling routine that finds and inserts the missing PTE into the TLB. The R2000 supports two different types of TLB miss vectors. The first, called the *user TLB* (uTLB) vector, is used to trap on missing translations for L1U pages. This vector is justified by the fact that TLB misses on L1U PTEs are typically the most frequent [DeMoney et al. 1986]. All other TLB miss types (such as those caused by references to kernel pages, invalid pages, or read-only pages) and all other interrupts and exceptions trap to a second vector, called the *generic-exception* vector.



Fig. 3. Page table structure in OSF/1 and Mach 3.0. The Mach page tables form a 3-level structure with the first two levels residing in virtual (mapped) space. The top of the page table structure holds the user pages which are mapped by level-1 user (L1U) PTEs. These L1U PTEs are stored in the L1 page table with each task having its own set of L1 page tables.

Mapping the L1 page tables are the level-2 (L2) PTEs. They are stored in the L2 page tables which hold both L2 PTEs and level-1 kernel (L1K) PTEs. In turn, the L2 pages are mapped by the level-3 (L3) PTEs stored in the L3 page table. At boot time, the L3 page table is fixed in unmapped physical memory. This serves as an anchor to the page table hierarchy because references to the L3 page table do not go through the TLB.

The MIPS R2000 architecture has a fixed 4 KByte page size. Each PTE requires 4 bytes of storage. Therefore, a single L1 page table page can hold 1,024 L1U PTEs, or 4 Megabytes of virtual address space. Likewise, the L2 page tables can directly map either 4 Megabytes of kernel data or indirectly map 4 GBytes of L1U data.

For the purposes of this case study, we define TLB miss types (Table III) to correspond to the linear page table structure implemented by OSF/1 and Mach 3.0.³ In addition to L1U TLB misses, we define five subcategories of kernel TLB misses (L1K, L2, L3, modify, and invalid). Table III shows also our measurements of the time required to handle the different types of TLB misses. The wide differential in costs is primarily due to the two different miss vectors and the way that the OS uses them. L1U PTEs can be retrieved within 16 cycles because they are serviced by a highly tuned handler inserted at the uTLB vector. However, other miss types require anywhere from about

³Other page table structures are possible and could lead to a different set of definitions and conclusions. Huck and Hays [1993] have examined the impact of forward-mapped, inverted, and hashed page table structures on TLB miss-handling times.

ACM Transactions on Computer Systems. Vol 12, No 3, August 1994.

184 • Richard Uhlig et al.

TLB Miss Type	Ultrix	OSF/1	Mach 3.0
L1U	16	20	20
L1K	333	355	294
L2	494	511	407
L3		354	286
Modify	375	436	499
Invalid	336	277	267

Table III. Costs for Different TLB Miss Types.

This table shows the number of machine cycles (at 60 ns/cycle) required to service different types of TLB misses. To determine these costs, Monster was used to collect a 128K-entry histogram of timings for each type of miss. We separate TLB miss types into the six categories described below. Note that Ultrix does not have L3 misses because it implements a 2-level page table.

L1U TLB miss on a level-1 user PTE.

L1K TLB miss on a level-1 kernel PTE

L2 TLB miss on level-2 PTE. This can only occur after a miss on a level-1 user PTE.

L3 TLB miss on a level-3 PTE Can occur after either a level-2 miss or a level-1

kernel miss.

Modify A page protection violation.

Invalid An access to a page marked as invalid (page fault).

300 to over 500 cycles because they are serviced by the generic handler residing at the generic-exception vector.

The R2000 TLB hardware supports partitioning of the TLB into two sets of slots. The lower partition is intended for PTEs with high retrieval costs, while the upper partition is intended to hold more frequently used PTEs that can be refetched quickly (e.g., L1U) or infrequently referenced PTEs (e.g., L3). The TLB hardware also supports random replacement of PTEs in the upper partition through a hardware index register that returns random numbers in the range 8 to 63. This, effectively, fixes the TLB partition at 8, so that the lower partition consists of slots 0 through 7, while the upper partition consists of slots 8 through 63.

4.2 OS Influence on TLB Performance

In the versions of the operating systems studied, there are three basic factors which account for the variation in the number of TLB misses and their associated costs (Table IV and Figure 4). They are: (1) the use of mapped memory by the kernel (both for page tables and other kernel data structures); (2) the placement of functionality within the kernel, with a user-level server process (service migration), or divided among several server processes (OS decomposition); and (3) the range of functionality provided by the system (additional OS services). The rest of this section uses our data to examine the relationship between these OS characteristics and TLB performance.

4.2.1 *Mapping Kernel Data Structures*. Unmapped portions of the kernel's address space do not rely on the TLB. Mapping kernel data structures, therefore, adds a new category of TLB misses: L1K misses. For the operating

OS	Mapped Kernel Data Structures	Service Migration	Service Decomposition	Additional OS Services
Ultrix	Few	None	None	X Server
OSF/1	Many	None	None	X Server
Mach 3.0	Some	Some	Some	X Server
Mach3+AFSin	Some	Some	Some	X Server & AFS CM
Mach3+AFSout	Some	Some	Many	X Server & AFS CM

Table IV. Characteristics of the Operating Systems Studied.

systems we examined, an increase in the number of L1K misses can have a substantial impact on TLB performance because each L1K miss requires several hundred cycles to service.

Ultrix places most of its data structures in a small, fixed portion of unmapped memory that is reserved by the OS at boot time. However, to maintain flexibility, Ultrix can draw upon the much larger virtual space if it exhausts this fixed-size unmapped memory. Table V shows that few L1K misses occur under Ultrix.

In contrast, OSF/1 and Mach 3.0^4 place most of their kernel data structures in mapped virtual place, forcing them to rely heavily on the TLB. Both OSF/1 and Mach 3.0 mix the L1K PTEs and L1U PTEs in the TLB's 56 upper slots. This contention produces a large number of L1K misses. Further, handling an L1K miss can result in an L3 miss.⁵ In our measurements, OSF/1 and Mach 3.0 both incur more than 1.5 million L1K misses. OSF/1 must spend 62% of its TLB handing time servicing these misses while Mach 3.0 spends 37% of its TLB handling time servicing L1K misses.

4.2.2 Service Migration. In a traditional operating system kernel such as Ultrix or OSF/1 (Figure 4), all OS services reside within the kernel, with only the kernel's data structures mapped into the virtual space. Many of these services, however, can be moved into separate server tasks, increasing the modularity and extensibility of the operating system [Anderson et al. 1991]. For this reason, numerous microkernel-based operating systems have been developed in recent years (e.g., Chorus [Dean and Armand 1991], Mach 3.0 [Accetta 1986], and V [Cheriton 1984]).

By migrating these services into separate user-level tasks, operating systems like Mach 3.0, fundamentally, change the behavior of the system for two reasons. First, moving OS services into user space requires, on the MIPS RISC architecture, that both their program text and their data structures be mapped. On other architectures that provide a small number of entries to

⁴Like Ultrix, Mach 3.0 reserves a portion of unmapped space for dynamic allocation of data structures. However, it appears that Mach 3.0 uses this unmapped space quickly and must begin to allocate mapped memory. Once Mach 3.0 has allocated mapped space, it does not distinguish between mapped and unmapped space despite their differing costs.

^bL1K PTEs are stored in the mapped, L2 page tables (Figure 3).

ACM Transactions on Computer Systems, Vol. 12, No. 3, August 1994.



user-level server (AFSout)

		Table			co.			
System	Total Run Time (sec)	L1U	L1K	L2	Г3	Invalid	Modify	Total
Ultrix	583	9,021,420	135,847	3,828		16,191	115	9,177,401
OSF/1	892	9,817,502	1,509,973	34,972	207,163	79,299	42,490	11,691,398
Mach3	975	21,466,165	1,682,722	352,713	556,264	165,849	125,409	24,349,121
Mach3+AFSin	1,371	30,123,212	2,493,283	330,803	690,441	168,429	127,245	33,933,413
Mach3+AFSout	1,517	31,611,047	2,712,979	1,042,527	987,648	168,128	127,505	36,649,834

Ξ
TLB
of
Number
Ņ.
able

ACM Transactions on Computer Systems, Vol. 12, No. 3, August 1994.

188 • Richard Uhlig et al.

map large blocks, such as the entire kernel, these entries may be exhausted [Digital 1992; Hewlett-Packard 1990; Kane and Heinrich 1992; Motorola 1990; 1993]. In either case, migrated services will have to share more of the TLB with user tasks, possibly conflicting with the user tasks' TLB footprints. Comparing the number of L1U misses in OSF/1 and Mach 3.0, we see a 2.2-fold increase from 9.8 million to 21.5 million. This is directly due to moving OS services into mapped user space. The second change comes from moving OS data structures from mapped kernel space to mapped user space. In user space, the data structures are mapped by L1U PTEs, which are handled by the fast uTLB handler (20 cycles for Mach 3.0). In contrast, the same data structures in kernel space are mapped by L1K PTEs, which are serviced by the general-exception handler.

4.2.3 Operating System Decomposition. Moving OS functionality into a monolithic Unix server does not achieve the full potential of a microkernelbased operating system. Operating system functionality can be further decomposed into individual server tasks. The resulting system is more flexible and can provide a higher degree of fault tolerance.

Unfortunately, experience with fully decomposed systems has shown severe performance problems. Anderson et al. [1991] compared the performance of a monolithic Mach 2.5 and a microkernel Mach 3.0 operating system with a substantial portion of the file system functionality running as a separate AFS cache manager task. Their results demonstrate a significant performance gap between the two systems with Mach 2.5 running 36% faster than Mach 3.0, despite the fact that only a single additional server task is used. Later versions of Mach 3.0 have overcome this performance gap by integrating the AFS cache manager into the Unix server.

We compared our benchmarks running on the Mach3+AFSin system against the same benchmarks running on the Mach3+AFSout system. The only structural difference between the systems is the location of the AFS cache manager. The results (Table V) show a substantial increase in the number of both L2 and L3 misses. Many of the L3 misses are due to missing mappings needed to service L2 misses.

The L2 PTEs compete for the R2000's 8 lower TLB slots. Yet, the number of L2 slots required is proportional to the number of tasks providing an OS service concurrently. As a result, adding just a single, tightly coupled service task overloads the TLB's ability to map L2 page tables. Thrashing results. This increase in L2 misses will grow ever more costly as systems continue to decompose services into separate tasks.

4.2.4 Additional OS Functionality. In addition to OS decomposition and migration, many systems provide supplemental services (e.g., X, AFS, NFS, Quicktime). Each of these services, when interacting with an application, can change the operating system behavior and how it interacts with the TLB hardware.

For example, adding a distributed file service (in the form of an AFS cache manager) to the Mach 3.0 Unix server adds 10.39 seconds to the L1U TLB miss-handling time (Table VI). This is due solely to the increased

		Table VI.	Time Spent	Handling 11	B Misses.			
System	Total TLB Service Time (sec)	LIU	L1K	Г2	٦	Invalid	Modify	% of Run Time
Ultrix	11.82	8.66	2.71	0.11		0.33	00.0	2.03%
OSF/1	51.85	11.78	32.16	1.07	4.40	1.32	1.11	5.81%
Mach3	80.01	25.76	29.68	8.61	9.55	2.66	3.75	8.21%
Mach3+AFSin	106.56	36.15	43.98	8.08	11.85	2.70	3.81	7.77%
Mach3+AFSout	134.71	37.93	47.86	25.46	16.95	2.69	3.82	8.88%
Tables V and VI sh systems studied. In OSF/1 and variou misses is due to OS	tow the number 1 Ultrix, most of Miss versions of Miss F/1 and Mach	of TLB miss of the TLB n ach 3.0, L1K 3.0's use of p	ses and amou iisses and T and L2 mis protection to	LB miss time s ses can overs implement co	pent handling is spent ser shadow the L py-on-write 1	g TLB misses vicing L1U ⁷ .1U miss tum nemory sharr	t for each of t TLB misses. e. The increa	he operating However, for se in Modify

Mis
TLB
Handling
Spent
Time
VI.
Table

190 · Richard Uhlig et al

functionality residing in the Unix server. However, L1K misses also increase, adding 14.3 seconds. These misses are due to the additional memory management the Mach 3.0 kernel must provide for the AFS cache manager. Increased functionality will have an important impact on how architectures support operating systems and to what degree operating systems can increase and decompose functionality.

5. IMPROVING TLB PERFORMANCE

This section examines both hardware- and software-based techniques for improving TLB performance. However, before we suggest changes, it is helpful to review the motivations behind the design of the R2000 TLB, described in Section 4.1.

The MIPS R2000 TLB design is based on two principal assumptions [DeMoney et al. 1986]. First, L1U misses are assumed to be the most frequent (> 95%) of all TLB miss types. Second, all OS text and most of the OS data structures (with the exception of user page tables) are assumed to be unmapped. The R2000 TLB design reflects these assumptions by providing two types of TLB miss vectors: the fast uTLB vector and the much slower general-exception vector. These assumptions are also reflected in the partitioning of the 64 TLB slots into the two disjoint sets of 8 lower slots and 56 upper slots. The 8 lower slots are intended to accommodate a traditional Unix task (which requires at least three L2 PTEs) and Unix kernel (2 PTEs for kernel data), with three L2 PTEs left for additional data segments.

Our measurements (Table V) demonstrate that these design choices make sense for a traditional Unix operating system such as Ultrix. For Ultrix, L1U misses constitute 98.3% of all misses. The remaining miss types impose only a small penalty. However, these assumptions break down for the OSF/1- and Mach 3.0-based systems. In these systems, the costly non-L1U misses account for the majority of time spent handling TLB misses. Handling these misses substantially increases the cost of software-TLB management (Table VI).

The rest of this section proposes and explores four ways in which softwaremanaged TLBs can be improved. First, the cost of certain types of TLB misses can be reduced by modifying the TLB vector scheme. Second, the number of L2 misses can be reduced by increasing the number of lower slots.⁶ Third, the frequency of most types of TLB misses can be reduced if more total TLB slots are added to the architecture. Finally, we examine the tradeoffs between TLB size and associativity.

5.1 Reducing Miss Costs

The data in Table V show a significant increase in L1K misses for OSF/1 and Mach 3.0 when compared against Ultrix. This increase is due to both systems' reliance on dynamic allocation of kernel mapped memory. The R2000's TLB performance suffers because L1K misses must be handled by the costly generic-exception vector, which requires 294 cycles (Mach 3.0).

⁶The newer MIPS R4000 processor [Kane and Heinrich 1992] implements both of these changes

ACM Transactions on Computer Systems, Vol. 12, No 3, August 1994

Type of PTE Miss	Previous Miss Costs	New Miss Costs	Counts	Previous Total Cost from Table 6 (sec)	New Total Cost (sec)	Time Saved (sec)
L1U	20	20	30,123,212	36.15	36.15	0.00
L2	294	20	330,803	8.08	0.79	7.29
L1K	407	40	2,493,283	43.98	2.99	40.99
L3	286	286	690,441	11.85	11.85	0.00
Modify	499	499	127,245	3.81	3.81	0.00
Invalid	267	267	168,429	2.70	2.70	0.00
	Total		33,933,413	106.56	58.29	48.28

Table VII. Benefits of Reduced TLB Miss Costs

This table shows the benefits of reducing TLB miss costs through the hardware-based approach of adding a separate interrupt vector for L2 misses and allowing the uTLB handler to service L1K misses This change reduces their cost to 40 and 20 cycles, respectively. Their contribution to TLB miss time drops from 8.08 and 43 98 seconds down to 0.79 and 2.99 seconds, respectively.

These miss costs can be reduced either through better hardware support or through more careful coding of the software miss handlers. For example, hardware support could consist of additional vectors for L1K and L2 misses. Based on our timing and analysis of the TLB handlers, we estimate that vectoring the L1K misses through the uTLB handler would reduce the cost of L1K misses from 294 cycles (for Mach 3.0) to approximately 20 cycles. We also estimate that dedicating a separate TLB miss vector for L2 misses would decrease their cost from 407 cycles (Mach 3.0) to under 40 cycles. Better hardware support could also come in the form of a hybrid hardware/software miss handler. For example, some systems implement the first step of a TLB miss-handling algorithm in hardware and invoke a software handler to implement the remaining steps of the algorithm only when necessary [Huck and Hays 1993]. Such a scheme retains the benefits of software handling (e.g., flexible page table structure and reduced hardware complexity relative to systems with full table-walking hardware), but the average cost of TLB misses is reduced. Alternatively, a software solution could test for L2 PTE misses at the *beginning* of the generic-exception vector before invocation of the code that saves register state and allocates a kernel stack.

Table VII shows the benefits of adding additional vectors. It uses the same data for Mach3 + AFSin as shown in Table V, but is recomputed with the new cost estimates resulting from the refinements above. The result of combining these two modifications is that the total TLB miss service time drops from 106.56 seconds down to 58.29 seconds. L1K service drops 93%, and L2 miss service time drops 90%. More importantly, the L1K and L2 misses no longer contribute substantially to overall TLB service time. This minor design modification enables the TLB to support much more effectively a microker-nel-style operating system with multiple servers in separate address spaces.

192 • Richard Uhlig et al.

Multiple TLB miss vectors provide additional benefits. In the generic trap handler, dozens of load and store instructions are used to save and restore a task's context. Many of these loads and stores cause cache misses requiring the processor to stall. As processor speeds continue to outstrip memory access times, the CPI in this save/restore region will grow, increasing the number of wasted cycles and making non-uTLB misses much more expensive. TLBspecific miss handlers will not suffer the same performance problems because they contain only the single data reference to load the missed PTE from the memory-resident page tables.

5.2 Partitioning the TLB

The MIPS R2000 TLB design mandates a partition of the TLB entries and fixes its position between the 8 lower slots and the 56 upper slots. On other architectures, it is possible to lock entries in the TLB, which can have much the same effect as a partition, but with more flexibility in choosing the number of slots to be locked [Milenkovic 1990; Motorola 1990]. The MIPS R2000 partitioning is appropriate for an operating system like Ultrix [DeMoney et al. 1986]. However, as OS designs migrate and decompose functionality into separate user-space tasks, having only 8 lower slots becomes insufficient. This is because in a decomposed system the OS services that reside in different user-level tasks compete by displacing each other's L2 PTE mappings from the TLB.

In the following sections, we examine the influences of operating system structure, workload, PTE placement policy, and PTE replacement policy on the optimal partition point. We then propose an adaptive mechanism for adjusting dynamically the partition point.

5.2.1 Influences on the Optimal Partition Point. To better understand the impact of the position of the partition point, we measured how L2 miss rates vary depending on the number of lower TLB slots available. Tapeworm was used to vary the number of lower TLB slots from 4 to 16, while the total number of TLB slots was kept fixed at 64. OSF/1 and all three versions of Mach 3.0 ran the mab benchmark over the range of configurations, and the total number of L2 misses was recorded (Figure 5).

For each operating system, two distinct regions can be identified. The left region exhibits a steep decline which levels off near zero seconds. This shows a significant performance improvement for every extra lower TLB slot made available to the system, up to a certain point. For example, simply moving from 4 to 5 lower slots decreases OSF/1 L2 miss-handling time by almost 50%. After 6 lower slots, the improvement slows because the TLB can hold most of the L2 PTEs required by OSF/1.⁷

In contrast, the Mach 3.0 system continues to show significant improvement up to 8 lower slots. The additional 3 slots needed to bring Mach 3.0's performance in line with OSF/1 are due to the migration of OS services from

 $^{^7\}mathrm{Two}$ L2 PTEs for kernel data structures and one each for a task's text, data, and stack segments.

ACM Transactions on Computer Systems, Vol. 12, No. 3, August 1994



Fig. 5. L2 PTE miss cost vs. number of lower slots. The total L2 miss time for the mab benchmark under different operating systems. As the TLB reserves more lower slots for L2 PTEs, the total time spent servicing L2 misses becomes negligible.

the kernel to the Unix server in user space. In Mach 3.0, whenever a task makes a system call to the Unix server, the task and the Unix server must share the TLB's lower slots. In other words, the Unix server's three L2 PTEs (text segment, data segment, stack segment) increase the lower slot requirement, for the system as a whole, to 8.

Mach3+AFSin's behavior is similar to Mach 3.0 because the additional AFS cache manager functionality is mapped by the Unix server's L2 PTEs. However, when the AFS cache manager is decomposed into a separate user-level server, the TLB must hold three additional L2 PTEs (11 total). Figure 5 shows how Mach3+AFSout continues to improve until all 11 L2 PTEs can reside simultaneously in the TLB.

Unfortunately, increasing the size of the lower partition at the expense of the upper partition has the side-effect of increasing the number of L1U, L1K, and L3 misses as shown in Figure 6. Coupling the decreasing L2 misses with the increasing L1U, L1K, and L3 misses yields an optimal partition point, shown in Figure 6.

This partition point, however, is only optimal for the particular operating system. Different operating systems with varying degrees of service migration have different optimal partition points. For example, the upper graph in Figure 7 shows an optimal partition point of 8 for Mach 3.0, 10 for Mach3 + AFSin, and 12 for Mach3 + AFSout, when running the Ousterhout benchmark.

Applications and their level of interaction with OS services also influence the optimal partition point. The lower graph in Figure 7 shows the results for various applications running under Mach 3.0. compress has an optimal partition point at 8 slots. However, video_play requires 14, and mpeg_play requires 18 lower slots to achieve optimal TLB performance. The need for additional lower slots is due to video_play and mpeg_play's increased interaction with services like the BSD server and the X server. It also underscores the importance of understanding both the decomposition of the system and how applications interact with the various OS services.



AFSout

AFSin

Mach3

26 ·

24

22

(sec)

18



Fotal Time 16 14 12 10 12 4 6 8 10 12 14 16 18 20 22 24 6 8 10 12 14 16 18 20 22 24 26 28 30 4 Number of Lower Slots Number of Lower Slots Fig. 7. Optimal partition points for various operating systems and benchmarks. As more lower

slots are allocated, fewer upper slots are available for the L1U, L1K, and L3 PTEs. This yields an optimal partition point which varies with the operating system and benchmark. The left graph shows the average of 3 runs of the ousterhout benchmark run under 3 different operating systems. The right graph shows the average of 3 runs for 3 different benchmarks run under Mach 3.0.

The TLB partition was implemented to allow operating systems to separate PTEs with low retrieval costs from PTEs with higher retrieval costs. All four of our operating systems use the lower slots for L2 PTEs and the upper slots for all other types of PTEs. However, it is unclear why the costly-to-retrieve L1K and L3 PTEs are not placed in the lower slots, but are mixed with the L1U PTEs. Further, if mixing costly-to-retrieve mappings with L1U PTEs is acceptable, it is unclear if PTE partitioning is required at all.

To determine the effects of other PTE placement choices, we modified the miss handlers of the baseline systems to implement other meaningful policies. The results are shown in Table VIII. Policy A is identical to that implemented by the baseline systems. The importance of some sort of partitioning is shown by Policy D, where all PTEs are mixed together, and which demonstrates very poor performance. At first glance, the baseline policy A appears to be the most desirable. However, note that with policies B and C, the lower partition was not permitted to grow beyond 8 slots to accommodate the additional PTE types allocated to this partition.

To see if the performance of policies B and C would improve with a larger lower partition, we varied the partition point from its fixed location at 8. Figure 8 shows the results for this experiment performed for PTE placement policies A, B, and C. Only the total curves are shown. Note that each policy has different optimal points, but at these optimal points the performance is roughly the same for each system. From this we can conclude that the most important PTE placement policy is the separation of L1U and L2 PTEs (to avoid the very poor performance of policy D). However, the differences between the other PTE placement policies A, B, and C are negligible, provided that the partition point is tuned to the optimal region.

Careful software management of the TLB can extend the degree to which separate services can coexist in a system before performance degrades to an unacceptable level. This is important because a key characteristic of microkernel system structuring is that logically independent services should reside in separate user-level tasks that communicate through message passing. To illustrate this more clearly, we constructed a workload that emulates the interaction between servers in a multiserver microkernel OS. In this workload, a collection of user-level tasks mimics the behavior of communicating OS servers by passing a token between each other. The number of servers and the number of pages that each server touches before passing the token along can be varied. Figure 9 shows the results of running this multiserver emulator on the Mach 3.0 kernel. With each additional server, the optimal partitioning point moves farther to the right. A system that leaves the partition point fixed at 8 will quickly encounter a performance bottleneck due to the addition of servers. However, if the TLB partition point is adjusted to account for the number of interacting servers in the system, a much greater number of servers can be accommodated. Nevertheless, note that as more servers are added, the optimal point still tends to shift upward, limiting the number of tightly coupled servers that can coexist in the system. This bottleneck is best dealt with through additional hardware support in the form of larger TLBs or miss vectors dedicated to level-2 PTE misses.

Policy	PTE Placement	Cost
A	Level 2 PTEs in lower partition. All other PTEs in upper partition.	1.91
В	Level 2 and 3 PTEs in lower partition. Level 1 user and kernel PTEs in upper partition.	3 92
С	All PTEs in lower partition, except for level 1 user PTEs which are placed in upper partition	2.46
D	No partitioning at all	11.92

Table VIII. Alternate PTE Placement Policies.

This table shows the affect of alternate PTE placement policies on TLB management cost The cost is the total time spent (in seconds) to handle TLB misses for the Mach 3.0 system running ousterhout. The partition point is fixed at 8 slots for the lower partition and 56 slots for the upper partition.



Fig. 8 TLB partitioning for different PTE placement policies. This is the same experiment as that of Table VIII, except with different PTE placement policies. Only the total TLB management costs are shown. The total cost for Policy D (no partition) is off the scale of the plot at 11.92 seconds

The baseline systems use a FIFO replacement policy for the lower partition and a random replacement policy for the upper partition when selecting a PTE to evict from the TLB after a miss. To explore the effects of the replacement policy in these two partitions, we modified the miss handlers to try other combinations of FIFO and random replacements in the upper and lower partitions. The results of one such experiment are shown in Figure 10. For these workloads, differences between the replacement policies are negligible over the full range of TLB partition points, indicating that the choice between these two replacement policies is not very important. Our trap-driven simulation method makes it difficult to simulate nonrecently used and other



Fig. 9. TLB partitioning under multiserver operating systems. This graph shows total TLB management costs for Mach 3.0 running a workload that emulates a multiserver system by passing a token among different numbers of user-level tasks.



Fig. 10. Replacement policies. This graph shows the performance of different replacement policies (random or FIFO) for the Mach 3.0 system implementing PTE placement policy A on ousterhout.

198 • Richard Uhlig et al.

Workload	Fixed Partitioning (sec)	Static Partitioning (sec)	Dynamic Partitioning (sec)
ousterhout	3 92 (8)	1.27 (18)	1 11
video_play	16.1 (8)	14 7 (18)	14 3
IOzone	1 30 (8)	0 43 (32)	0 43

Table IX Different TLB Partitioning Schemes.

This table compares the total TLB management costs when fixing the partition at 8, when setting it to the static optimal point (shown in parentheses), and when using dynamic partitioning. The PTE placement policy is B.

pseudo-LRU policies, so currently we cannot extend this statement to other policies.

5.2.2 Dynamic Partitioning. We have shown that the best place to set the TLB partition point varies depending on the operating system structure, workload, and PTE placement policy, but not the replacement policy. Given knowledge of these factors ahead of time, it is possible to determine the optimal partition point and then fix it statically for the duration of some processing run. However, although an operating system can control PTE placement policy and have knowledge of its own structure, it can do little to predict the nature of future workloads that it must service. Although system administrators might have knowledge of the sort of workloads that are typically run at a given installation, parameters that must be tuned manually are often left untouched or are set incorrectly.

To address these problems, we have designed and implemented a simple, adaptive algorithm that self-tunes the TLB partition dynamically to the optimal point. The algorithm is invoked after some fixed number of TLB misses at which time it decides to move the partition point either up, down, or not at all. It is based on a hill-climbing approach, where the objective function is computed from the two most recent settings of the partition point. At each invocation, the algorithm tests to see if the most recent partition change resulted in a significant increase or decrease in TLB miss-handling costs when compared against the previous setting. If so, the partition point is adjusted appropriately. This algorithm tends to home in on the optimal partition point and tracks this point as it changes with time.

We tested this algorithm on each of the benchmarks in our suite and compared the resultant miss-handling costs against runs that fix the partition at 8 and at the static optimal point for the given benchmark. The results are shown in Table IX. Note that dynamic partitioning yields results that at times are slightly better than the static optimal. To explain this effect, we performed another experiment that records the movement of the TLB partition point during the run of gcc, a component of the mab benchmark. The results (see Figure 11) show that the optimal partition point changes with



Fig. 11. Dynamic TLB partitioning during gcc. This graph shows the movement of the TLB partition with time while running gcc on a system that implements the dynamic, adaptive partitioning algorithm.

time as a benchmark moves among its working sets. Because the dynamic partitioning algorithm tracks the optimal point with time, it has the potential to give slightly better results than the static optimal which remains fixed at some "good average point" for the duration of a run.

The invocation period for the dynamic partitioning algorithm can be set so that its overhead is minimal. It should be noted, however, that there is an additional cost for maintaining the TLB miss counts that are required to compute the objective function. Although this cost is negligible for the already costly L2 and L3 PTE misses, it is more substantial for the highly tuned L1 PTE miss handler.⁸ Hardware support in the form of a register that counts L1 misses could help to reduce this cost.

5.3 Increasing TLB Size

In this section we examine the benefits of building TLBs with additional upper slots. The tradeoffs here can be more complex because the upper slots are used to hold three different types of mappings (L1U, L1K, and L3 PTEs), whereas the lower slots only hold L2 PTEs.

To better understand the requirements for upper slots, we used Tapeworm to simulate TLB configurations ranging from 32 to 512 upper slots. Each of these TLB configurations was fully associative and had 16 lower slots to minimize L2 misses.

Figure 12 shows TLB performance for all seven benchmarks under OSF/1. For smaller TLBs, the most significant components are L1K misses; L1U and

⁸Maintaining a memory-resident counter in the level-1 miss handler requires a load-incrementstore sequence. On the R2000, this can require anywhere from 4 to 10 cycles, depending on whether the memory reference hits the data cache. This is a 20% to 50% increase over the 20-cycle average currently required by the level-1 miss handler.

ACM Transactions on Computer Systems, Vol. 12, No. 3, August 1994.





L3 misses account for less than 35% of the total TLB miss-handling time. The prominence of L1K misses is due to the large number of mapped data structures in the OSF/1 kernel. However, as outlined in Section 5.1, modifying the hardware trap mechanism to allow the uTLB handler to service L1K misses reduces the L1K service time to an estimated 20 cycles. Therefore, we recomputed the total time using the lower-cost L1K miss service time (20 cycles) for the OSF/1, Mach 3.0, and Mach3+AFSout systems (Figure 13).

With the cost of L1K misses reduced, TLB miss-handling time is dominated by L1U misses. In each system, there is a noticeable improvement in TLB service time as TLB sizes increase from 32 to 128 slots. For example, moving from 64 to 128 slots decreases Mach 3.0 TLB handling time by over 50%.

After 128 slots, invalid and modify misses dominate (listed as "other" in the figures). Because the invalid and modify misses are constant with respect to TLB size, any further increases in TLB size will have a negligible effect on overall TLB performance. This suggests that a 128- or 256-entry TLB may be sufficient to support both monolithic operating systems like Ultrix and OSF/1 and microkernel operating systems like Mach 3.0. Of course, even larger TLBs may be needed to support large applications such as CAD programs. However, this article is limited to TLB support for operating systems running a modest workload. The reader is referred to Chen et al. [1992] for a detailed discussion of TLB support for large applications.

5.4 TLB Associativity

Large, fully associative TLBs (128 + entries) are difficult to build and can consume nearly twice the chip area of a direct mapped structure of the same capacity [Mulder et al. 1991]. To achieve high TLB performance, computer architects could implement larger TLBs with lesser degrees of associativity. The following section explores the effectiveness of TLBs with varying degrees of associativity.



Fig. 13. Modified TLB service time vs. number of upper TLB slots. The total cost of TLB miss servicing (for all seven benchmarks) assuming L1K misses can be handled by the uTLB handler in 20 cycles and L2 misses are handled in 40 cycles. The top graphs are for OSF/1 and Mach 3.0, and the bottom for Mach3+AFSout. Note that the scale varies for each graph.

Many current-generation processors implement fully associative TLBs with sizes ranging from 32 to more than 100 entries (Table X). However, technology limitations may force designers to begin building larger TLBs which are not fully associative. To explore the performance impact of limiting TLB associativity, we used Tapeworm to simulate TLBs with varying degrees of associativity.

The top two graphs in Figure 14 show the total TLB miss-handling time for the mpeg_play benchmark under Mach3+AFSout and the video_play benchmark under Mach 3.0. Throughout the range of TLB sizes, increasing associativity reduces the total TLB handling time. These figures illustrate the general "rule of thumb" that doubling the size of a caching structure will yield about the same performance as doubling the degree of associativity [Patterson and Hennessy 1990].

202 • Richard Uhlig et al.

Processor	Associativity	Number of Instruction Slots	Number of Data Slots
DEC Alpha 21064	full	8+4	32
IBM RS/6000	2-way	32	128
TI Viking	full	64 unified	
MIPS R2000	full	64 unified	
MIPS R4000	full	48 unified	
HP 9000 Series 700	full	96+4	96+4
Intel 486	4-way	32 unified	

Table X. Number of TLB Slots for Current Processors

Note that page sizes vary from 4K to 16 Meg and are variable in many processors. The MIPS R4000 actually has 48 double slots. Two PTEs can reside in one double slot if their virtual mappings are to consecutive pages in the virtual address space [Digital 1992]



Fig. 14. Total TLB service time for TLBs of different sizes and associativities ACM Transactions on Computer Systems, Vol 12, No 3, August 1994

Some benchmarks, however, can perform badly for TLBs with a small degree of set associativity. For example, the bottom graph in Figure 14 shows the total TLB miss-handling time for the compress benchmark under OSF/1. For a 2-way set-associative TLB, compress displays pathological behavior. Even a 512-entry, 2-way set-associative TLB is outperformed by a much smaller 32-entry, 4-way set-associative TLB.

These three graphs show that reducing associativity to enable the construction of larger TLBs is an effective technique for reducing TLB misses.

6. SUMMARY

This article seeks to demonstrate to architects and operating system designers the importance of understanding the interactions between hardware and software features, and the importance of tools for the measurement and simulation of these interactions. In the case study presented here, numerous performance problems resulted when a microkernel operating system was ported to a new platform. A significant component of the problem was attributable to an increased utilization of the TLB. Software management magnified the TLB's impact on overall performance because of the large variations in TLB miss service times that can exist.

The construction of the TLB can significantly influence TLB performance. TLB behavior depends on the kernel's use of virtual memory to map its own data structures, including the page tables themselves. TLB behavior is dependent also on the division of service functionality between the kernel and separate user tasks. Currently popular microkernel approaches rely on multiple server tasks, but here fell prey to performance difficulties with even a modest degree of service decomposition.

In our study, we have presented measurements of actual systems on a current machine, together with simulations of architectural problems, and have related the results to the differences between operating systems. We have outlined four architectural solutions to the problems experienced by microkernel-based systems: changes in the vectoring of TLB misses, flexible partitioning of the TLB, providing larger TLBs, and limiting the degree of associativity to enable construction of larger TLBs. The first two can be implemented at little cost, as is done in the R4000. These solutions are most applicable to the MIPS RISC architectural family on which the studies were performed. The more general lessons regarding the trends in operating system design, their potential for interaction with architectural features on machines to which they are ported, and the utility of tools for assessing these impacts are, we feel, more broadly applicable.

REFERENCES

- ACCETTA, M., BARON, R., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. 1986. Mach: A new kernel foundation for UNIX development. In the *Summer 1986 USENIX Conference*. USENIX Assoc., Berkeley, Calif.
- AGARWAL, A., HENNESSY, J., AND HOROWITZ, M. 1988. Cache performance of operating system and multiprogramming workloads. ACM Trans. Comput. Syst. 6, 4, 393-431.

- ALEXANDER, C. A., KESHLEAR, W. M., AND BRIGGS, F. 1985 Translation buffer performance in a UNIX environment. Comput. Arch. News 13, 5, 2–14.
- AMERICAN MICRO DEVICES. 1991 Am29050 Microprocessor User's Manual, American Micro Devices, Inc., Sunnyvale, Calif.
- ANDERSON, T. E. LEVY, H. M., BERSHAD, B. N, AND LAZOWSKA, E. D. 1991. The interaction of architecture and operating system design. In the 4th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, New York.
- CHEN, B. 1993 Software methods for system address tracing. In Proceedings of the 4th Workshop on Workstation Operating Systems. IEEE Computer Society Press, Los Alamitos, Calif, 178-185.
- CHEN, J B., BORG, A., AND JOUPPI, N. P. 1992. A simulation based study of TLB performance. In *The 19th Annual International Symposium on Computer Architecture* IEEE, New York
- CHERITON, D. R. 1984. The V Kernel: A software base for distributed systems IEEE Softw 1, 2, 19–42
- CLAPP, R. M, DUCHESNEAU, I J., VOLZ, R. A, MUDGE, T. N, AND SCHULTZE, T. 1986 Toward real-time performance benchmarks for Ada *Commun. ACM* 29, 8 (Aug), 760–778.
- CLARK, D. W., AND EMER, J S 1985. Performance of the VAX-11/780 translation buffer: Simulation and measurement. ACM Trans Comput. Syst. 3, 1, 31-62
- DEAN, R. W. AND ARMAND, F. 1991. Data movement in kernelized systems. In *Micro-Kernels* and Other Kernel Architectures. USENIX Assoc, Berkeley, Calif.
- DEMONEY, M., MOORE, J., AND MASHEY, J. 1992. Operating system supports on a RISC In COMPCON IEEE, New York
- DIGITAL. 1992. Alpha Architecture Handbook. Digital Equipment Corp., Bedford, Mass.
- HEWLETT-PACKARD 1990 PA-RISC 11 Architecture and Instruction Set Reference Manual. Hewlett-Packard, Inc.
- HUCK, J AND HAYES, J. 1993. Architectural support for translation table management in large address space machines. In *The 20th Annual International Symposium on Computer Architecture*. IEEE, New York.
- KANE, G. AND HEINRICH, J 1992. MIPS RISC Architecture Prentice-Hall, Englewood Cliffs, N.J.
- LARUS, J R 1990. Abstract execution: A technique for efficiently tracing programs Univ of Wisconsin-Madison.
- MCKUSIC, M. K, JOY, W. N., LEFFLER, S. J., AND FABRAY, R. S. 1984 A fast file system for UNIX. ACM Trans. Comput. Syst. 2, 3, 181–197.
- MILENKOVIC, M. 1990. Microprocessor memory management units. IEEE Micro. 10, 2, 75–85 MOTOROLA. 1993 PowerPC 601 RISC Microprocessor Users' Manual Motorola, Inc., Phoenix, Ariz.
- MOTOROLA. 1990. MC88200 Cache/Memory Management Unit User's Manual. 2nd ed Prentice-Hall, Englewood Chffs, N J
- MULDER, J. M., QUACH, N. T., AND FLYNN, M. J. 1991. An area model for on-chip memories and its application. *IEEE J. Solid-State Circ*, 1, 2, 98–106.
- NAGLE, D, UHLIG, R., AND MUDGE, T. 1992 Monster: A tool for analyzing the interaction between operating systems and computer architectures. Tech Rep. CSE-TR-147-92. The Univ. of Michigan, Ann Arbor, Mich
- OUSTERHOUT, J. 1989. Why aren't operating systems getting faster as fast as hardware? WRL Tech. Note, (TN-11)
- PATEL, K., SMITH, B. C., AND ROWE, L. A. 1992. Performance of a software MPEG video decoder Univ of California, Berkeley.
- PATTERSON, D. AND HENNESSY, J. 1990. Computer Architecture. A Quantative Approach. Morgan Kaufmann, San Mateo, Calif
- RASHID, R, TEVANIAN, A, YOUNG, M., GOLUB, D., BARON, R., BLACK, D., BOLOSKY, W., AND CHEW, J. 1988. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Trans. Comput.* 37, 8, 896–908
- SATYANARAYANAN, M. 1990. Scalable, secure, and highly available distributed file access. *IEEE Comput. 23*, 5, 9–21.

- SMITH, J. E., DERMER, G. E., AND GOLDSMITH, M. A. 1988. Computer system employing virtual memory. Assignee: Astronautics Corporation of America Patent No. 4,774,659.
- SMITH, M. D. 1991. Tracing with Pixie Tech. Rep. CSL-TR-91-497, Computer Systems Laboratory, Stanford Univ., Palo Alto, Calif.
- TALLURI, M., KONG S., HILL, M. D., AND PATTERSON, D. A. 1992. Tradeoffs in supporting two page sizes. In *The 19th Annual International Symposium on Computer Architecture*. IEEE, New York.
- WELCH, B. 1991. The file system belongs in the kernel. In USENIX Mach Symposium Proceedings. USENIX Assoc., Berkeley, Calif.
- WILKES, J. AND SEARS, B. 1992. A comparison of protection lookaside buffers and the PA-RISC protection architecture. HP Laboratories.

Received October 1993; revised May 1994; accepted July 1994