Software Management Techniques for Translation Lookaside Buffers

by

Kavita Bala

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1995

Author .... [illegible] ....
Department of Electrical Engineering and Computer Science
January 19, 1995

Certified by .... [illegible] ....
William E. Weihl
Associate Professor
Thesis Supervisor

Certified by .... [illegible] ....
M. Frans Kaashoek
Assistant Professor
Thesis Supervisor

Accepted by .... [illegible] ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Software Management Techniques for Translation Lookaside Buffers

by

Kavita Bala

Submitted to the Department of Electrical Engineering and Computer Science
on January 19, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

A number of interacting trends in operating system structure, processor architecture, and memory systems are increasing both the rate of translation lookaside buffer (TLB) misses and the cost of servicing a TLB miss. This thesis presents two novel software schemes, implemented under Mach 3.0, to decrease both the number and the cost of *kernel* TLB misses (*i.e.*, misses on kernel data structures, including user page tables). The first scheme is a new use of prefetching for TLB entries on the IPC path, and the second scheme is a new use of software caching of TLB entries for hierarchical page table organizations.

For a range of applications, prefetching decreases the number of kernel TLB misses by 40% to 60%, and caching decreases TLB penalties by providing a fast path for over 90% of the misses. Our caching scheme also decreases the number of cascaded TLB traps due to the page table hierarchy, reducing the number of kernel TLB misses by 20% to 40%. For these applications, TLB penalties range from 1% to 5% of application runtime; our schemes are very effective in reducing kernel TLB penalties, improving application performance by up to 3.5%. We also demonstrated the impact of increasing the number of client/server processes, and increasing the data accessed by a process, on TLB miss handling times. For these synthetic benchmarks, our schemes, especially the integrated scheme with both prefetching and caching, perform very well improving runtimes for fine-grained benchmarks by up to 10%. Processor speeds continue to increase relative to memory speeds; a simple analytical model indicates that our schemes should be even more effective in improving application performance on future architectures.

Thesis Supervisor: William E. Weihl
Title: Associate Professor of Computer Science and Engineering

Thesis Supervisor: M. Frans Kaashoek
Title: Assistant Professor of Computer Science and Engineering

Acknowledgments

I would like to thank my advisors, Bill and Frans, for their ideas, and suggestions. Under their guidance I have learned how to approach research problems. I would also like to thank them for their patience during the rough patches of this work. I have learned a lot over these past two years, and I am sure everything I have learned will help me in the years to come.

I would like to specially thank Carl Waldspurger for his invaluable advice and time. Carl has been a great support and his technical contributions have helped the work immensely. It has been a pleasure interacting with him as a colleague and friend.

Thanks to both Wilson Hsieh and Carl for their contribution to early ideas in this work.

I would like to thank Krishna, my brother, for the support he has given me over the years, and the many discussions we have had on how the world of Computer Science works. Thanks also to friends, Donald Yeung, Patrick Sobalvarro, and Sivan Toledo, for their support. I would also like to thank Simrat, Joemama, and my parents for their encouragement.

Thanks to Anthony Joseph for his help with X, and Rich Uhlig for his comments and help with Mach. Thanks to everybody who helped me with drafts: Andrew Myers, Debby Wallach, Kevin Lew, Paige Parsons, Ulana Legedza. I would also like to thank everybody who gave me valuable feedback on my presentation: especially John Kubiakowitz, Ken Mackenzie, Kirk Johnson, and Steve Keckler. Thanks to Professor Anant Agarwal and Kirk for lending me their machine.

Thanks also to Paul Leach and the anonymous referees of the OSDI committee for their comments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A number of interacting trends in operating system structure, processor architecture, and memory systems are increasing both the rate of translation lookaside buffer (TLB) misses and the cost of servicing a TLB miss. This thesis presents two novel software schemes to decrease both the number and the cost TLB misses. The first scheme is a new use of prefetching of TLB entries between communicating processes. The second scheme is a new use of software caching of TLB entries for hierarchical page table organizations.

We have implemented these techniques under the Mach 3.0 microkernel on an R3000-based machine, and have evaluated their performance on a range of video and file-system applications. We have also studied synthetic benchmarks that demonstrate the impact of increasing the number of client/server processes, and increasing the data accessed by a process, on TLB miss handling times. In the applications studied, our schemes proved to be very effective in decreasing TLB penalties. Processor speeds continue to increase relative to memory speeds; we developed a simple analytical model to understand the impact of architectural trends on TLB miss handling.

First, we motivate the importance of software TLB mangement schemes. The TLB is an on-chip hardware cache of virtual address to physical address translations. TLBs have traditionally been fairly small and organized as highly associative caches, *e.g.*, the MIPS R2000/3000 architecture has a 64-entry fully associative TLB. When an address translation is found cached in the TLB, the TLB accelerates memory references. However, when an address translation is not found in the TLB, a *TLB miss* takes place and the translation has to be looked up in the operating system page tables. CISC architectures in the past handled TLB misses in hardware, *e.g.*, the i386 has a hardware-managed TLB. However, the trend in processor architectures over the past few years has been towards RISC-based organizations. This trend has motivated some architecture designers to move handling TLB misses into software. Several modern RISC architectures, *e.g.*, the MIPS R2000/3000/4000, Digital's Alpha, and the HP PA-RISC, adopt this approach. This move to software handling of TLB misses simplifies the hardware design considerably. The ability to handle TLB misses in software also permits greater flexibility to the operating system in the organization of page tables. For example, different operating systems can implement different page table organizations for the same architecture. However, handling TLB misses in software increases the time to handle a miss.

Another trend in memory system development further increases TLB miss times. As CPU speeds continue to increase relative to memory speeds [21], the time to access entries in the operating system page tables increases. This further increases the time to service a TLB miss. Also, studies indicate that operating systems benefit less than applications from improvements in caching techniques [8]. Therefore, the time spent in servicing TLB misses relative to the application runtime is likely to increase on future architectures.

While these trends in processor architecture and memory system latency increase the time to service a TLB miss, trends in operating system organization are also increasing the number of TLB misses. Modern microkernel-based operating systems such as Mach 3.0 [1] and Amoeba [19] provide minimal primitives within the kernel for VM management, inter-process communication and scheduling. All other OS functionality is provided by user-level server processes which reside in different protection domains. This is unlike traditional monolithic operating systems, such as Ultrix [11], which provide all functionality within the kernel. The benefits of microkernel-based operating systems are increased flexibility, portability and modularity. However, the existence of user-level server processes requires frequent communication between client processes, the kernel, and server processes. Since more communicating processes have to be simultaneously mapped by the TLB, the number of TLB misses in these systems is greater than that in monolithic operating systems [28]. Furthermore, many microkernel-based systems use virtual memory rather than physical memory for most OS data structures. This increases the number of pages in the active working set that require TLB entries.

As researchers continue to improve inter-process communication (IPC) performance [5, 18], TLB penalties will become an increasing fraction of the IPC cost [4, 14]. In addition, recent commercial standards, such as OLE [12] and OpenDoc [25], place an increasing emphasis on inter-application communication. This emphasis on increased communication between processes, could result in an increase in the number of TLB misses.

Thus, current trends are increasing both the rate of TLB misses and the relative cost of servicing a miss. The net effect of all these factors is that the impact of TLB misses on overall system performance is increasing [2, 8]. While TLB penalties have been recognized as a problem, proposed solutions typically require expensive hardware [28, 20].

In this thesis we present two schemes to address this problem that rely only on software mechanisms; no additional hardware is required. These techniques reduce both the number and the cost of TLB misses. The first technique involves *prefetching* TLB entries during IPC. This well-known technique has not been applied before in the context of TLB management. After an IPC, many page table accesses that cause TLB misses are highly predictable. Prefetching these entries in the kernel during the IPC can eliminate many misses. We implemented and evaluated our techniques under the Mach 3.0 microkernel on an R3000-based machine. For a range of applications, prefetching decreases the number of kernel TLB misses by 40% to 60%.

The second technique introduces a software cache for the TLB, called the *software TLB* (STLB). On several architectures, handling some types of TLB traps is quite expensive (on the order of several hundred cycles). By introducing a large software cache for TLB entries and checking it early in the TLB miss trap handler, this approach reduces the time for servicing expensive misses. In addition, in systems with hierarchical page table organizations, page tables themselves are mapped. A hit in the STLB avoids further references to the page tables. Preventing cascaded misses reduces the total number of TLB misses by about 20% to 40% for the applications we studied. For a range of applications, caching decreases TLB penalties by providing a fast path for over 90% of the misses.

For these applications, TLB penalties range from 1% to 5% of application runtime; our schemes are very effective in reducing kernel TLB penalties, improving application performance by up to 3.5%. We implemented a synthetic benchmark to study the impact of increasing the number of client/server processes, and increasing the data accessed by a process, on TLB miss handling times. Our schemes perform very well under these scenarios, improving runtimes for fine-grained benchmarks by up to 10%. Since processor speeds continue to increase relative to memory speeds, we expect our schemes to be even more effective in improving application performance on future architectures.

Our experiments assume a microkernel-based operating system. However, these techniques can be applied to other operating system organizations, such as large single address space systems [7] and systems with software segmentation [29]. Furthermore, with the current emphasis on application-controlled resource management [3, 13], our prefetching techniques could become even more effective, since the prefetching strategy can be tailored for individual applications. Prefetching can also be integrated with other VM functions such as prefetching cache entries.

An outline of the thesis follows: Chapter 2 presents background material on page table organizations and VM management. Chapter 3 describes the two schemes, the scheme integrating both techniques, and implementation details. In Chapter 4, the experimental methodology followed to evaluate the schemes is explained, and results are presented. Chapter 4 also presents a synthetic benchmark that demonstrates the impact of multiple clients and servers on application runtime. Chapter 5 presents a model for TLB penalties on faster architectures. In Chapter 6 we discuss related work, and finally, in Chapter 7 we present conclusions.

Chapter 2

Background

In this chapter we present some background material on virtual memory organizations and the impact that the memory organization has on TLB miss handling penalties.

2.1 Virtual Memory Organizations

Virtual memory (VM) supports the abstraction of different processes, giving each process its own private address space. The VM system translates virtual memory addresses to physical memory addresses. These translations and associated protection information are stored in page tables; one page table for each address space. Translation lookaside buffers (TLB) speedup this translation by caching page table entries.

On architectures supporting software-managed TLBs, the VM management system has the flexibility to choose its page table organization. Some common page table organizations are inverted page tables and forward-mapped page tables. Inverted page tables are typically used on large address space architectures, *e.g.,* HP-UX implements an inverted page table for the PA-RISC [15]. Forward-mapped page tables are typically used on 32-bit architectures, *e.g.,* Mach 3.0 implements a 3-level page table hierarchy for the MIPS R2000/3000 [27]. The organization of the page table affects the time to handle TLB misses.

Figure 2-1 depicts the page table hierarchy implemented by Mach 3.0 for a MIPS

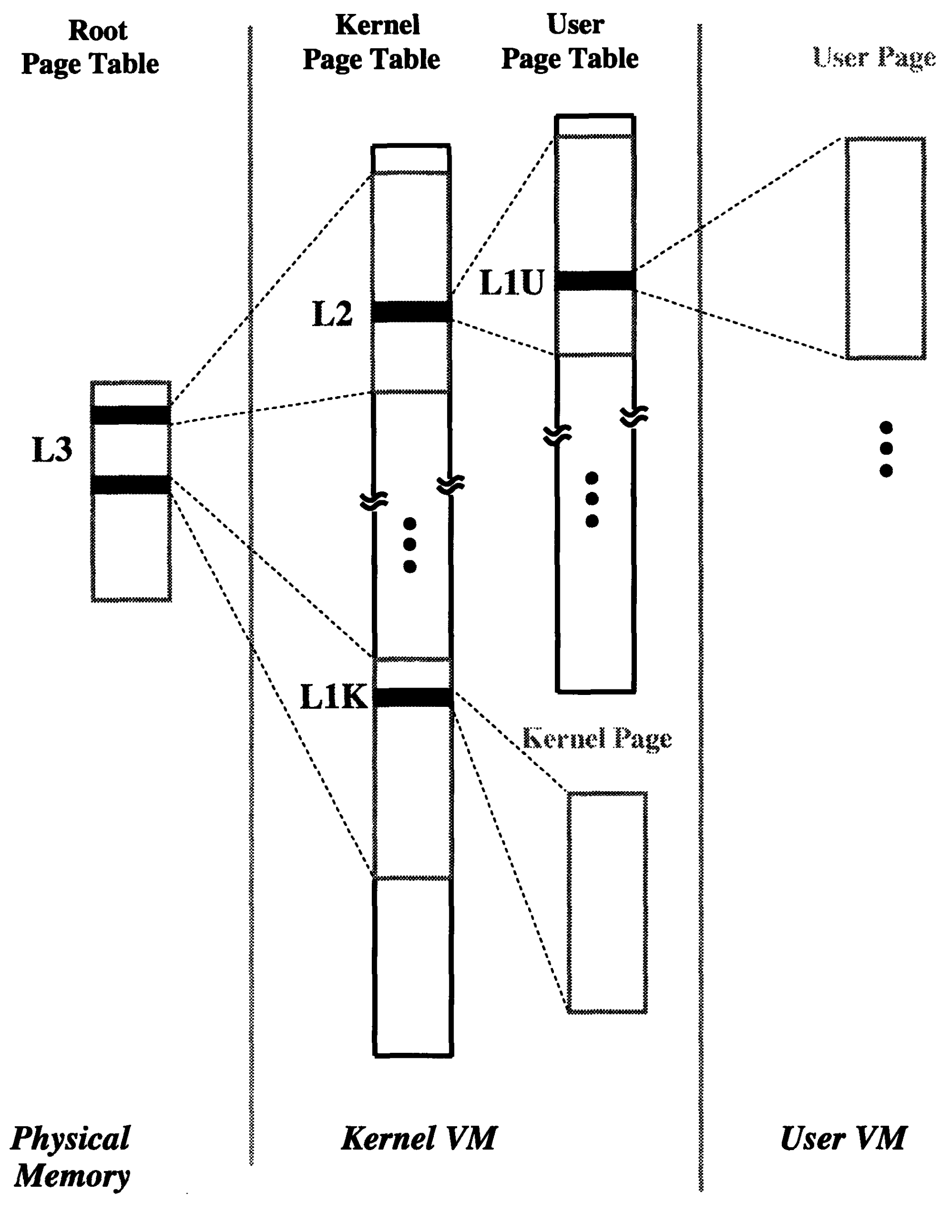

Figure 2-1: **Mach 3.0 Page Table Hierarchy**. Mach 3.0 implements a 3-level page table hierarchy on a MIPS R3000-based machine.

R3000-based machine. The light gray rectangles correspond to pages. There are four types of page table entries (PTEs): L1U, L2, L1K, and L3. As can be seen from the figure, L1U (*Level 1 User*) PTEs map the pages of user address spaces, and are stored in user page tables. The user page tables themselves reside in the kernel's virtual address space, and therefore need to be mapped. L2 (*Level 2*) PTEs map user page tables, and are stored in the kernel page table. L1K (*Level 1 Kernel*) PTEs map kernel data structures, and are also stored in the kernel page table, like L2 PTEs. Since the kernel page table resides in the kernel's virtual memory itself, it needs to be mapped. L3 (*Level 3*) PTEs map the kernel page table. A page pinned down in physical memory at the root of the hierarchy holds the L3 PTEs [28].

2.2 TLB Miss Handling

The page table organization affects the types of TLB misses, and the cost of TLB misses. For the three-level page table hierarchy depicted in Figure 2-1, each type of PTE corresponds to a type of TLB miss. In addition, TLB-invalid and TLB-modify misses manipulate protection bits. Thus, there are six types of TLB misses on the MIPS R3000: L1U, L1K, L2, L3 misses, TLB-invalid, and TLB-modify misses. TLB-modify misses modify protection bits associated with TLB entries. Our proposed schemes do not deal with TLB-modify misses and they are not mentioned further. A TLB-invalid miss takes place on user addresses that are marked invalid by the virtual memory system. These misses require the intervention of the VM system and are handled identically both in unmodified Mach and in the modified versions of Mach implementing our proposed schemes. Typically, these misses require bringing in a page from the disk.

Table 2.1 presents the average cost of each miss type, in CPU cycles, for a 40MHz R3000-based DECStation 5000/240 running Mach 3.0. The cycle counts reported in the table were measured using the IOASIC counter on the 25 MHz system bus. This counter is a free-running 32 bit counter that is incremented once per TURBOchannel clock cycle (40 ns for the DECstation 5000/240). The time to service TLB traps

Type of miss	**Penalty (cycles)**
L1U	10 or 30-40
L1K	512
L2	555
L3	407
TLB-invalid	338

Table 2.1: **Average TLB Miss Penalties.** Average penalties, measured in CPU cycles, for TLB misses under Mach 3.0 on a 40 MHz R3000-based DECstation. These averages were measured using the IOASIC counter. The cycles counts for the misses exhibited fairly high variability.

exhibited fairly high variability, on the order of a couple hundred cycles. This variability is probably due to different number of cache misses on different invocations of the TLB miss handler.

Since L1U misses are the most frequent, the architecture provides a special trap handler, making them fast. The special L1U trap handler executes 9 instructions and one memory load. If the memory load hits in the cache the handler executes in 10 cycles; otherwise, the L1U miss handler executes in about 30-40 cycles. All other misses are slow, since they are serviced by a generic trap handler that handles all traps except for L1U TLB misses.

Our proposed prefetching and caching schemes decrease the number of L1K, L2, and L3 misses, from now on referred to as the *kernel TLB misses.* Kernel TLB misses typically account for greater than 50% of TLB penalties [28]. Since L1U miss handling is extremely fast, and TLB-invalid misses require the intervention of the VM system, our schemes do not service these types of TLB misses.

Chapter 3

Software TLB Mangement

This chapter presents the two software TLB management schemes proposed and studied in this thesis. The first section presents the issues involved in prefetching TLB entries followed by a brief discussion of the prefetching implementation. The second section presents software caching for TLB entries, and discusses its implementation. In the third section of this chapter we discuss implementation issues involved in integrating the two schemes, and finally, we summarize the schemes in the last section.

3.1 Prefetching TLB Entries

One software approach to decrease TLB overheads is to prefetch page table entries into the hardware TLB. "Prefetching" in this context means that the TLB entries are looked up in an auxiliary data structure and entered into the hardware TLB before they are needed. Thus, if a TLB entry that is prefetched is subsequently used, this scheme avoids taking a software TLB trap at the time of use. Successful prefetching reduces the number of TLB misses and thus eliminates the overhead of invoking TLB miss handlers. It should be noted that, unlike in some architectures, "prefetching" in this context does not mean overlapping memory latency with useful computation.

When prefetching TLB entries, three issues must be resolved:

- when to prefetch,

- what entries to prefetch, and
- how many TLB entries to prefetch.

To resolve these issues we collected trace data of the TLB misses in the system. This trace data helped us make informed decisions about prefetching.

3.1.1 When to prefetch

Prefetching TLB entries would appear to be useful on context switches between different protection domains. The intuitive reason for this is that when a context is switched to a different protection domain, the TLB entries of the domain being switched to may not be resident in the TLB. Also, TLB manipulation instructions on current architectures can only be done in "privileged mode", and therefore, manipulating the TLB from user space is expensive. Since control is transferred to the kernel on a context switch, prefetching TLB entries on a context switch is much cheaper.

Contexts are switched implicitly on inter process communication (IPC) between concurrently executing processes, and explicitly by the scheduler. When several communicating processes are executing they compete for slots in the TLB. Since there are a relatively small number of available TLB slots, the number of TLB misses increases. In this situation, prefetching on the communication path can be useful.

When a task is being activated by the scheduler, the TLB entries of the task being switched to may not be resident in the TLB. Therefore, scheduler calls to `thread wakeup` are another place where prefetching would appear to be useful.

3.1.2 What to prefetch

A process is typically structured such that its stack segment and code segment are allocated at opposite ends of the process's virtual address space, while the data segment is allocated somewhere in the middle of the address space. L1U entries are the entries which map the pages of a process's stack or code segment. L2 entries map these L1U entries. Since the code, stack and data segments are scattered in the process's virtual address space, *at least* three L2 entries will be required to map a process. Therefore, we hypothesized that prefetching the L2 entries mapping the process's stack, code, and data segments on IPCs could be beneficial.

3.1.3 How many entries to prefetch

There is a trade-off between the benefits of prefetching and the overheads added by prefetching. On the one hand, prefetching more entries could result in increased benefits by eliminating more TLB misses. On the other hand, prefetching entries that are not subsequently used could evict useful entries from the TLB, and thus increase the total number of TLB misses. Also, more overhead is added to the IPC path as more entries are prefetched. Therefore, aggressive prefetching could result in performance degradation.

3.1.4 Trace data

We collected trace data to help us make informed decisions about the different ways in which prefetching could be implemented. In our experiments with the microkernel-based operating system Mach 3.0, for the applications we studied, we found that context switches through IPC were far more frequent than explicit context switches through the scheduler. Therefore, we decided to prefetch TLB entries on the IPC path. However, prefetching can also be implemented on the scheduler context switch. Depending on the granularity of the scheduling quantum, the benefits of prefetching TLB entries on explicit context switches might or might not be worthwhile.

We hypothesized that prefetching the TLB entries that map a process's data segment, PC and SP would be useful. To test this hypothesis we collected traces of TLB misses, and the PC and SP of the process taking the misses, and compared the miss addresses with the addresses we planned to prefetch. From the traces we learned that in addition to the above TLB entries, for communicating processes, prefetching TLB entries that map message buffers and those that map IPC data structures is also useful. Table 3.1 presents a list of the frequent TLB miss addresses and their contribution to overall kernel TLB misses.

Address Type	Percentage TLB Misses
Kernel IPC data, L1K miss	14%
Kernel IPC data, L3 miss	11%
User PC, L2 miss	10%
User SP, L2 miss	15%
Message buffers, L2 miss	10%
Message buffers, L3 miss	10%

Table 3.1: **Trace data: Percentage of TLB misses.** This table presents the percentage of TLB misses for the kernel's IPC data, and a user-level process's PC, SP and message buffers.

Prefetching L1K entries mapping kernel data structures could eliminate the associated L3 misses, thus eliminating about 25% of kernel TLB misses. Prefetching L2 entries mapping the code and stack segments of processes could eliminate another 25%, while prefetching L2 entries mapping a process's message buffers could eliminate another 20%.

Prefetching the data segment of a process is not easy under Mach. The reason is that Mach 3.0 supports sparse memory allocation, i.e., a process can map physical pages to any location in its virtual address space. Thus allocation of memory in non-contiguous locations of the virtual address space of a process is permitted [6]. While this is a useful feature of Mach's VM management system, it is not easy to dynamically determine the location of the "data segment" of a process. UNIX maintains a variable called the `break` which points to the data segment of a process. A Mach process does not have such a UNIX style `break`. Therefore, L2 entries mapping the data segment of a process could not be easily prefetched.

The inability to prefetch the TLB entry mapping a process's data segment makes prefetching L3 entries beneficial. To understand this we present some details of Mach's implementation of the page table hierarchy on the MIPS R2000/3000. An L1U entry maps one page of a user process's address space. Each page on the MIPS R2000/3000 is of size 4KB; therefore, each L1U entry maps 4KB of data. Since each page table entry is 4 bytes in size, a page of L1U entries has 1024 L1U entries. Thus, an L2 entry mapping a page of L1U entries, maps 1024 L1U entries, i.e., $1024 \times 4KB = 4\ MB$ of user data. Similarly an L3 entry maps 1 page of L2 entries, i.e., $1024 \times 4\ MB = 4\ GB$ of data (which is bigger than the address space of a process; a process has a 2 GB virtual address space). Thus, only one L3 entry is required to map the entire virtual address space of a process. Therefore, even though the L2 entry for the data of a process is not prefetched, prefetching the L3 entry associated with message buffers at least eliminates cascaded misses when handling subsequent L2 misses on the process's data.

Therefore, we decided to prefetch the following TLB entries:

- L1K entries mapping IPC data structures,
- L2 entries mapping the process's stack and code segments,
- L2 entries mapping message buffers, and
- L3 entries associated with the L2 entries mapping message buffers.

Thus, the trace data indicated that up to 70% of kernel TLB misses could be eliminated by prefetching. In our experiments, we found that prefetching eliminated 25% to 65% of kernel TLB misses for the applications we studied. We also learned from the trace data that the remaining 30% of TLB misses were scattered over a range of addresses; a TLB miss on any particular address was relatively infrequent. Therefore, more aggressive prefetching was not worthwhile.

3.1.5 Implementation

Our implementation maintains TLB entries to be prefetched in a separate auxiliary data structure called the *Prefetch TLB* (PTLB). The PTLB is a table maintained in unmapped, cached physical memory. Since it is unmapped, a cascade of misses is avoided when looking up an entry in the PTLB. The PTLB is maintained as a direct-mapped table storing L1K, L2, and L3 TLB entries. PTLB entries are kept consistent with the kernel page tables efficiently, by invalidating them when the kernel invalidates page table entries.

TLB entries are prefetched on the IPC path between different protection domains. On the first IPC made by a process, the L2 entries corresponding to the process's program counter (PC), stack pointer (SP), and message buffers are probed in the hardware TLB. This probe is done using the MIPS instruction `tlbp`. If the requested address is found in the hardware TLB, its associated PTE is returned and stored in the PTLB. If the requested address is not found in the hardware TLB, the probe is repeated on the next IPC made by this process. However, this is relatively infrequent because the TLB entries of a process making an IPC are typically in the hardware TLB. Subsequently when the process is receiving a message, its PC, SP, and message buffer addresses are looked up in the PTLB. If present, their associated PTEs are placed in the hardware TLB.

Similarly for kernel IPC data structures, the first access to the data invokes the generic trap handler. After returning from the generic trap handler, the address is stored in the PTLB. On subsequent accesses to the data, the PTE is prefetched and then the data is accessed. It should be noted that prefetching does not add overhead to the TLB trap handler, but only to the IPC path; however, this overhead is not very large (adding only about 25-60 cycles to a path of length 3500-4000 cycles).

3.2 Software Cache of TLB Entries

In this section we present our second scheme to decrease the cost and number of TLB misses: a software cache for TLB entries. Architectures with software-managed TLBs incur large penalties in TLB miss handling due to

1. the use of generic trap handlers, and
2. cascaded TLB misses resulting from hierarchical page table organizations.

To address this problem, our second scheme proposes using a large second-level software cache of TLB entries, which we refer to as the software TLB (or STLB). When an L1K, L2, or L3 entry is entered into the hardware TLB, it is also stored in the STLB. On subsequent misses in the hardware TLB, the STLB code branches off

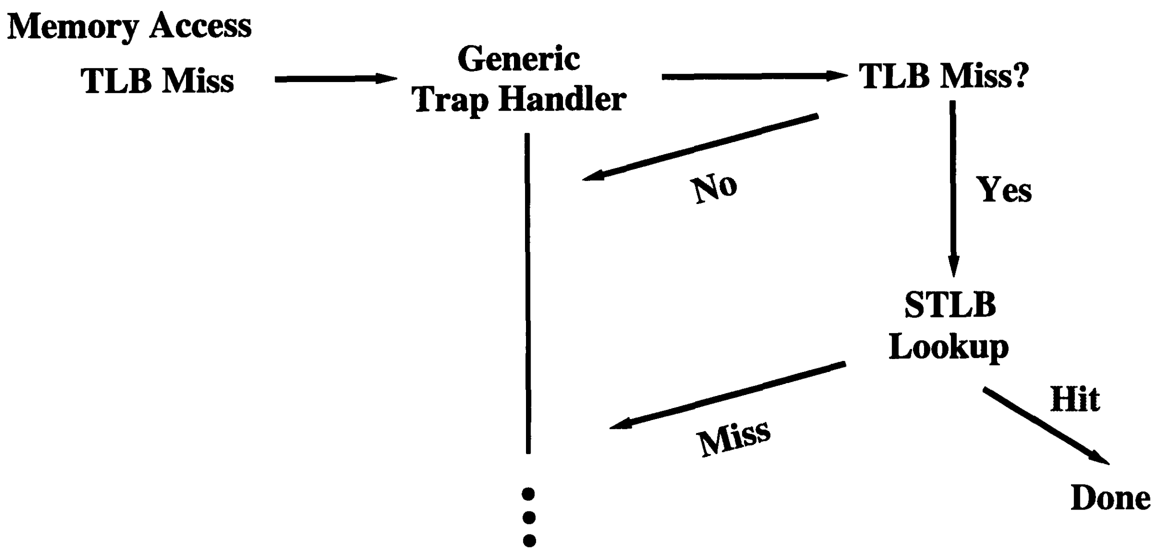

Figure 3-1: **STLB lookup path.** A pictorial representation of the STLB lookup path.

the generic trap handler path at the beginning, and does a quick lookup in the STLB. On an STLB hit, the STLB entry is inserted into the hardware TLB. On an STLB miss, the code branches back to the generic trap handler path. When the generic trap handler updates the hardware TLB, it also updates the STLB with the PTE. Figure 3-1 depicts the code path for an STLB lookup.

Thus, the first benefit of the STLB is that it provides a fast trap path for TLB misses, and on an STLB hit this avoids the overhead (such as saving and restoring registers) of the generic trap handler. The second benefit of the STLB is that it eliminates *cascaded TLB misses.* Cascaded TLB misses occur because the page tables are hierarchically organized with the lower levels of the hierarchy in mapped memory. The TLB trap handler code tries to resolve a TLB miss by looking in the next higher level in the page table hierarchy. However, this lookup can itself cause a TLB miss (up to a depth of three in Mach's page table organization), resulting in a cascade of TLB misses. The STLB provides a flat space of TLB entries in unmapped physical memory, and thus prevents such cascaded TLB misses.

	Penalty (cycles)		
	STLB		
Miss Type	Hit	Miss	Mach
L1K	105	582	512
L2, Path 1	114	625	555
L2, Path 2	160	625	555
L3	105	408	338

Table 3.2: **Average TLB penalties for a direct-mapped STLB.** An STLB miss adds about 70 cycles to the TLB trap path. The first L2 path through the STLB is more frequent than the second L2 path, and has been optimized to take less time.

As with any cache, the STLB can be organized in many different ways — direct-mapped, direct-mapped with a victim cache [17], n-way associative, or fully associative [21].

3.2.1 Implementation

Like the PTLB, the STLB resides in unmapped, cached physical memory, and therefore does not occupy page table entries in the hardware TLB. The organization and the size of the STLB affect its hit rate. We first studied a direct-mapped organization of the STLB, with the following sizes: 1K entries (DM 1K), 4K entries (DM 4K). The direct-mapped organizations index into the STLB and check only one entry to find a match with the faulting address.

Table 3.2 presents the average time, in CPU cycles, that the direct-mapped STLB takes to service TLB misses. The two different L2 paths correspond to L2 misses that take place when in user space, and L2 misses that take place when in kernel space. A direct-mapped STLB hit takes about 115-160 cycles for L2 hits and about 100 cycles for L1K and L3 hits. The difference in timings is because Mach 3.0 replaces L2 entries using a FIFO policy, and a random replacement policy is followed for L1K and L3 entries. An STLB miss adds about 70 cycles to the trap path. On an STLB hit, this scheme decreases the overhead of the generic trap handler by providing fast access to a large number of page table entries.

Since the STLB code branches off the generic trap handler path, it adds some overhead to other traps. However, this code has been optimized so that it adds only 4 cycles to system calls and interrupts, which are by far the most frequent kinds of traps[1].

3.2.2 STLB organization

To evaluate the benefits of associative organizations of the STLB, we implemented a 2-way set-associative STLB. In the 2-way set-associative organization, the check of the first set is referred to as the *first level of associativity*, and the check of the second set as the *second level of associativity.* If the associativity were implemented in hardware then these two *levels* would be checked simultaneously. However, since the associativity is implemented in software, the *levels* are checked sequentially. We measured the average TLB miss times for this STLB organization, and found that the access time for an L1K miss in the *first level of associativity* is 105 cycles (the same as for a direct-mapped organization), while the access time for the *second level of associativity* is about 170 cycles. Thus, for the same size of STLB, if the hit rates of the two organizations are comparable, the direct-mapped organization will be faster. However, if an associative organization does provide higher hit rates, an STLB hit would still be faster than the generic trap handler, and a set-associative organization would be beneficial.

Preliminary experiments with large direct-mapped organizations showed high hit rates which were comparable to the hit rates obtained with set-associative organizations. Therefore, incurring the penalty of the software associativity was not worthwhile for the benchmarks studied. However, if later studies indicate that higher associativity results in significantly higher hit rates, then the extra penalty of implementing associativity in software could be compensated by its overall benefits.

[1]The STLB code adds only 11 cycles to the other traps, such as Bus Error, Address Error, and Floating Point exceptions, which are relatively infrequent.

3.3 Prefetching with the Software TLB

Prefetching decreases the number of L1K, L2, and L3 TLB misses, while the STLB makes TLB misses faster and eliminates cascaded misses. We integrated the two schemes to study if the joint scheme further decreases TLB penalties. Integrating the two schemes is fairly straightforward since both schemes use a table in unmapped, cached physical memory to store page table entries. Therefore, in the integrated scheme, both the prefetching and the STLB code access the same table of cached page table entries from the IPC code and the trap handler code respectively.

While prefetching and the STLB are both very effective at reducing TLB penalties, they eliminate opportunities to reduce TLB penalties from each other. This is because L3 misses are typically the result of a cascade of misses in the page table hierarchy. Since the STLB decreases the number of cascaded TLB misses, prefetching L3 entries is not very useful. In fact, it could have the negative effect of evicting useful entries from the TLB. Therefore, the integrated scheme only prefetches L1K and L2 entries on the IPC path. Since the DM 4K organization of the STLB performed well, the PTLB+STLB implementation uses a direct-mapped table with 4K entries.

3.4 Summary

In this chapter we presented our proposed TLB management schemes, and the issues involved in their implementation. Our first technique prefetches TLB entries on the IPC path between communicating processes. We used trace data to decide which TLB entries to prefetch, and we prefetch TLB entries mapping IPC data structures, and the program counter, stack pointer, and message buffers of user-level processes.

Our caching (STLB) scheme maintains a flat cache of TLB entries; thus, this scheme provides a fast path for TLB misses, and eliminates cascaded misses. We studied different organizations of the STLB, and found that a direct-mapped STLB has high hit rates and is effective in speeding up TLB misses. We also discussed the issues involved in integrating both the prefetching and caching scheme.

Chapter 4

Experiments

In this chapter, we discuss the benchmarks used to evaluate our proposed TLB management schemes, and the experimental methodology followed. We then present experimental data that demonstrates the impact of these schemes on the number of TLB misses and the time spent in handling kernel TLB misses. We found that prefetching decreases the number of kernel TLB misses by 40% to 60%, and caching decreases TLB penalties by providing a fast path for over 90% of the misses. For the applications studied, kernel TLB penalties account for 1% to 5% of overall application runtime. For these applications, our schemes improve application runtime by up to 3.5%. Finally, we present a synthetic benchmark to model applications with different runtimes, and TLB behavior. We found that increasing the number of communicating processes, and increasing the amount of data accessed by processes, increases TLB miss handling penalties. Our schemes are very effective in these scenarios, speeding up runtimes of fine-grained benchmarks by up to 10%; coarser-grained benchmarks improve by up to 5%.

4.1 Platform and Benchmarks

In this section, we discuss briefly the platform on which the experiments were conducted, and the benchmarks used to evaluate the performance of our schemes.

Our experimental platform consists of Mach 3.0 running on an R3000-based DEC-

	Kernel TLB misses (thousands)		
Application	**L1K**	**L2**	**L3**
video_play	41.5	98.4	71.1
jpeg_play	6.2	10.0	9.4
mpeg_play	46.8	116.0	82.7
IOzone	2.9	0.1	2.9
ousterhout	11.2	16.3	15.5
mab	33.3	22.4	33.9

Table 4.1: **Baseline application statistics.** This table gives the breakdown of the different types of kernel TLB misses under unmodified Mach 3.0. The counts are in thousands.

station 5000/240. In our experiments, we use the Mach 3.0 microkernel MK82, the Unix server UX41, and the X11R5 server Xcfbpmax. Thus, our experiments consist of three user-level processes: the Unix server, the X server, and the client application, all communicating through the kernel using IPC.

The R3000 has a 64-entry fully associative TLB. The hardware supports using the TLB in two partitions. The upper partition of 56 entries supports a random replacement policy and Mach uses it to hold L1U, L1K, and L3 entries. Mach uses the lower partition of 8 entries to hold L2 entries with a FIFO policy.

The benchmarks studied were `mpeg_play`, `jpeg_play`, and `video_play`, which are X applications, and `ousterhout`, `IOzone`, and `mab` (modified Andrew benchmark), which are file-system-oriented applications. A full description of the benchmarks can be found in [20]. Table 4.1 presents the number of L1K, L2, and L3 misses for each application. This data was obtained by running each application multiple times on a freshly booted system.

Since each of these applications has a different runtime, it is important to consider the rate of TLB misses. We define the *pressure* on the TLB as the rate of TLB misses. Therefore, a great pressure on the lower partition of the TLB means that the rate of L2 TLB misses is high. Similarly, a great pressure on the upper TLB partition means that the rate of L1K, L3, and L1U misses is high.

Application	Total Kernel TLB Misses	Runtime (sec)	Rate (misses/sec)
video_play	211000	37	5700
jpeg_play	25600	58	440
mpeg_play	245500	78	3150
IOzone	5900	114	50
ousterhout	43000	52	830
mab	89600	214	418

Table 4.2: **Rate of TLB misses for applications.** This table presents the rate of TLB misses for each of the benchmarks.

Table 4.2 presents the total number of TLB misses, the runtime, and the rate of TLB misses for each of the benchmarks. The video applications `video_play`, `mpeg_play`, and `jpeg_play` communicate with the X server, which in turn communicates with the UX server. Therefore, these applications communicate with the X server *and* the UX server. For these applications the TLB has to simultaneously map the client, and multiple servers. The file-system applications `mab`, `ousterhout`, and `IOzone` communicate only with the UX server. For these applications the TLB has to simultaneously map only one client and one server. This explains why the rate of TLB misses is higher for the video "multiple-server" applications `video_play`, and `mpeg_play`, as opposed to the "single-server" applications. `jpeg_play` is an exception compared to the other two video applications because it accesses much less data, while running for a substantial amount of time. `ousterhout` has a higher TLB miss rate than the other two "single-server" applications because it accesses more data.

4.2 Methodology

There are several sources of variability in application runtimes: network traffic, Mach's random page replacement policy [8], and the "aging" of the kernel [26]. To eliminate these sources of variability and obtain accurate timings, we took the following steps. First, we took the machine off the network. Second, we ran all experiments with freshly booted kernels. Third, we collected 50 data points for each benchmark.

As the system stays up for a long time the number of L1K misses increases. This is because as the kernel "ages" it starts allocating kernel data structures from mapped virtual memory. Our schemes will give increasing benefits as the number of L1K misses. However, to keep the variability in the measurements low we collected all data with freshly booted kernels.

Collecting multiple data points decreased the variability in runtime measurements. TLB penalties are a relatively small percentage of overall application runtime, therefore, it is important that the variability in measurements be small. For all the experiments conducted average application runtimes ($\bar{X}$) and standard deviations (σ) were measured. For all the data presented in this thesis, $\sigma/\bar{X}$ for `jpeg_play` was less than 0.3%, for `video_play` it was less than 0.8%, for `mpeg_play` $\sigma/\bar{X}$ was less than 1.0%, for `mab` it was less than 1.2%, for `IOzone` it was less than 1.5%, and for `ousterhout` it was less than 4.0%.

4.3 Results: TLB Miss Penalties

In this section, we present the impact of our schemes on the number of TLB misses, and the kernel TLB miss penalties. The impact of our schemes on overall runtime is presented in Section 4.4.

4.3.1 Prefetching TLB entries

We implemented our prefetching scheme under Mach 3.0 with a PTLB of size 4K entries. This size was chosen because experiments with the STLB suggested that 4K entries with a direct-mapped organization of the PTLB would achieve high hit rates.

Figure 4-1 shows the decrease in the number of kernel TLB misses for each of the three types of misses — L1K, L2, and L3. The figure indicates that prefetching TLB entries decreases the number of each type of TLB miss by about 50% for all of the applications except `IOzone` and `mab`; we discuss these two exceptions below. Table 4.3 summarizes the data presented in Figure 4-1. The last column of Table 4.3

	Kernel TLB Misses		
Application	**Mach**	**PTLB**	**Removed**
video_play	211.0	103.9	50.8%
jpeg_play	25.6	14.9	42.0%
mpeg_play	245.4	116.1	52.7%
IOzone	5.9	5.7	4.1%
ousterhout	43.0	22.2	48.5%
mab	89.6	67.0	25.2%

Table 4.3: **Kernel TLB miss counts: Mach, PTLB.** The first two columns present the total kernel TLB miss counts, in thousands. The last column presents the percentage of kernel TLB misses eliminated by prefetching.

presents the overall decrease in the number of TLB misses due to prefetching.

Prefetching TLB entries on the IPC path eliminates 40% to 50% of the kernel TLB misses for all benchmarks except `IOzone` and `mab`. As explained in Chapter 4.2, `IOzone` and `mab` communicate only with the UNIX server, whereas the video-oriented applications communicate with the UNIX server and the X server. Therefore, the pressure on the TLB is lower for these applications, as indicated by their lower rate of TLB misses. This explains why prefetching does not significantly benefit these applications. `ousterhout` has a higher rate of TLB misses; therefore, even though it is a "single-server" application it benefits from prefetching.

Figure 4-2 shows normalized kernel TLB miss penalties for each of these application. In this figure, the time taken to service kernel TLB misses under unmodified Mach is assumed to be 1. The solid gray bars represent the normalized time to service kernel TLB misses under the modified kernel with the PTLB. The striped gray bars represent the prefetching overheads added to the IPC path. Prefetching an entry into the TLB takes about 60 cycles, while probing the TLB and not prefetching an entry because it already exists takes about 25 cycles (this case occurs about twice as often as successful prefetching). The figure shows that prefetching typically eliminates about 50% of the kernel TLB miss penalties. As can also be seen, the overhead added to the IPC path is not very high for these applications.

Table 4.4 presents the measured kernel TLB penalties, in millions of CPU cycles,

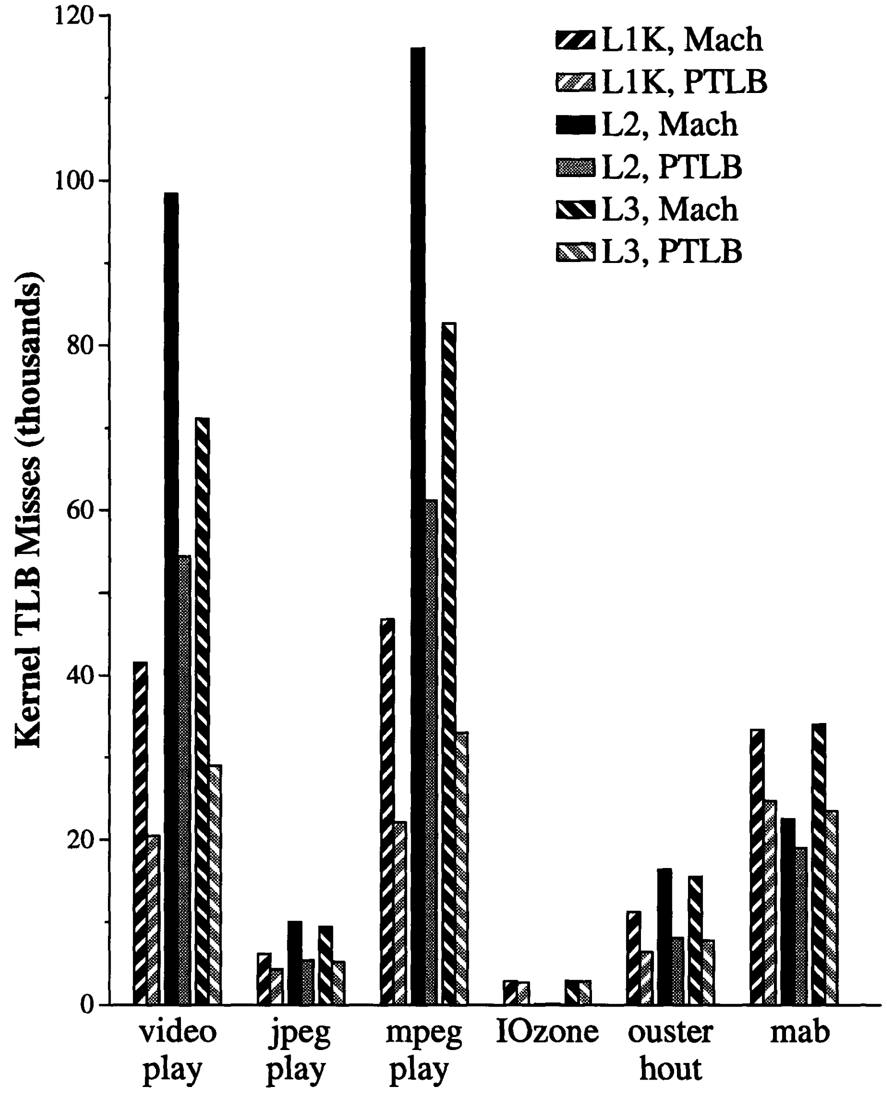

Figure 4-1: **Kernel TLB misses: Mach, PTLB.** Kernel misses under unmodified Mach are shown in black; misses under PTLB are shown in gray.

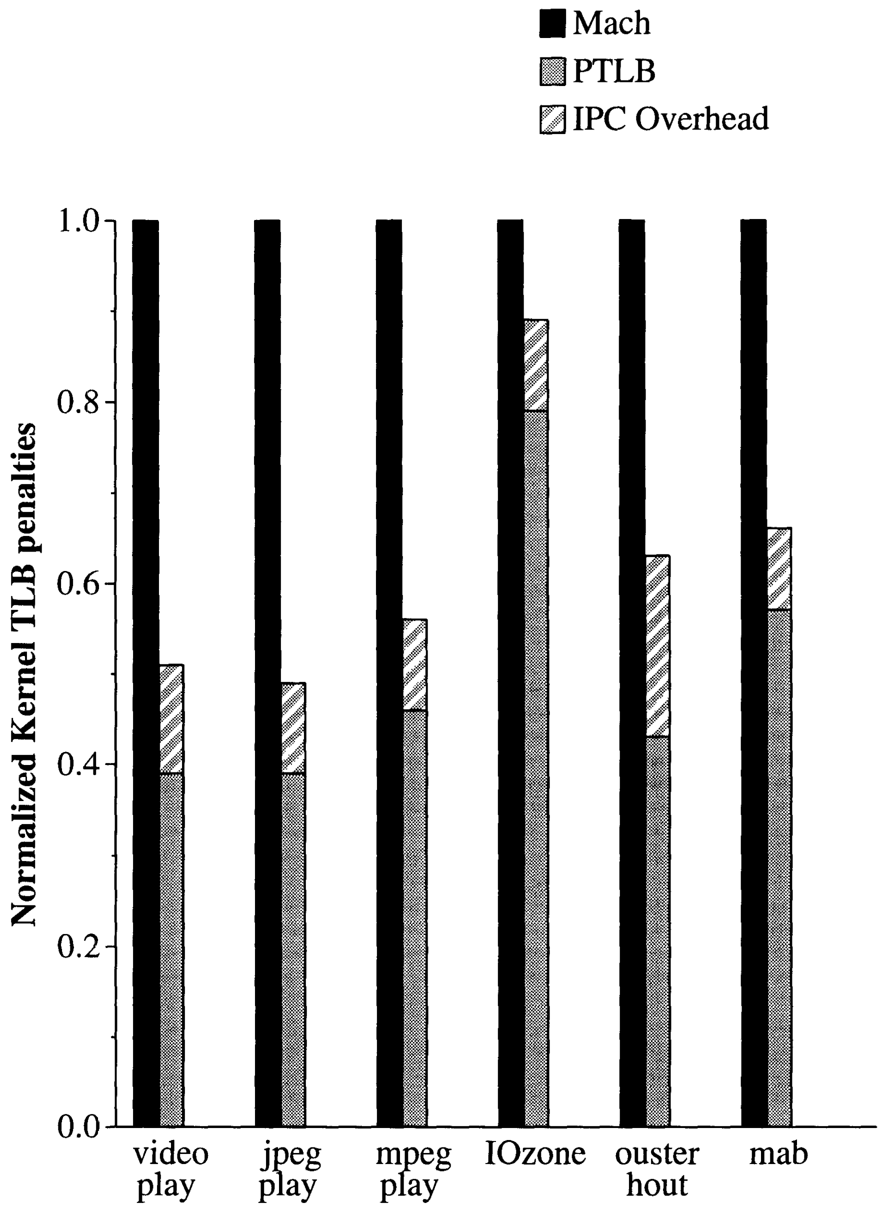

Figure 4-2: **Costs and benefits of PTLB.** Normalized kernel TLB penalties are reported; the normalization is done with respect to unmodified Mach, i.e., the time measured to service kernel TLB misses under unmodified Mach is considered to be 1. The solid gray bars show the normalized time to handle kernel TLB penalties under the modified kernel with the PTLB. The striped gray bars represent the normalized overhead added on the IPC path due to prefetching.

	Kernel TLB Penalty (million cycles)	
Application	**Mach**	**PTLB**
video_play	108.0	41.6
jpeg_play	13.0	6.0
mpeg_play	124.6	48.2
IOzone	3.4	2.7
ousterhout	28.6	17.4
mab	52.0	29.8

Table 4.4: **Kernel TLB penalties: Mach, PTLB.** The table presents the measured time to service kernel TLB misses under unmodified Mach and PTLB, in millions of CPU cycles.

for these applications under Mach and PTLB. Kernel TLB misses typically constitute greater than 50% of the overall TLB penalties in an application. The remaining penalties are due to L1U and TLB-invalid misses. Even though L1U misses are highly optimized, their frequency is so high that they contribute significantly to overall TLB penalties.

In summary, prefetching benefits all applications and decreases TLB overheads significantly, usually eliminating about 50% of the kernel TLB miss penalties.

4.3.2 Software Cache of TLB Entries

The experimental environment and benchmarks for the STLB are the same as those for the PTLB, as described in Section 4.1. As discussed in Section 3.2.2, the direct-mapped organizations were found to have very high hit rates and so set-associative organizations were not explored further. In the results presented below, DM 1K refers to a direct-mapped organization of the STLB with 1K entries. Similarly DM 4K refers to a direct-mapped organization of the STLB with 4K entries.

Table 4.5 presents the number of kernel TLB misses for the two STLB configurations and the associated STLB hit rates. The number of kernel TLB misses under unmodified Mach 3.0 has been included for comparison. The hit rates achieved by DM 1K range from 70% to 99%. The hit rates achieved by DM 4K are higher, and

	Mach	DM 1K		DM 4K	
Application	**Total**	**Total**	**Hit rate**	**Total**	**Hit rate**
video_play	211.0	202.4	70.9%	163.3	100.0%
jpeg_play	25.6	20.2	84.4%	17.4	99.8%
mpeg_play	245.4	197.5	82.5%	173.8	97.2%
IOzone	5.9	4.4	98.5%	4.3	99.3%
ousterhout	43.0	33.2	85.9%	29.0	99.3%
mab	89.6	88.0	70.6%	63.9	92.5%

Table 4.5: **TLB miss counts and hit rates under Mach and STLB.** This table presents total kernel misses in thousands under unmodified Mach 3.0, and the two direct-mapped STLB configurations, DM 1K and DM 4K. Hit rates for the STLB configurations are also presented. DM 1K is the direct-mapped STLB with 1K entries, while DM 4K is the direct-mapped STLB with 4K entries.

are close to 100%.

The two STLB configurations also result in an overall decrease in the number of kernel TLB misses. This is because when a TLB miss hits in the STLB, the cascade of TLB misses to higher levels of the page table hierarchy is eliminated. Table 4.6 shows that under Mach 3.0 nearly 30% of the kernel TLB misses for each application are such cascaded misses. Due to its higher hit rate the larger STLB eliminates almost all of these cascaded misses.

Figure 4-3 presents the normalized kernel TLB penalties under the different STLB configurations and unmodified Mach. The striped bars represent the overhead imposed by the STLB on system calls. Table 4.7 presents the unnormalized kernel TLB penalties under unmodified Mach and the two STLB configurations. As is evident from the figure and the table, DM 4K reduces TLB penalties significantly.

4.3.3 Prefetching with the Software TLB

Table 4.8 presents the counts of kernel TLB misses under unmodified Mach and PTLB+STLB, and the percentage of kernel TLB misses eliminated. The last column presents the STLB hit rates. The hit rates are lower than those obtained for the STLB configuration DM 4K in Table 4.5. This is because the total number of kernel

	Cascaded Misses (thousands)		
Application	Mach	DM 1K	DM 4K
video_play	61.4	35.8	0.0
jpeg_play	7.9	1.0	0.0
mpeg_play	72.3	13.1	3.3
IOzone	1.7	0.0	0.0
ousterhout	13.1	2.9	0.0
mab	26.4	12.6	0.2

Table 4.6: **Cascaded TLB misses, in thousands.** Cascaded TLB misses, in thousands, under unmodified Mach 3.0, and the two direct-mapped STLB configurations, DM 1K and DM 4K.

	TLB penalty (million cycles)		
Application	Mach	DM 1K	DM 4K
video_play	108.0	60.7	15.7
jpeg_play	13.0	4.2	1.9
mpeg_play	124.6	41.7	21.4
IOzone	3.4	0.7	0.6
ousterhout	24.0	7.9	3.9
mab	52.0	31.6	13.2

Table 4.7: **Kernel TLB penalties: Mach, STLB.** Kernel TLB penalties under Mach 3.0, and the two direct-mapped STLB configurations, DM 1K and DM 4K, in millions of CPU cycles.

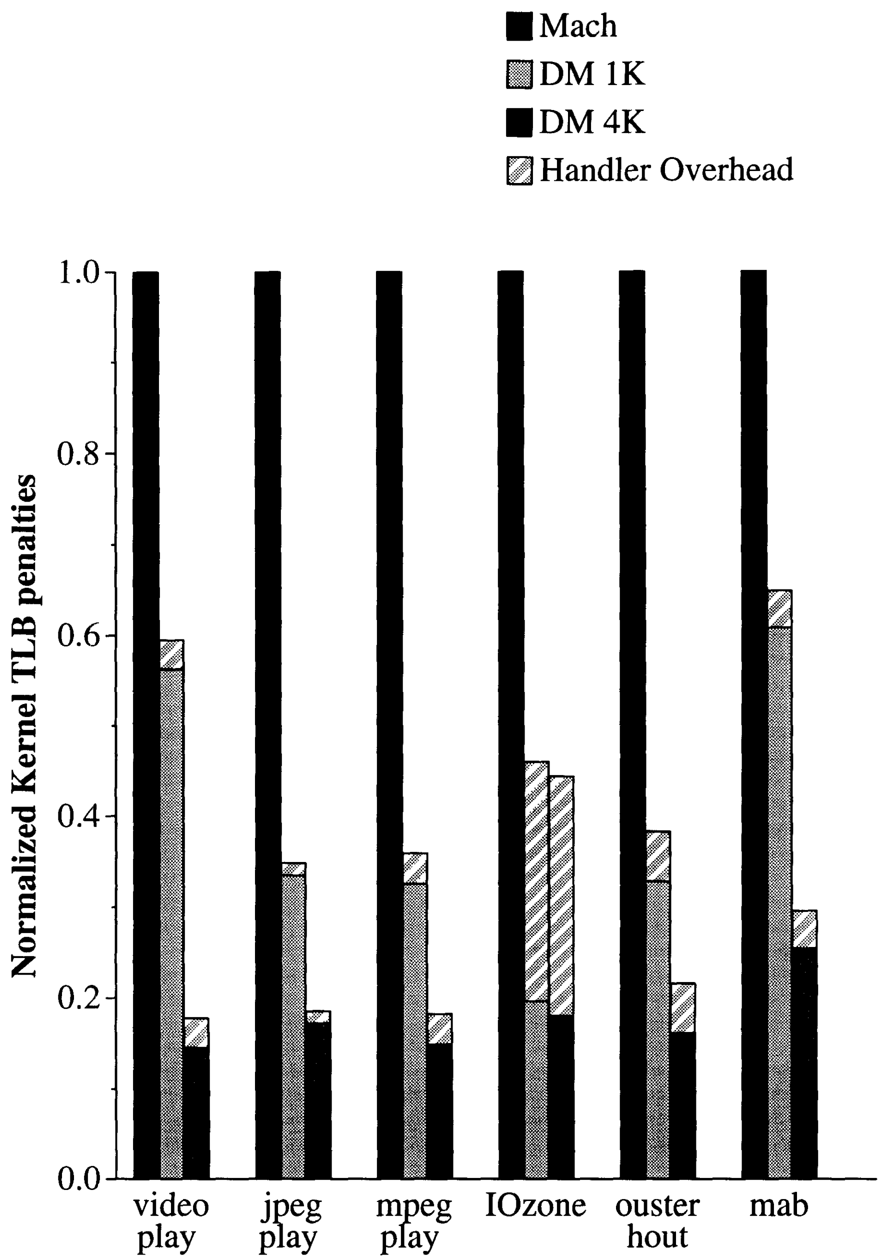

Figure 4-3: **Costs and benefits of STLB.** Normalized TLB penalties are reported. The dark gray bars represent the normalized kernel TLB penalties for the larger STLB configuration, while the lighter gray bars correspond to the smaller STLB configuration. The striped gray bars indicate the normalized measured overhead added. The overhead is the same for both STLB configurations, since it depends on the number of system calls, and is independent of the STLB size.

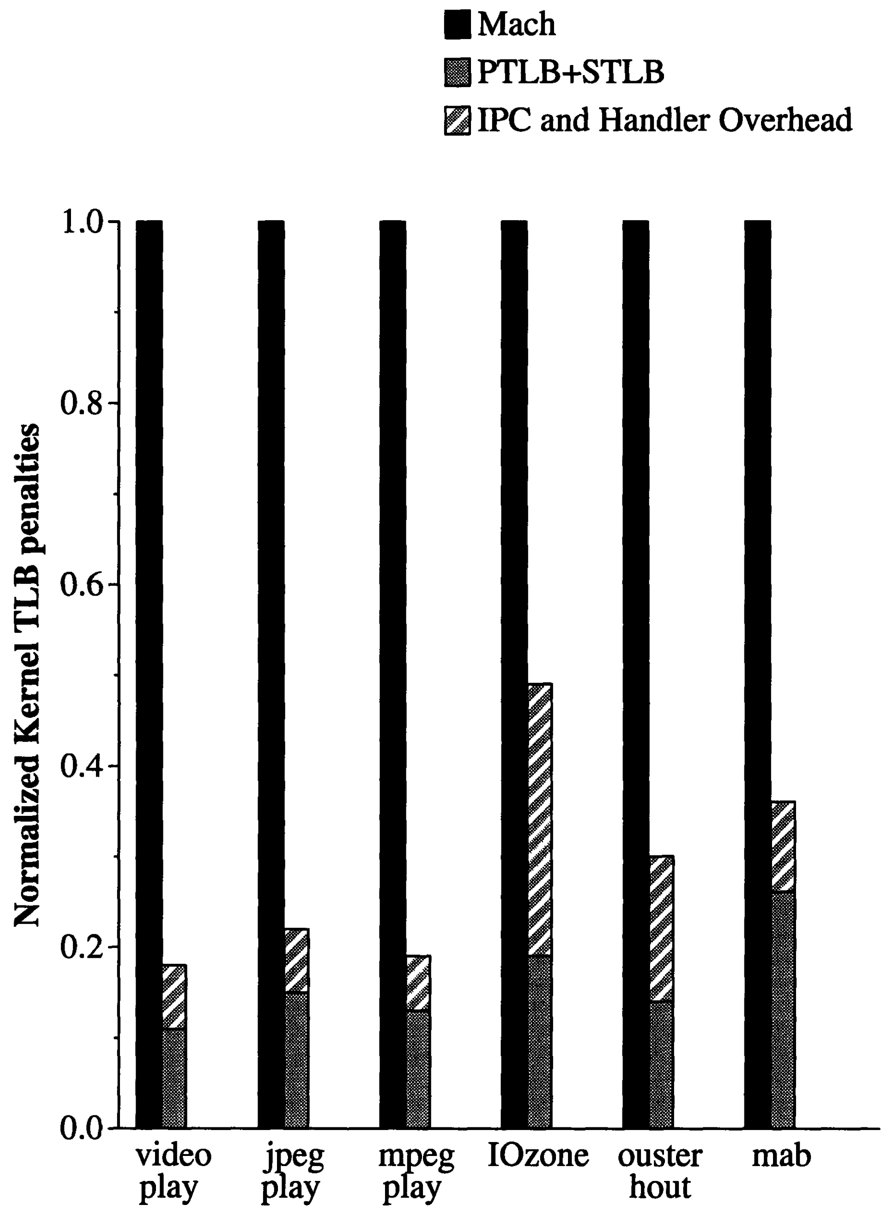

Figure 4-4: **Costs and benefits of PTLB+STLB.** Normalized TLB penalties are reported. The gray bars represent the normalized kernel TLB penalties for the PTLB+STLB. The striped gray bars indicate the normalized measured overhead added to the IPC path by prefetching, and the trap handler path by the STLB.

	Kernel TLB Misses			
Application	Mach	PTLB+ STLB	Percent Removed	Hit Rate
video_play	211.0	111.8	47.0%	100.0%
jpeg_play	25.6	13.8	46.2%	99.7%
mpeg_play	245.4	128.6	47.6%	96.2%
IOzone	5.9	4.1	31.2%	99.3%
ousterhout	43.0	21.1	50.9%	99.1%
mab	89.6	52.8	41.1%	90.9%

Table 4.8: **Kernel TLB miss counts: Mach, PTLB+STLB.** Total kernel TLB miss counts, in thousands, under Mach and PTLB+STLB. The third column presents the percentage of kernel TLB misses eliminated by the STLB and prefetching, and the last column gives the hit rate for the STLB.

	Kernel TLB Penalty (million cycles)	
Application	Mach	PTLB+STLB
video_play	108.0	11.4
jpeg_play	13.0	1.6
mpeg_play	124.6	18.4
IOzone	3.4	0.6
ousterhout	24.0	3.4
mab	52.0	13.6

Table 4.9: **Kernel TLB penalties: Mach, PTLB+STLB.** The kernel TLB penalties are measured in millions of CPU cycles.

TLB misses is decreased due to prefetching. Figure 4-4 shows normalized kernel TLB miss penalties under unmodified Mach in black. The solid gray bars represent the normalized kernel miss penalties under the PTLB+STLB, and the striped gray bars indicate the normalized overheads added by prefetching and the STLB. Table 4.9 presents the measured kernel TLB penalties, in millions of CPU cycles, under Mach and the PTLB+STLB. As seen in Figure 4-4 and Table 4.9, the integrated scheme is very effective in decreasing TLB penalties.

		STLB		
Application	**Prefetch**	**DM 1K**	**DM 4K**	**PTLB+STLB**
video_play	3.52%	2.18%	3.53%	3.04%
jpeg_play	0.91%	0.15%	0.21%	0.27%
mpeg_play	1.69%	0.27%	1.73%	1.09%
IOzone	0.19%	0.85%	0.59%	0.99%
ousterhout	0.77%	0.76%	1.53%	1.65%
mab	0.96%	0.76%	0.80%	0.25%

Table 4.10: **Application Speedup.** The overall application speedup obtained by the proposed schemes is presented in this table. Speedup is measured as the percentage of application time saved by using our schemes.

4.4 Results: Application Speedup

In this section, we present the impact of our proposed schemes on overall application performance. Speedup is measured as the decrease in application runtime as a percentage of the application runtime under unmodified Mach. As seen from Table 4.10, prefetching results in speedups of up to 3.5% of overall application runtime; the larger STLB, DM 4K, also provides speedups of up to 3.5%. DM 1K does not perform as well for all applications, due to its lower hit rate. The integrated scheme performs well, improving application performance by up to 3% of overall application runtime. The integrated scheme does not outperform the other two schemes for these applications, because it incurs the overhead of both schemes. The synthetic benchmark presented in the next section motivates the importance of the integrated scheme.

For these applications, kernel TLB penalties account for 1% to 5% of overall application runtime [28]. Therefore, our proposed schemes eliminate a significant fraction of these penalties.

4.5 Synthetic benchmark

To better understand the impact of our proposed schemes on a range of applications, we implemented a synthetic benchmark which we present in this section. We wanted to capture the TLB behavior of a range of applications with different data access patterns, different number of communicating processes, and different runtimes. We study applications with different runtimes by classifying applications on the basis of their "granularity". A fine-grain process communicates very frequently, while a coarser-grain process computes more, and communicates less frequently.

Keeping the above goal in mind, the parameters that can be specified to this synthetic benchmark are as follows:

- Number of servers and clients.
- Amount of data accessed by a user process.
- The granularity of a user process.

Each run of this benchmark consists of a number of iterations of a client randomly selecting a server to communicate with. If a client picks a server i out of the N servers, the communication pattern for that iteration of the benchmark is $Client \rightarrow Server_i \rightarrow Server_{i+1} \rightarrow .... \rightarrow Server_N$. This pattern, of a chain of servers communicating with each other, is similar to the pattern of an application communicating with the X server, which in turn communicates with the UX server. Other patterns of communication can also be specified to the benchmark, however, we expect the above pattern to be the typical pattern of communication in multi-server environments.

We used this benchmark to test the following hypotheses:

- **Hypothesis 1:** The number of user processes affects the number of TLB misses.
- **Hypothesis 2:** The amount of data accessed by a user process affects the number of TLB misses.
- **Hypothesis 3:** The granularity of communicating processes impacts the effect of TLB miss handling on overall runtime.

4.5.1 Hypothesis 1

We wanted to test the hypothesis that the number of user processes affects the total number of TLB misses. Therefore, we considered the case of a single client communicating with N servers, where N varies from 1 to 10. Figure 4-5 presents the number of TLB misses under unmodified Mach and each of the proposed schemes, for each of the cases considered. The number of data elements accessed by a process is fixed at 30 data elements per IPC. This value corresponds to the case of some benchmarks presented in earlier sections.

From Figure 4-5 we see that when the number of servers increases from one to two, the number of TLB misses increases dramatically. This is because when there is only one server, the lower partition of the TLB has to map only one client and one server. Since each process requires not more than four entries (one each for its code, stack, data, message buffers), all the L2 entries of the two processes fit in the lower partition of the TLB. As the number of processes increases, the pressure on the lower partition keeps increasing. The prefetching scheme does uniformly well over the range of servers, eliminating 60% to 65% of the total kernel TLB misses. The STLB eliminates 25% to 30% of kernel TLB misses by eliminating cascaded misses. The PTLB+STLB scheme benefits both from prefetching and the elimination of cascaded misses by the STLB. Therefore, the number of kernel TLB misses under the PTLB+STLB scheme is the lowest of all, until the number of servers exceeds seven. At seven servers, the hit rates of the STLB and the PTLB+STLB decrease because of their direct-mapped organization; when the number of servers exceeds seven, both these schemes start providing decreasing benefits.

The TLB misses under the STLB and the PTLB+STLB depicted in Figure 4-5 are primarily serviced through hits in the STLB. Therefore, they are less expensive than the TLB misses in the prefetching scheme. To understand the impact on overall kernel TLB miss penalties, Figure 4-6 presents the normalized kernel TLB penalties for each scheme and the associated overheads. All the schemes are very effective in decreasing overall kernel TLB penalties. The prefetching scheme does uniformly well. As mentioned above, the STLB and PTLB+STLB scheme do very well until the number of servers exceeds seven after which they provide decreasing benefits. However, for the entire range of servers, the PTLB+STLB scheme does the best, decreasing TLB penalties by 70% to 80%.

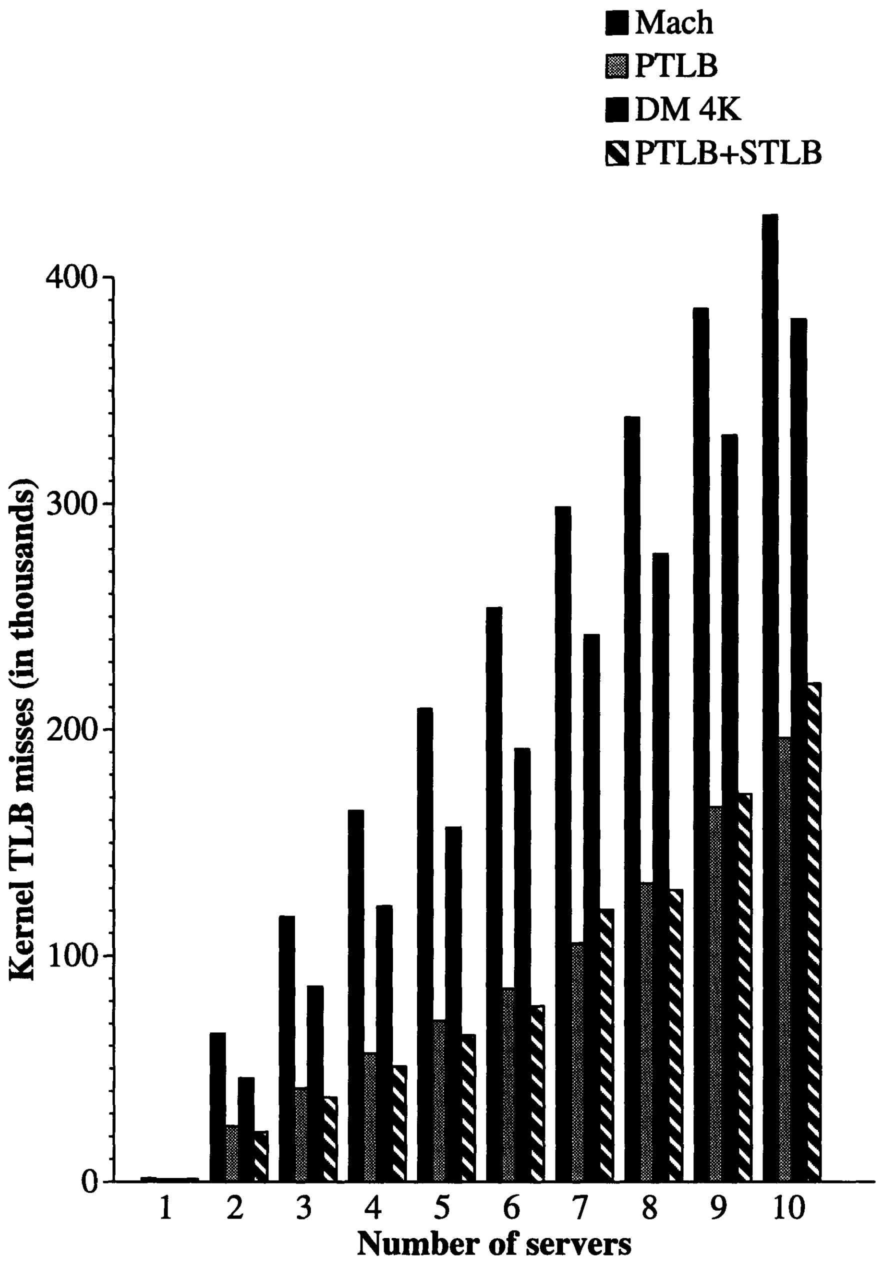

Figure 4-5: **Increase in the number of servers: TLB miss counts.** This figure depicts the number of total kernel TLB misses under Mach 3.0 and each of the proposed schemes when the number of servers is increased.

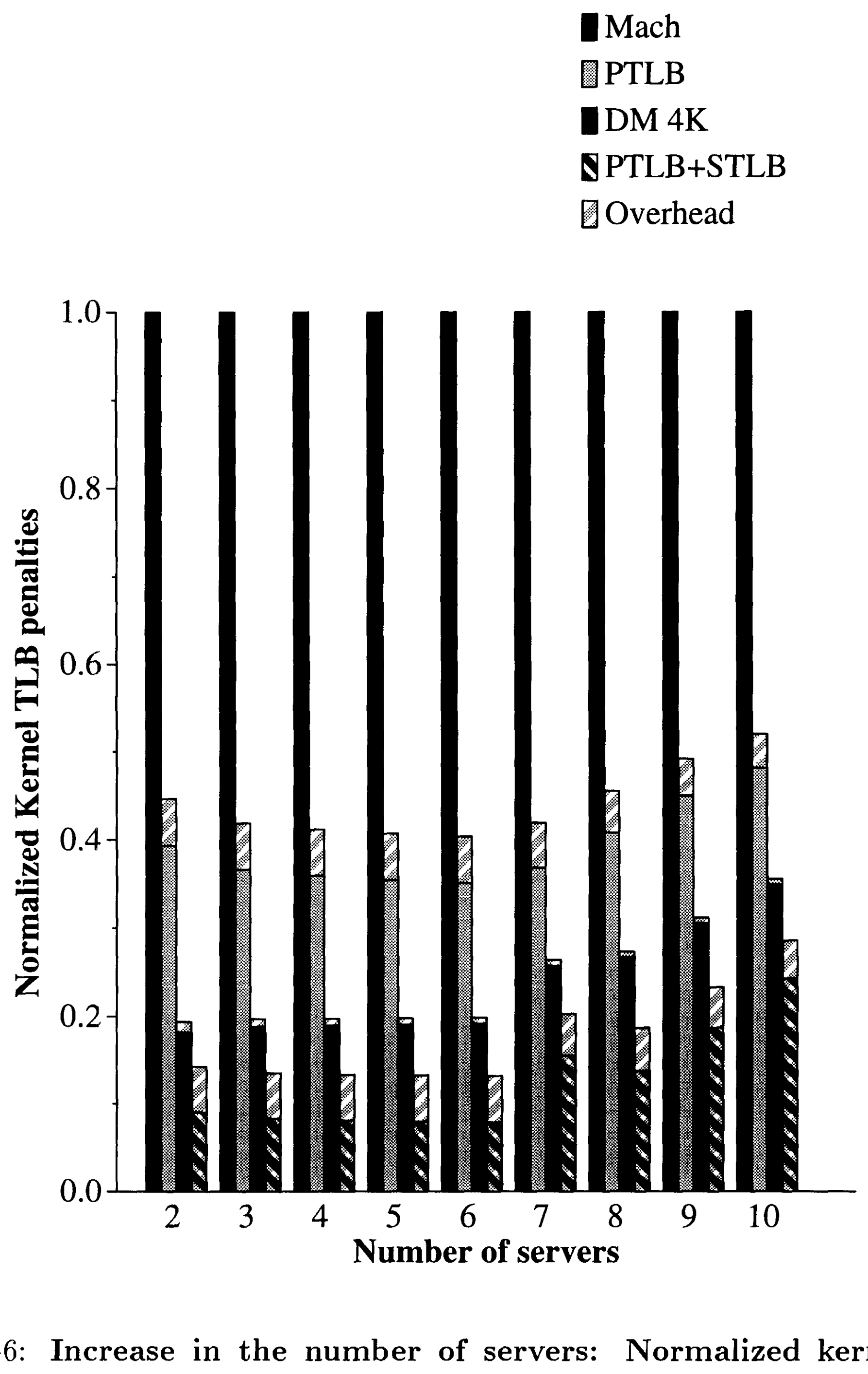

Figure 4-6: **Increase in the number of servers: Normalized kernel TLB penalties.** This figure depicts the normalized kernel TLB penalties under Mach 3.0 and each of the proposed schemes when the number of servers is increased.

To study the impact of an increase in the number of client processes on TLB miss handling, we ran a similar experiment with an increase in the number of clients while keeping the number of servers fixed at three and four. We found that the qualitative trend of the results was the same. However, increasing the number of clients does not increase the number of TLB misses as fast as increasing the number of servers. This is because Mach permits clients to handoff control to the server they are sending requests to. Thus, in one scheduling quantum, control can be passed several times between a client and a server before the client gets descheduled and the next client is selected to run. Therefore, client execution is not as finely interleaved as the servers, thus relieving some of the pressure on the TLB.

As mentioned above, the STLB achieves very high hit rates, nearly 100%, until the number of servers becomes greater than seven. Then the STLB hit rate starts dropping. This is a consequence of the limited capacity and the direct-mapped organization of the STLB. To explore if a set-associative organization would help in performance we ran our experiments with a 2-way set-associative STLB. We found that the set-associative organization provided slightly higher hit rates than the direct-mapped organization for our experiment with one client and multiple servers. However, in our experiment with a fixed number of servers and multiple clients, the set-associative organization provided greater hit rates than the direct-mapped organization. Thus, depending on the pattern of TLB accesses the 2-way set-associative organization could provide greater benefits than the direct-mapped organization.

We conclude that increasing the number of user-level processes increases the number of TLB misses. It should be noted that techniques which propose using superpages will suffer from the same problems. This is because the code, data, and stack segment of a process cannot be contiguously mapped in physical memory. Therefore, the number of L2 misses will be the same for those techniques. Our schemes are very effective in decreasing the number of TLB misses as the number of user processes increases. At some point our schemes also reach their limitations, but this is to be expected given the finite capacity and associativity of software schemes.

4.5.2 Hypothesis 2

We wanted to test the hypothesis that the amount of data accessed by a user-level process affects the number of TLB misses. To test this hypothesis we ran two experiments; the first experiment consisted of one client communicating with three servers, the second experiment consisted of one client communication with four servers. In each experiment we varied the amount of data accessed by each user process. We present below the results for the 4-server case (the 3-server case exhibited the same qualitative trends).

A measure of the amount of data accessed by a process is the number of L1U TLB misses per IPC. This captures the extent to which each process uses the TLB before making an implicit context switch through IPCs. For the applications presented in Section 4.1, the number of L1U misses per IPC ranged from 10 to 40. In our experiments each user process randomly selects a data element from a 4MB table in its address space. A 4MB table consists of 1024 4KB pages. A random access into such a large table is likely to result in an L1U miss. Therefore, the number of words accessed per IPC in our experiment is likely to be the same as the number of L1U misses per IPC. In our experiments the data words accessed per IPC ranged from 5 to 50.

Figure 4-7 presents the total number of kernel TLB misses under unmodified Mach and the proposed schemes. As the amount of data accessed by a process increases, the number of L1U misses increases. Therefore, a greater number of L1K, L1U, and L3 entries are evicted from the upper partition. This effect puts more pressure both on the upper and lower partition of the TLB. Beyond the point of 40 data elements accessed per IPC, the upper partition of the TLB is being almost completely overwritten on every IPC, and the number of kernel TLB misses saturates. Therefore, as the amount of data accessed increases, the rate of increase in TLB misses decreases.

Prefetching eliminates about 60% to 65% of the kernel TLB misses. The STLB eliminates about 25% of kernel TLB misses. The hit rates achieved by the STLB and the PTLB+STLB are very high (almost 100%). Figure 4-8 presents the normalized kernel TLB penalties for each scheme. Prefetching decreases kernel TLB miss penal-

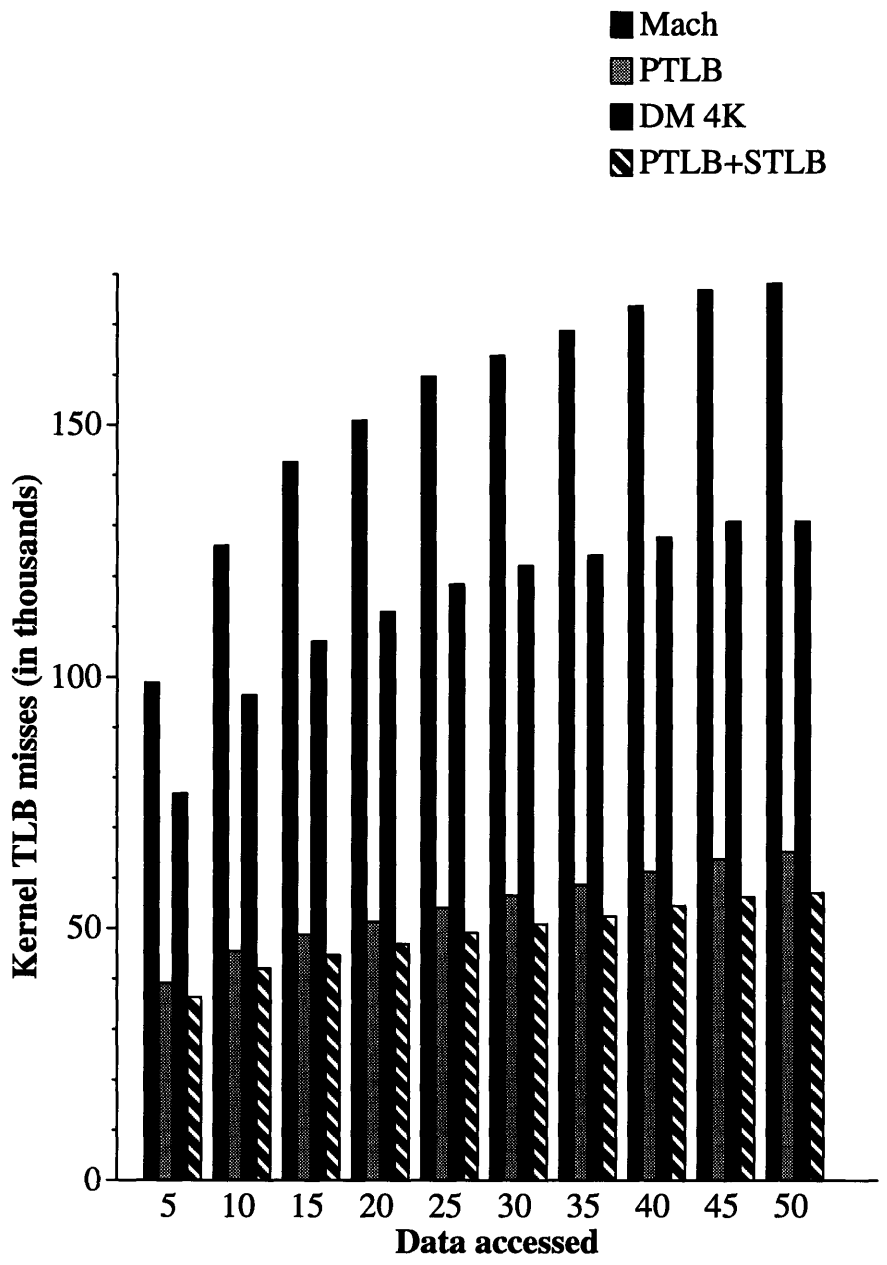

Figure 4-7: **Increase in data accessed: kernel TLB miss counts.** This figure depicts the total number of kernel TLB misses under Mach 3.0 and each of the proposed schemes when the amount of data accessed by each process is increased. The X-axis depicts the number of words accessed by each process per IPC.

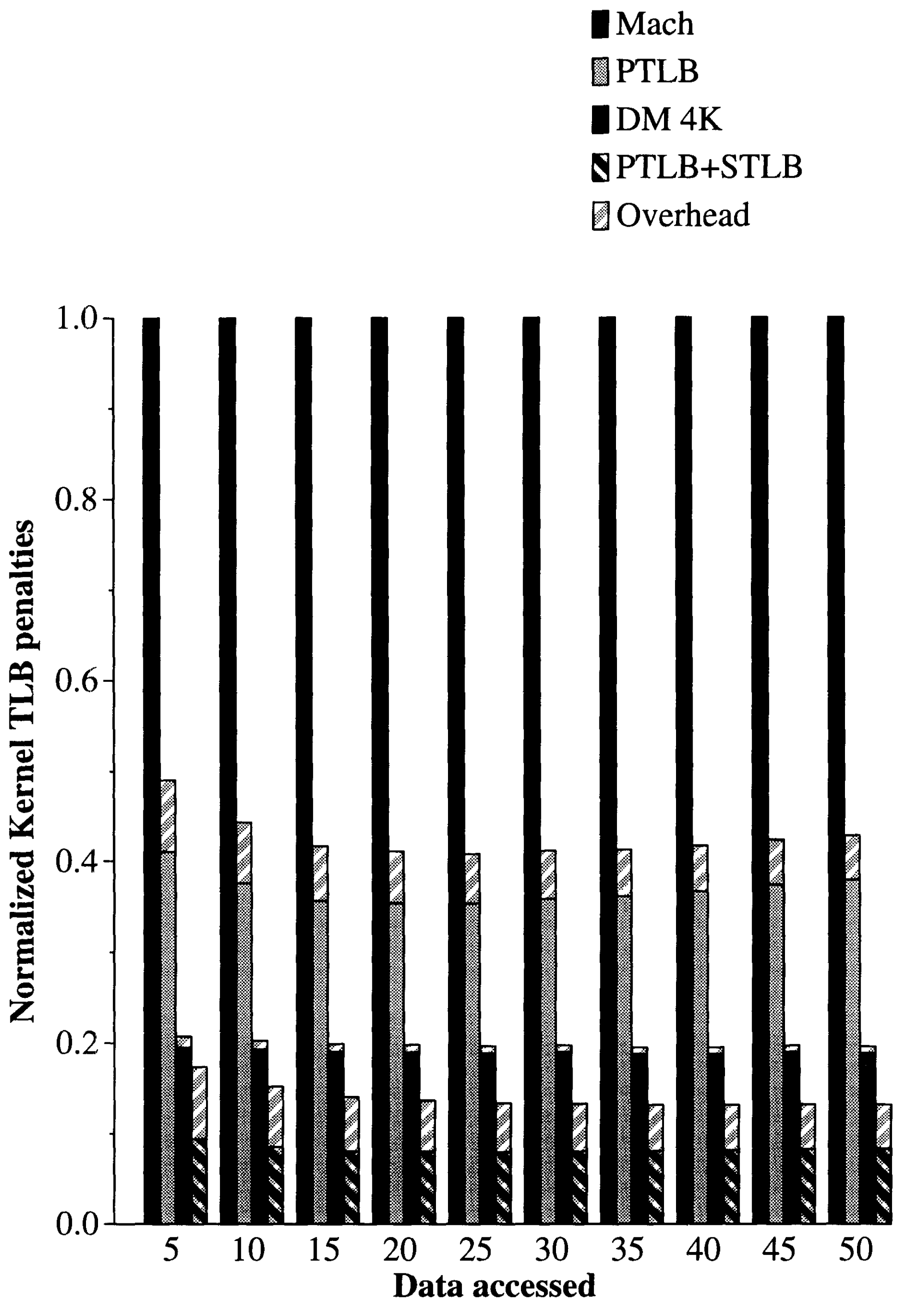

Figure 4-8: **Increase in data accessed: Normalized kernel TLB penalties.** This figure depicts the normalized kernel TLB penalties under Mach 3.0, and each of the proposed schemes when the amount of data accessed is increased. The X-axis depicts the number of words accessed by a user-level process per IPC.

	Fine Granularity		
Servers	PTLB	DM 4K	PTLB+STLB
3	3.0%	8.3%	6.8%
4	2.3%	6.4%	6.4%
5	2.3%	10.2%	7.5%
6	4.8%	7.1%	10.3%
7	5.1%	7.5%	9.6%
8	3.6%	7.8%	11.0%
9	2.7%	7.4%	9.5%
10	2.3%	6.0%	9.8%

Table 4.11: **Speedup for fine-grained applications.** This table presents the speedup achieved by each of the proposed schemes, for fine-grained applications.

ties by about 50% to 55%, the STLB decreases TLB penalties by about 80%, and the integrated PTLB+STLB decreases TLB penalties by about 80% to 85%.

We conclude that as the data accessed by user-level processes increases, the number of TLB misses increases and the proposed schemes are effective in decreasing kernel TLB penalties.

4.5.3 Hypothesis 3

We wanted to test the hypothesis that the granularity of a process affects the impact of the TLB management schemes on the overall runtime. The granularity of an application is determined by the amount of computation the process performs between communications. Fine-grained applications are very IPC-intensive, communicating often; coarser-grained applications have a lower rate of IPCs. The granularity of an application impacts its overall runtime. We measure the granularity of an application in terms of the time to make an IPC, referred to as T_{IPC}. Our benchmark models fine-grained applications, which continuously makes IPCs to the server, medium-grained applications which compute for 2 T_{IPC} before making an IPC call, and coarse-grained applications which compute for 5 T_{IPC} before making an IPC. Each of these benchmarks accesses 30 words of data between IPCs.

For the multiple-server experiment the application speedups for the fine-grained

	Medium Granularity		
Servers	PTLB	DM 4K	PTLB+STLB
3	2.4%	3.2%	3.2%
4	2.5%	3.2%	5.5%
5	3.0%	4.3%	6.6%
6	2.2%	3.6%	7.4%
7	1.9%	4.3%	6.3%
8	1.6%	3.7%	6.9%
9	1.7%	4.6%	5.6%
10	1.1%	3.2%	5.9%

Table 4.12: **Speedup for applications with medium granularity.** This table presents the speedup achieved by each of the proposed schemes, for applications of medium granularity.

	Coarse Granularity		
Servers	PTLB	DM 4K	PTLB+STLB
3	1.8%	2.8%	3.8%
4	1.8%	2.9%	4.4%
5	1.8%	2.4%	4.8%
6	1.5%	2.0%	3.6%
7	0.5%	2.2%	4.0%
8	0.4%	2.0%	4.4%
9	0.0%	2.1%	3.9%
10	0.3%	2.2%	3.2%

Table 4.13: **Speedup for coarse-grained applications.** This table presents the speedup achieved by each of the proposed schemes, for coarse-grained applications.

benchmarks are listed in Table 4.11, for the medium-grained benchmarks in Table 4.12, and for the coarse-grained benchmarks in Table 4.13. We see that as the granularity of an application increases the benefits of the TLB management schemes decreases. For very fine-grained benchmarks the benefits are high ranging from 6% to 10%. The STLB, and the integrated scheme do very well when compared to the prefetching scheme because they speedup all TLB misses. As the number of servers increases, the integrated scheme outperforms the STLB because it decreases the total number of TLB misses to a greater extent than the STLB. For coarse-grained benchmarks, the benefits of these schemes range from 2% to 4.5%.

We ran the experiment with a fixed number of clients and servers and varying amounts of data, with all three granularities mentioned above. The results obtained were qualitatively the same.

We conclude that the granularity of the benchmark affects the impact of the TLB handling schemes on overall application runtime. The integrated scheme performs the best of all the schemes; for coarse-grained benchmarks it improves benchmark runtimer by 3% to 5%, while for fine-grained benchmarks it improves performance by up to 11%.

4.5.4 Conclusions

In this section we implemented a synthetic benchmark to study the impact of our TLB management schemes on applications with different data access patterns, different number of communicating processes, and different runtimes. We used the benchmark to test several hypothesis about the TLB behavior of applications.

We found that increasing the number of clients and servers increases the pressure on the TLB. Increasing the data accessed by each user-level process also increases the pressure on the TLB. In these cases our schemes, especially the integrated scheme, are very effective in decreasing kernel TLB penalties. We modelled applications with different runtimes using our synthetic benchmarks, and found that for coarse-grained benchmarks our schemes improve benchmark runtimes by 3% to 5%, while for fine-grained benchmarks they improve performance by up to 11%.

Chapter 5

Analytical Model of TLB Schemes

In this chapter we present an analytical model of application runtime and TLB miss handling penalties. The model captures the architecture and application parameters which determine the time spent by an application in handling TLB misses. As CPU speeds increase faster than memory speeds we expect that TLB miss penalties will become an increasing fraction of application runtime. Also, we expect that improvements in caching techniques on modern architectures will benefit applications more than the TLB handler [8]. Our goal is to use our model to understand how these architectural trends will impact TLB miss handling times and application runtimes.

5.1 Parameters of Model

In this model the parameters to characterize the runtime of an application are:

- The number of instructions that the application executes: I_{App}.
- The number of instructions (excluding TLB miss handling) that the operating system executes: I_{OS}.
- The number of instructions executed in handling TLB misses: I_T.
- The number of cache misses induced by an application: C_{App}. The cache miss rate for an application is denoted by M_{App} and is given by C_{App}/I_{App}. (Note:

The "standard" definition of cache miss rate is the fraction of cache accesses that miss. M, as expressed above, captures the number of cache misses per instruction.)

- The number of cache misses induced by the operating system activity excluding the TLB miss handler: C_{OS}. The cache miss rate for the OS is denoted by M_{OS} and is given by C_{OS}/I_{OS}.
- The number of cache misses induced by the TLB Miss Handler: C_T. The cache miss rate for the TLB miss handler is denoted by M_T and is given by C_T/I_T.
- The time, in cycles, to service a cache miss on an architecture: T_C.
- The *TLB speedup*, denoted by S_T, captures the effect of the proposed TLB management schemes on the TLB miss handling time. S_T is the percentage decrease in TLB management time; i.e., if a TLB management scheme, for example prefetching, decreases the TLB penalties by 50%, $S_T = 0.5$. Note that the *speedup* is with respect to TLB miss handling time and not with respect to the entire application runtime.

5.2 Model

In this model, the runtime of an application (in cycles) is expressed as:

$$\begin{aligned} R &= I_{App} + C_{App}T_C + I_{OS} + C_{OS}T_C + I_T + C_TT_C \\ &= I_{App}(1 + M_{App}T_C) + I_{OS}(1 + M_{OS}T_C) + I_T(1 + M_TT_C) \end{aligned}$$

We wish to concentrate on the TLB miss handling performance, therefore, we merge the application and the OS component, and now use the subscript A to refer to the application+OS component.

Therefore,

$$R = I_A(1 + M_AT_C) + I_T(1 + M_TT_C) \tag{5.1}$$

δ denotes the benefit of a particular TLB management scheme, like prefetching or the STLB. δ is the decrease in the TLB miss handling time divided by the runtime of the application under the unmodified kernel. Using the notation presented above, the time saved in application runtime is: $S_T \times$ *Time to handle TLB misses.*

Therefore,

$$\begin{aligned}\delta &= \frac{(S_T \times Time\ to\ handle\ TLB\ misses)}{R} \\ &= \frac{S_T\ I_T(1 + M_T T_C)}{I_A(1 + M_A T_C) + I_T(1 + M_T T_C)} \qquad (5.2)\end{aligned}$$

5.3 δ on Different Architectures

The goal is to obtain values of δ on different architectures to study the importance of TLB miss handling on modern architectures. The experiments conducted in this thesis used a MIPS R3000-based machine. We instantiate the above equations to this architecture, denoted by the superscript *R3K*. We denote the modern new architecture with the superscript of *new*.

Instantiating equations 5.1 and 5.2 for the R3000 and *new* architecture we get,

$$R^{R3K} = I_A^{R3K}(1 + M_A^{R3K} T_C^{R3K}) + I_T^{R3K}(1 + M_T^{R3K} T_C^{R3K}) \qquad (5.3)$$

$$R^{new} = I_A^{new}(1 + M_A^{new} T_C^{new}) + I_T^{new}(1 + M_T^{new} T_C^{new}) \qquad (5.4)$$

$$\delta^{R3K} = \frac{S_T^{R3K} I_T^{R3K}(1 + M_T^{R3K} T_C^{R3K})}{I_A^{R3K}(1 + M_A^{R3K} T_C^{R3K}) + I_T^{R3K}(1 + M_T^{R3K} T_C^{R3K})} \qquad (5.5)$$

$$\delta^{new} = \frac{S_T^{new}\ I_T^{new}(1 + M_T^{new} T_C^{new})}{I_A^{new}(1 + M_A^{new} T_C^{new}) + I_T^{new}(1 + M_T^{new} T_C^{new})} \qquad (5.6)$$

The difference in the cache miss rates M^{new} and M^{R3K} captures the change in the size and organization of the caches for the two architectures.

In order to evaluate δ^{new} we have to know I^{new}, M^{new}, T_C^{new}, and S_T^{new}. One way of doing this is to use δ^{R3K} as a starting point, and make assumptions comparing I^{new} with I^{R3K}, M^{new} with M^{R3K}, T_C^{new} with T_C^{R3K}, and S_T^{new} with S_T^{R3K}.

5.3.1 Assumptions

Different instruction sets of the two architectures could make the instruction counts vary dramatically. However, for RISC-based architectures the difference should not be as great. Another factor that impacts the time to execute I instructions is the move to dual-issue architectures. To simplify the model, we will assume that the number of instructions executed remains the same, i.e. I_A^{new} is the same as I_A^{R3K}.

The new architecture might have a different organization of TLB miss handling when compared to the MIPS R3000. For example, both the Alpha and the R3000 have a specialized fast trap for L1U misses. Additionally, the Alpha has a specialized fast trap for L1K misses. However, for this model we will assume that I_T^{new} is the same as I_T^{R3K}.

We also assume that $S_T^{new} = S_T^{R3K}$. This assumption seems reasonable because the effectiveness of the TLB mangement schemes to decrease TLB miss penalties should not be different on an architecture with a similar TLB miss handling organization.

There are two cases to be considered: when the cache miss rate M on the new architecture is the same as on the R3000, and when it is different. The rest of the chapter will deal with these two cases.

5.4 Same Cache Miss Rate

In this section we present the model assuming that the cache miss rate remains the same on the two architectures being studied. This assumption simplifies the equations considerably and helps in gaining an insight into the parameters which affect the behavior of TLB miss handling. We conclude this section with a prediction of the benefits of the TLB management schemes on a faster architecture, using some realistic values for the model parameters.

Now we study the case where the *new* architecture has the same processor and memory architecture as the MIPS R3000 but operates at a different clock speed, (i.e., the *new* architecture has a faster CPU). M remains the same since the memory organization does not change, i.e., cache sizes and organization are the same. Therefore, since CPU speeds increase faster than memory speeds, the only change will be that $T_C^{R3K} < T_C^{new}$.

If the cache miss rate remains the same, δ^{new} and δ^{R3K} are of the form,

$$\begin{aligned}\delta^{new} &= \frac{(a + bT_C^{new})}{(a' + b'T_C^{new})}\\ \delta^{R3K} &= \frac{(a + bT_C^{R3K})}{(a' + b'T_C^{R3K})}\end{aligned}$$

where,

$$\begin{aligned}a &= S_T I_T\\ b &= S_T C_T\\ a' &= I_T + I_A\\ b' &= C_T + C_A\end{aligned}$$

$$\begin{aligned}\delta^{new} &= \delta^{R3K} + \frac{a + bT_C^{new}}{a' + b'T_C^{new}} - \frac{a + bT_C^{R3K}}{a' + b'T_C^{R3K}}\\ &= \delta^{R3k} + \frac{(a'b - b'a)(T_C^{new} - T_C^{R3K})}{R^{new}R^{R3K}}\\ \delta^{new} &= \delta^{R3K} + \frac{S_T}{R^{new}R^{R3K}}(I_A C_T - I_T C_A)(T_C^{new} - T_C^{R3K})\end{aligned}$$

In terms of the cache miss rate,

$$\delta^{new} = \delta^{R3K} + \frac{S_T I_T I_A}{R^{new}R^{R3K}}(M_T - M_A)(T_C^{new} - T_C^{R3K}) \tag{5.7}$$

We conclude that on a faster architecture with the same cache organization and instruction set, δ increases if the cache miss rate of the TLB handler is greater than the cache miss rate of the application+OS.

The intuitive explanation of this result is as follows: on a faster architecture with a slower memory speed relative to the CPU, the benefits of the TLB management schemes will increase if the TLB component of runtime is affected more than the application component of runtime by the slower cache miss time.

Predictions

To make some predictions about the faster architecture we instantiate Equation 5.7 with some realistic values of the parameters and study the impact of these parameter settings on δ.

- $S_T = 0.5$. This value corresponds to a 50% decrease in TLB handling penalties due to a TLB management scheme, for example, prefetching.
- $I_A = 0.5R^{new}$. This value implies that we estimate the number of application+OS instructions to be about 50% of the application runtime on the faster architecture. The rest of the time is spent in cache miss servicing and TLB miss handling.
- $I_T = 0.06R^{R3K}$. For the applications we have studied, I_T is about 50% of TLB miss handling times, and the TLB miss handling times are about 5-15% of the application runtime. Therefore, this value of I_T seems reasonable.
- $M_T - M_A = 0.05$. This difference in the cache miss rate of the TLB and the application is fairly conservative. If the application exhibits better cache performance this difference gets even higher. This value corresponds to some applications in [8].

We see that,

$$\delta^{new} = \delta^{R3K} + 0.015(M_T - M_A)(T_C^{new} - T_C^{R3K})$$

Substituting for $M_T - M_A$,

$$\delta^{new} = \delta^{R3K} + 0.00075(T_C^{new} - T_C^{R3K}) \tag{5.8}$$

Equation 5.8 gives a rough rule of the benefits of the TLB management schemes for an application on a faster architecture with the same cache miss rate. From this equation we expect that on a faster architecture the TLB management schemes provide increasing benefits proportional to the change in the cache miss times of the two architectures.

We use Equation 5.8 to study the impact of TLB miss handling on an extension of the MIPS R3000 architecture, denoted by the superscript *R3K-ext*. *R3K-ext* has the same processor and memory architecture as the MIPS R3000, but it has a faster CPU. T_C^{R3K} is 24 cycles [9]. $T_C^{R3K-ext}$ is the time to service a cache miss on a machine with a faster CPU of 100 MHz. Assuming that the memory system is the same on the faster architecture, $T_C^{R3K-ext} = 60$. Substituting in Equation 5.8 we get,

$$\delta^{R3K-ext} = \delta^{R3K} + 0.027 \tag{5.9}$$

In terms of the percentage of application runtime, the benefits of the TLB schemes on the faster architecture increase by 2.7% of the application runtime.

To summarize, in this section we used the model to compare two architectures with the same CPU and memory architecture but with different clock speeds. We also used the model to predict the impact that the TLB management schemes have on application performance on such architectures.

5.5 Different Cache Miss Rate

In this section we use the model to study architectures which have different cache sizes/organizations, and different clock speeds; i.e., this section considers the effect that the change in the cache miss rate has on δ. We use the model to predict the impact of the TLB management schemes on these new architectures. Specifically, we will use our model to study the impact of TLB miss handling on a modern architecture with architectural parameters similar to Digital's Alpha. We will use the superscript α to denote this *new* architecture, and as earlier the MIPS R3000 will be denoted by *R3K*.

$$\begin{aligned}
\delta^{\alpha} &= \delta^{R3K} + \frac{S_T I_T(1 + M_T^{\alpha} T_C^{\alpha})}{I_A(1 + M_A^{\alpha} T_C^{\alpha}) + I_T(1 + M_T^{\alpha} T_C^{\alpha})} \\
&\quad - \frac{S_T I_T(1 + M_T^{R3K} T_C^{R3K})}{I_A(1 + M_A^{R3K} T_C^{R3K}) + I_T(1 + M_T^{R3K} T_C^{R3K})} \\
&= \delta^{R3K} + \frac{S_T I_T I_A}{R^{\alpha} R^{R3K}} [T_C^{\alpha}(M_T^{\alpha} - M_A^{\alpha}) - T_C^{R3K}(M_T^{R3K} - M_A^{R3K}) \\
&\quad + T_C^{\alpha} T_C^{R3K}(M_T^{\alpha} M_A^{R3K} - M_A^{\alpha} M_T^{R3K})]
\end{aligned} \tag{5.10}$$

We can check that if the cache miss rate remains the same, i.e., $M_A^\alpha = M_A^{R3K}$, and $M_T^\alpha = M_T^{R3K}$, then the above formula works out to be the same as Equation 5.7.

A modern faster architecture might have a bigger on-chip cache or a better cache organization. The effect of this change in cache size/organization is that M_A^α and M_T^α could be less than the values of M_A^{R3K} and M_T^{R3K} respectively. Let $M_A^\alpha = \gamma_A \times M_A^{R3K}$ (where $0 \leq \gamma_A \leq 1$), and $M_T^\alpha = \gamma_T \times M_T^{R3K}$ (where $0 \leq \gamma_T \leq 1$). We expect that $\gamma_A \leq \gamma_T$, because the TLB handler code is invoked infrequently and we expect that the TLB handler will not benefit as much as the application and the OS from the better/bigger cache.

Note: a large value of γ means the better/bigger cache is not helping much. A small value of γ means the cache miss rate has been reduced by the better/bigger cache. In the extreme case, $\gamma = 1$ means the better/bigger cache did not make any difference, and $\gamma = 0$ means the better/bigger cache *completely* eliminate cache misses.

Substituting the above in Equation 5.10 and dropping the superscript $R3K$ on M, we get,

$$\begin{aligned}\delta^\alpha &= \delta^{R3K} + \frac{S_T I_T I_A}{R^\alpha R^{R3K}}[T_C^\alpha(\gamma_T M_T - \gamma_A M_A) - T_C^{R3K}(M_T - M_A) \\ &\qquad + T_C^\alpha T_C^{R3K}(\gamma_T M_T M_A - \gamma_A M_A M_T)] \\ &= \delta^{R3K} + \frac{S_T I_T I_A}{R^\alpha R^{R3K}}[M_T(\gamma_T T_C^\alpha - T_C^{R3K}) - M_A(\gamma_A T_C^\alpha - T_C^{R3K}) \\ &\qquad + T_C^\alpha T_C^{R3K} M_T M_A(\gamma_T - \gamma_A)] \qquad (5.11)\end{aligned}$$

To further analyze Equation 5.11 we consider different scenarios. These scenarios study the impact of γ on the value of δ. We expect that $\gamma_T > \gamma_A$, and therefore, the second scenario studies this case.

Scenario 1 (Worst Case): $\gamma_A = \gamma_T = \gamma$

This scenario corresponds to the case where both the TLB handler and the application cache miss rates are affected in the same way by the better/bigger cache. With this assumption we find that,

$$\delta^{\alpha} \quad = \quad \delta^{R3K} + \frac{S_T I_T I_A}{R^{\alpha} R^{R3K}}(M_T - M_A)(\gamma T_C^{\alpha} - T_C^{R3K}) \tag{5.12}$$

To intuitively understand Equation 5.12 we note that on a faster architecture, if we assume that the TLB and the application component are affected in the same way by the new cache organization (i.e., same γs), then δ increases if:

- $M_T > M_A$, i.e., the cache miss rate of the TLB component of runtime is greater than the cache miss rate of the application component. This implies that on a faster architecture the TLB handler component will be affected more than the application component by the the slower cache access time. Therefore, δ will increase.

- $\gamma > T_C^{R3K}/T_C^{\alpha}$. If γ is larger than T_C^{R3K}/T_C then the better/bigger cache does not provide very large benefits. Therefore, the faster architecture with the slower cache access time is greatly impacted by cache behavior and δ increases.

  However, if the faster architecture with the better/bigger cache has a greatly reduced number of cache misses, i.e., γ is small, then cache misses have less of an impact on the overall performance of the application (when compared to the old architecture). Therefore TLB miss penalties are not as important as on the old architecture, and δ decreases. In the worst case, $\gamma = 0$, i.e. all cache misses are eliminated: in this case, $\delta^{\alpha} < \delta^{R3K}$ because the TLB component of runtime *decreases* on a faster architecture and therefore, the TLB schemes provide less benefits. The break even point $\delta^{\alpha} = \delta^{R3K}$ is achieved when $\gamma = T_C^{R3K}/T_C^{\alpha}$.

We reproduce Equation 5.7 here for ease of reference.

$$\delta^{new} \quad = \quad \delta^{R3K} + \frac{S_T I_T I_A}{R^{new} R^{R3K}}(M_T - M_A)(T_C^{new} - T_C^{R3K}) \tag{5.7}$$

Comparing this equation with Equation 5.12 we see that the equations are basically the same except for the γ factor in Equation 5.12. The intuition is that if the application and the TLB are both affected equally by the better/bigger cache, the lower the value of γ the less is the importance of the TLB management schemes.

We use the above equation to compare the TLB behavior of an application on the MIPS R3000 with an architecture similar to Digital's Alpha. The superscript $R3K$ denotes the MIPS R3000 40 MHz chip, and the superscript α denotes a 150 MHz chip with architectural parameters similar to Digital's Alpha. We instantiate this equation using the same assumptions as those made for Equation 5.8 and get,

$$\delta^{\alpha} \;=\; \delta^{R3K} + 0.00075(\gamma T_C^{\alpha} - T_C^{R3K}) \tag{5.13}$$

T_C^{R3K} as before is 24 cycles; $T_C^{\alpha} = 60$ cycles [10]. It should be noted that this value takes into account the fact that when building faster chips architects also improve the memory system of the computer. Substituting in Equation 5.13, and assigning a value of $\gamma = 0.8$ we get,

$$\delta^{\alpha} \;=\; \delta^{R3K} + 0.018$$

This is lower than what was obtained in Equation 5.9 because the better/bigger caches make TLB miss handling less of a problem. From this equation we learn that the benefits of the TLB management schemes increase by 1.8%. Therefore, for an application such as `video_play`, for which δ^{R3K} was 3.5%, $\delta^{\alpha} \;=\; 5.3\%$.

However, this scenario of $\gamma_A = \gamma_T$ is not very realistic. As mentioned earlier, the TLB handler is unlikely to benefit from the better cache organization as much as the application. Therefore, we expect that $\gamma_T > \gamma_A$ and study this scenario below.

Scenario 2 (Realistic case): $\gamma_T > \gamma_A$

This scenario studies the case where applications exhibit greater locality than the TLB handler code. TLB misses are infrequent, and therefore, when invoked the TLB handler code tends not to hit in the cache. However, we expect that applications benefit to a greater extent by the improved cache. This statement is not always true and is application-dependent.

We reproduce Equation 5.11 here for convenience.

$$\begin{aligned}\delta^{\alpha} &= \delta^{R3K} + \frac{S_T I_T I_A}{R^{\alpha} R^{R3K}}[M_T(\gamma_T T_C^{\alpha} - T_C^{R3K}) - M_A(\gamma_A T_C^{\alpha} - T_C^{R3K}) \\ &\quad + T_C^{\alpha} T_C^{R3K} M_T M_A(\gamma_T - \gamma_A)] \qquad (5.14)\end{aligned}$$

We have already studied the worst case for the TLB management schemes of $\gamma_T = \gamma_A$ in Scenario 1. We now study the best case for the TLB management schemes:

Best case: $\gamma_T = 1; \gamma_A = 0$

This case corresponds to the case when the entire application fits in the better/bigger cache. However, the TLB handler has the same cache miss rate as earlier (i.e., the better/bigger cache does not help the TLB miss handling code). Substituting in Equation 5.11 we get,

$$\begin{aligned}\delta^{\alpha} &= \delta^{R3K} + \frac{S_T I_T I_A}{R^{\alpha} R^{R3K}}[M_T(T_C^{\alpha} - T_C^{R3K}) + M_A T_C^{R3K} \\ &\quad + T_C^{\alpha} T_C^{R3K} M_T M_A] \qquad (5.15)\end{aligned}$$

Using the same values used in obtaining Equation 5.8 and setting $M_T = 0.07$ and $M_A = 0.02$, we get

$$\delta^{\alpha} = \delta^{R3K} + 0.00075$$

Therefore, for an application such as `video_play`, for which δ^{R3K} is 3.5%, δ^{α} would be 11%. Thus, in this scenario the TLB management schemes can improve performance by up to 11% of the application performance. We should note however that this is an ideal case.

Realistic cases: $\gamma_A = 0.5$

We expect that bigger and better organized cache would not eliminate *all* application cache misses. Therefore, depending on the application, $\gamma_A = 0.5$ is a more realistic value. γ_T depends on several parameters like the application, and the conflicts in the cache, but we expect $\gamma_T > \gamma_A$. We explore two different values of γ_T, and the impact γ_T has on δ.

$\gamma_T = 1$: In this case,

$$\delta^\alpha = \delta^{R3K} + 0.051$$

Therefore, for an application such as `video_play`, for which δ^{R3K} is 3.5%, δ^α is 8.6%.

$\gamma_T = 0.8$: In this case,

$$\delta^\alpha = \delta^{R3K} + 0.032$$

Therefore, for `video_play`, δ^α is 6.7%. Thus, even in more realistic cases we see that the TLB management schemes can provide greater benefits on a faster architecture.

5.6 Conclusions

In this chapter we have studied an analytical model which captures the architecture and application parameters which characterize the benefits of the TLB management schemes proposed in this thesis. We further analyzed the model to compare the TLB behavior of an application on a fast architecture with the same cache organization as a slow architecture. Also, we studied the impact of moving to architectures with better/bigger cache organizations and predicted the behavior of the TLB management schemes for some realistic values of the parameters of the model.

Chapter 6

Related Work

Prefetching and caching are two well-known techniques that have been widely applied in computer systems [22, 24], but have not been applied to software management of TLBs. Most of the previous work has focussed on hardware caches.

Huck and Hays [15] have explored an idea similar to the STLB in the context of the HP PA-RISC. Their *Hashed Page Table* (HPT) is a page table organization suited for large address spaces. The faulting virtual address is used to index into the HPT and a match is attempted between the faulting address and the HPT entry. If no match is found, a fault is taken to the OS. Collision entries are then looked up in software. This scheme replaced an earlier one in the HP-UX operating system, which used an inverted page table organization. Huck and Hays simulated both hardware and software implementations of the HPT.

The work in this thesis, which has been implemented on a different architecture, complements Huck and Hays work in a number of ways. The entirely software-based implementation is in the context of hierarchically-organized forward-mapped page tables and we study the effect of nested TLB traps. We obtained measurements from a running system for a range of applications and found that a direct-mapped organization of the STLB was very effective. Experiments showed that a 2-way set-associative software STLB adds more overhead, as compared to a simple direct-mapped organization. Further, our work studies the benefit of prefetching TLB entries in the context of a microkernel-based OS environment with several communicating processes, where

TLB performance is more critical.

Another complementary approach to decreasing TLB miss handling costs is to provide variable-sized pages called "superpages"; thus, a single TLB entry can map large contiguous regions of memory. The PowerPC architecture [16] supports the use of superpages. Researchers [23] have looked at the architectural and OS support required to effectively use superpages.

Our proposed schemes deal with several issues not addressed by the "superpage" approach. A process's program counter, stack pointer, and message buffers, are allocated at different places in the process's address space. It is not practical to allocate the code, stack, and data segments of these processes contiguously in physical memory. Therefore, superpages will not benefit these entries and they will still need different L2 entries in the TLB. Our schemes are effective in dealing with such TLB misses.

As operating systems move to modular organizations supporting finer-grained processes interacting with each other, the number of L2 TLB misses will increase. Since the TLB is a shared resource, contention for the TLB will be high. As our synthetic benchmark indicated, our schemes continue to decrease TLB penalties as the number of user-level processes increases.

Our schemes are also useful for dynamically allocated data structures, and in systems supporting garbage collection, where ensuring that all important data structures are contiguous in physical memory is not easy. Thus, our schemes can complement the superpage solution in reducing TLB miss penalties.

Chapter 7

Conclusions

A number of interacting trends in operating system structure, processor architecture, and memory systems are increasing both the rate of translation lookaside buffer (TLB) misses and the cost of servicing a TLB miss. This thesis presents two entirely software-based solutions to decrease TLB penalties, and the issues involved in their implementation.

The first scheme prefetches TLB entries on the IPC path between communicating processes. Trace data indicated that prefetching TLB entries which map IPC data structures, and the program counter, stack pointer, and message buffers of user-level processes, is beneficial. For a range of applications, prefetching decreases the number of kernel TLB misses by 40% to 60%.

The second scheme maintains a flat cache of TLB entries (STLB). This scheme provides a fast path for TLB misses, and eliminates cascaded misses. For the application we studied, eliminating cascaded misses reduces the total number of TLB misses by about 20% to 40%. We also found that a direct-mapped organization of the STLB achieves very hit rates of greater than 90% for a range of applications.

For these applications, TLB penalties range from 1% to 5% of application runtime; our schemes are very effective in reducing kernel TLB penalties, improving application performance by up to 3.5%.

Using synthetic benchmarks, we also demonstrated the impact of increasing the number of client/server processes, and increasing the data accessed by a process, on

TLB miss handling times. For these synthetic benchmarks, our schemes, especially the integrated scheme with both prefetching and caching, perform very well improving runtimes for fine-grained benchmarks by up to 10%.

Processor speeds continue to increase relative to memory speeds; we developed a simple analytical model which indicates that our schemes should be even more effective in improving application performance on future architectures.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Summer USENIX*, pages 93–113, Atlanta, GA, June 1986.

[2] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth Conference on Architectural Support for Programming Languages and Systems*, pages 108–119, Santa Clara, CA, April 1991.

[3] B. Bershad, C. Chambers, S. Eggers, C. Maeda D., McNamee, P. Pardyak, S. Savage, and E. Sire. SPIN - an extensible microkernel for application-specific operating system services. Technical Report TR94-03-03, University of Washington, February 1994.

[4] B.N. Bershad. The increasing irrelevance of IPC performance for microkernel-based operating systems. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 205–211, Seattle, WA, April 1992.

[5] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 102–113, Litchfield Park, AZ, December 1989.

[6] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso. *Programming under Mach.* Addison-Wesley Publishing Company, 1993.

[7] J.S. Chase, H.M. Levy, E.D. Lazowska, and M. Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *Proceedings of the Conference on Object-*

Oriented Programming Systems, Languages and Applications, pages 397–413, Vancouver, Canada, October 1992.

[8] B. Chen and B. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 120–133, Asheville, NC, December 1993.

[9] Digital Equipment Corporation. *DECstation and DECsystem 5000 Model 240 Technical Overview.* Digital Equipment Corporation, 1991.

[10] Digital Equipment Corporation. *DEC3000 Model 500/500S Technical Summary.* Digital Equipment Corporation, 1992.

[11] Digital Equipment Corporation. *The Ultrix Operating System, V4.4.* Digital Equipment Corporation, 1994.

[12] Microsoft Corporation. *Microsoft OLE programmer's reference.* Microsoft Press, 1993.

[13] D. Engler, M. F. Kaashoek, and J. O'Toole. The operating system kernel as a secure programmable machine. In *Proceedings of the 6th European SIGOPS Workshop*, Germany, September 1994.

[14] W. Hsieh, M. F. Kaashoek, and W. E. Weihl. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, pages 186–190, Napa, CA, October 1993.

[15] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 39–50, San Diego, CA, May 1993.

[16] Motorola Inc. *PowerPC 601: RISC Microprocessor User's Manual.* Prentice-Hall, Inc., 1993.

[17] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990.

[18] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 175–188, Asheville, NC, December 1993.

[19] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: a distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[20] D. Nagle, R. Uhlig, T. Mudge, and S. Sechrest. Optimal allocation of on-chip memory for multiple-API operating systems. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 358–369, Chicago, IL, April 1994.

[21] D. Patterson and J. Hennessy. *Computer architecture: a quantitative approach.* Morgan Kaufman Publishers, 1989.

[22] D. Patterson and J. Hennessy. *Computer organization and design: the hardware/software interface.* Morgan Kaufman Publishers, 1993.

[23] M. Talluri and M. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the Sixth Conference on Architectural Support for Programming Languages and Systems*, pages 171–182, San Jose, CA, October 1994.

[24] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, Inc., 1992.

[25] OpenDoc Design Team. OpenDoc technical summary. In *Apple's World Wide Developers Conference Technologies CD*, San Jose, CA, April 1994.

[26] R. Uhlig. Private communication, 1993.

[27] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. Software TLB management in OSF/1 and Mach 3.0. Technical report, University of Michigan, 1993.

[28] Richard Uhlig, David Nagle, Tim Stanley, Trevor Mudge, Stuart Sechrest, and Richard Brown. Design tradeoffs for software managed TLBs. *ACM Transactions on Computer Systems*, 12(3):175–205, August 1994.

[29] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, December 1993.