Appears in *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994

Software Prefetching and Caching for Translation Lookaside Buffers

Kavita Bala * M. Frans Kaashoek * William E. Weihl*
MIT Laboratory for Computer Science
Cambridge, MA 02139, USA

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 miss. This paper 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 50%, and caching decreases TLB penalties by providing a fast path for over 90% of the misses. Our caching scheme also decreases the number of nested TLB traps due to the page table hierarchy, reducing the number of kernel TLB miss traps for applications by 20% to 40%. Prefetching and caching, when used alone, each improve application performance by up to 3.5%; when used together, they improve application performance by up to 3%. On synthetic benchmarks that involve frequent communication among several different address spaces (and thus put more pressure on the TLB), prefetching improves overall performance by about 6%, caching improves overall performance by about 10%, and the two used together improve overall performance by about 12%.

Our techniques are very effective in reducing kernel TLB penalties, which currently range from 1% to 5% of application runtime for the benchmarks studied. Since processor speeds continue to increase relative to memory speeds, our schemes should be even more effective in improving application performance in future architectures.

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 relative cost of servicing a miss. Microkernel-based operating systems achieve modularity and flexibility by providing OS functionality in user-level server processes. Frequent communication between client processes, the kernel, and server processes results in more TLB misses than in monolithic-kernel systems [17]. 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.

Several modern RISC architectures (*e.g.*, the MIPS R2000/3000, the DEC Alpha, and the HP PA-RISC) handle TLB misses in software. This simplifies the hardware considerably, and provides greater flexibility to the operating system. However, the penalty for a TLB miss increases. This penalty becomes even more significant as CPU speeds increase relative to memory speeds, since the cost of accessing page table entries is growing [18].

As researchers continue to improve inter-process communication (IPC) performance [4, 15], TLB penalties will become an increasing fraction of the IPC cost [3, 11]. In addition, recent commercial standards, such as OLE [9] and OpenDoc [21], place an increasing emphasis on inter-application communication. The net effect of all these factors is that the impact of TLB misses on overall system performance is increasing [1, 7].

While TLB penalties have been recognized as a problem, proposed solutions typically require expensive hardware [17, 16]. We present two schemes to address this problem that rely only on software mechanisms; no additional hardware is required. Our techniques reduce both the number and 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. By prefetching these entries in the kernel during the IPC, many misses can be avoided.

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 sys-

*E-mail: {kaybee, kaashoek, weihl}@lcs.mit.edu. World Wide Web URL: http://www.psg.lcs.mit.edu/. Prof. Weihl is currently supported by DEC while on sabbatical at DEC SRC. This research was supported in part by the Advanced Research Projects Agency under Contract N00014-94-1-0985, by grants from AT&T and IBM, and by an equipment grant from DEC. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

tems 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 significantly. To the best of our knowledge, this paper is the first to present a technique to reduce nested TLB traps in hierarchical page table organizations. A scheme similar to the STLB was simulated with a different organization of the hardware TLB and the page tables [12], and is discussed with related work in Section 7.

We have implemented our prefetching and caching techniques in the Mach 3.0 kernel running on a MIPS R3000-based DECstation 5000/240. Prefetching TLB entries benefited the entire range of applications we examined, decreasing the number of *kernel TLB misses* (*i.e.*, misses on kernel data structures, user page tables, and kernel page tables) by 40% to 50%. This improves overall application performance by up to 3.5%. The penalties of kernel TLB misses for these applications range from 1% to 5% [16]; benchmarks in other studies have higher TLB penalties [12]. Thus, prefetching eliminates a significant fraction of kernel TLB penalties. The software TLB achieves high hit rates for kernel TLB misses that range from 90% to nearly 100%. It also decreases the number of kernel TLB misses by eliminating nested TLB traps. As with prefetching, this improves performance by up to 3.5%.

Using our two techniques together improves application performance by up to 3%. On synthetic benchmarks that involve frequent communication among several different address spaces (and thus put more pressure on the TLB), prefetching improves overall performance by about 6%, caching improves overall performance by about 10%, and the two used together improve overall performance by about 12%. Thus, for systems that use more frequent communication, we expect our techniques to provide increasing benefits, and provide even greater benefits when used together.

Although currently the impact of TLB miss handling on overall application performance is small, TLB miss handling penalties are expected to have a larger impact on overall application performance on newer architectures. Our proposed techniques reduce both the number of misses and the cost of each miss, and we expect both techniques to provide increasing benefits on future architectures.

Our experiments assume a microkernel-based operating system. However, our techniques can be applied to other operating system organizations, such as large single address space systems [6] and systems with software segmentation [24]. Furthermore, with the current emphasis on application-controlled resource management [2, 10], 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.

Section 2 presents some background material on page table organizations and VM management. Section 3 describes our prefetching scheme, implementation details, and results. In Section 4, we present our software caching scheme, implementation details, and results. In Section 5, results for the integration of our two schemes are presented. In Section 6, we discuss the benefits of our schemes in the context of client-server database applications. We also present a model for TLB penalties on faster architectures. In Section 7 we discuss related work, and we summarize our conclusions in Section 8.

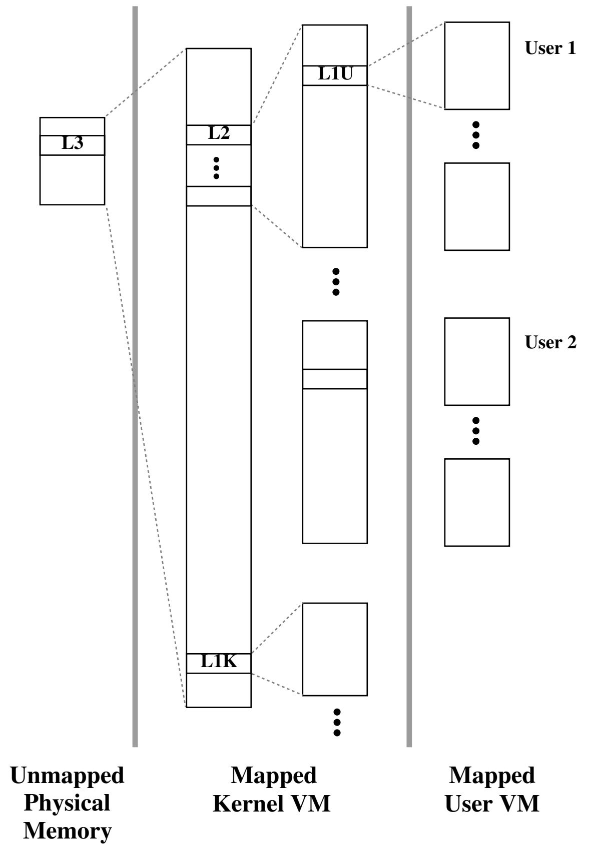

Figure 1: **Mach 3.0 Page Table Hierarchy.** Mach 3.0 implements a 3-level page table hierarchy on the DECstation 5000/240.

2 Background

Architectures such as the DEC Alpha, the HP PA-RISC, and the MIPS R2000/3000 provide software-managed TLBs. This provides greater flexibility to the operating system and eliminates hardware complexity. For example, different operating systems can implement different page table organizations for the same architecture. Some common page table organizations are inverted page tables and forward-mapped page tables. For example, HP-UX implements an inverted page table for the PA-RISC [12], while Mach 3.0 implements a 3-level page table hierarchy for the

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

Table 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.

MIPS R2000/3000 [23].

Figure 1 depicts the page table hierarchy implemented by Mach 3.0 for an R3000-based machine. There are four types of page table entries (PTEs): L1U, L2, L1K, and L3. L1U PTEs map the pages of user address spaces. L2 PTEs map the user page tables. L1K PTEs map kernel data structures. L3 PTEs map the pages containing L2 and L1K PTEs. A page pinned down in physical memory at the root of the hierarchy holds L3 PTEs [17].

There are five types of TLB misses: L1U, L1K, L2, and L3 misses, as well as TLB-invalid misses. A TLB-invalid miss takes place on user addresses that are marked invalid by the virtual memory system. These misses are handled identically by the VM system both in Mach and in our modified versions of Mach.

Table 1 gives the the average cost of each miss type, in CPU cycles, for a 40MHz R3000-based architecture running Mach 3.0. The cycle counts reported in the table were measured using the IOASIC counter on the 25 MHz system bus. The 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 exhibited fairly high variability, on the order of a couple hundred cycles. We also observed that the trap time was related to the time between consecutive traps. We believe that trap times are significantly affected by cache misses.

Since L1U misses are the most frequent, the architecture provides a special trap handler, making them fast. The L1U special 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 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 [17]. 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.

3 Prefetching TLB Entries

One software approach to decrease TLB overheads is to prefetch page table entries. Successful prefetching reduces the number of TLB misses and thus eliminates the overhead of invoking TLB miss handlers. As a beneficial side-effect, prefetching decreases the pollution of caches caused by executing trap handlers. When prefetching TLB entries, three issues must be resolved: where to place the prefetching code, what entries to prefetch, and how many entries to prefetch at a time.

Prefetching can be done on IPCs, or context switches between different protection domains or even between different threads running in a single large address space. Communication between concurrently executing protection domains significantly increases the number of TLB misses in the system, since several protection domains must be mapped at the same time. In this situation, prefetching on the communication path can be useful. Communication takes place very frequently between clients and servers in Mach; therefore, we decided to prefetch entries on the IPC path.

A process is typically structured such that its stack segment and code segment are allocated at opposite ends of the address space, while the data segment is allocated somewhere in the middle of the process virtual address space. This means that more than one L2 TLB entry is required to map the process. Therefore, prefetching the L2 entries mapping the process stack, code, and data segments on IPCs can be beneficial. For communicating processes, prefetching TLB entries that map message buffers and those that map IPC data structures is useful. Therefore, we decided to prefetch L1K entries mapping IPC data structures. We also prefetch L2 entries mapping message buffers and their associated L3 entries, and L2 entries mapping the process stack and code segments.

There is a trade-off between the benefits of aggressive prefetching and the overheads added to the system. Prefetching entries that are not subsequently used can evict useful entries from the TLB and thus increase the total number of TLB misses. Therefore, aggressive prefetching can result in performance degradation. In our experiments, the remaining TLB misses were not frequent enough to warrant the overhead added by prefetching. Therefore, more aggressive prefetching was not worthwhile.

3.1 Implementation

Our implementation maintains TLB entries to be prefetched in a separate *Prefetch TLB* (PTLB). The PTLB is a table maintained in unmapped, cached physical memory, to avoid a cascade of TLB misses while prefetching. PTLB entries are kept consistent with the kernel page tables efficiently, by invalidating them when the kernel invalidates page table entries. The PTLB can hold up to 4K entries and stores L1K, L2, and L3 TLB entries. TLB entries are

	Kernel TLB misses (thousands)			
Application	**Total**	**L1K**	**L2**	**L3**
mpeg_play	245.5	46.8	116.0	82.7
jpeg_play	25.6	6.2	10.0	9.4
video_play	211.0	41.5	98.4	71.1
IOzone	5.9	2.9	0.1	2.9
ousterhout	43.0	11.2	16.3	15.5
mab	89.6	33.3	22.4	33.9

Table 2: **Baseline application statistics.** This figure gives the breakdown of the different types of kernel TLB misses under unmodified Mach 3.0. The counts are in thousands.

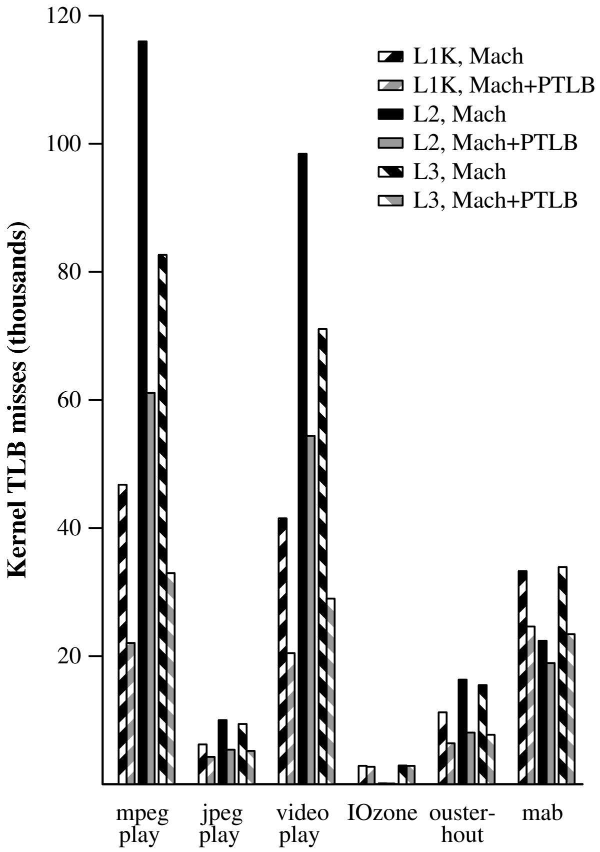

Figure 2: **Kernel TLB misses: Mach, Mach+PTLB.** Kernel misses under unmodified Mach are shown in black; misses under Mach+PTLB are shown in gray.

prefetched on the IPC path between different protection domains. On the first IPC call between a pair of protection domains, TLB misses are handled through the generic trap handler. The hardware TLB is then probed on the IPC path and the TLB entries are entered into the PTLB. On subsequent IPC calls, a hashed lookup is done into the PTLB to prefetch the entries. Thus, the PTLB does not add overhead to the TLB trap handler, but only to the IPC path; however, this overhead is not significant (adding only about 25-60 cycles to a path of length 3500-4000 cycles).

Trace data collected during experiments indicated that prefetching L1K entries mapping kernel data structures eliminates about 20% of kernel TLB misses. Prefetching L2 entries mapping the code and stack segments of processes eliminates about 15%, while prefetching L2 entries mapping server message buffers and their associated L3 entries eliminates another 15%. We decided to dynamically prefetch these entries on the IPC path between processes.

TLB misses on L2 entries for the data segment of processes were also frequent. Mach supports sparse memory allocation, and therefore permits allocation of memory in non-contiguous locations of the virtual address space of a process [5]. While this is a useful feature of the VM system, it is not easy to dynamically determine the location of the data segment of a process. Therefore, we did not prefetch L2 entries mapping the data segment of a process.

3.2 Experimental Environment

Our experimental platform consists of Mach 3.0 running on an R3000-based DECstation 5000/240. MK82 is the Mach 3.0 microkernel used in our experiments with the user-level server, UX41, to provide Unix functionality. Xcfbpmax is the X11R5 server used in our experiments. 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`, `video_play`, which are X applications, and `ousterhout`, `IOzone`, and `mab` (modified Andrew benchmark), which are file-system-oriented applications. Table 2 lists some relevant statistics for each benchmark. For a full description of the benchmarks refer to [16].

Network traffic, Mach's random page replacement policy [7], and the "aging" of the kernel [22], result in variability in application runtimes. The machine was taken off the network to eliminate variability due to network traffic. As the system stays up for a long time, the number of L1K misses increases. This happens because the kernel starts allocating kernel data structures from mapped virtual memory. To eliminate this effect, all timings were taken with freshly booted kernels. However, our schemes will give increasing benefits as the number of L1K misses in the system increases with the aging of the kernel. In order to obtain accurate timings, 50 data points were collected for each benchmark. This decreased the variability in the average time to acceptable amounts.

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

Table 3: **Kernel TLB miss counts: Mach, Mach+PTLB.** Total kernel TLB miss counts, in thousands, and the percentage of kernel TLB misses eliminated by prefetching.

	Kernel TLB Penalty (million cycles)		
Application	Mach	Mach+ PTLB	Application Speedup
mpeg_play	124.6	48.2	1.69%
jpeg_play	13.0	6.0	0.91%
video_play	108.0	41.6	3.52%
IOzone	3.4	2.7	0.19%
ousterhout	28.6	17.4	0.77%
mab	52.0	29.8	0.96%

Table 4: **Kernel TLB penalties: Mach, Mach+PTLB.** The TLB penalties are reported in millions of CPU cycles. The last column gives the percentage of application time saved by prefetching.

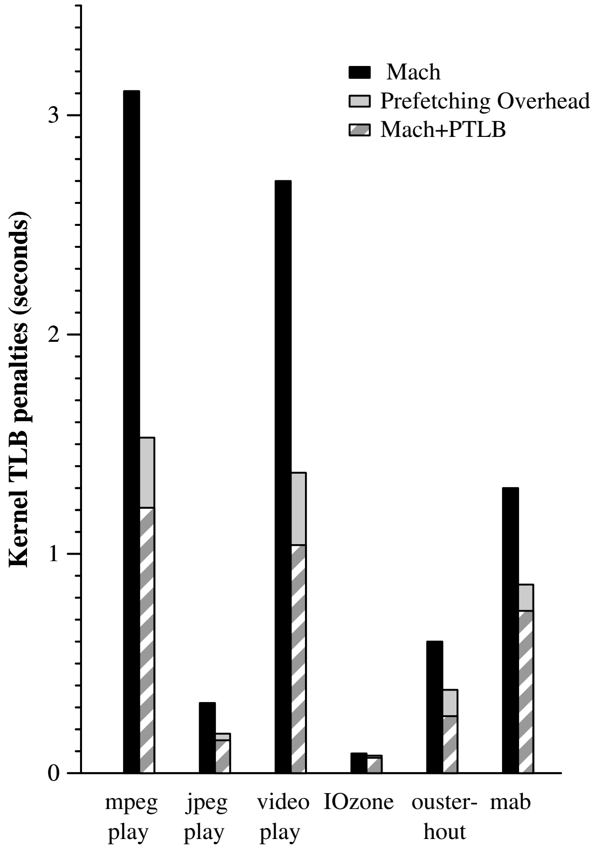

Figure 3: **Prefetching costs and benefits.** The measured timing contributions of kernel TLB misses in unmodified Mach and Mach+PTLB. The solid gray bars show the measured overhead added on the IPC path due to prefetching.

We measured average application runtimes ($\bar{X}$) and standard deviations (σ), for all our experiments. For all the data presented in this paper, we found that for `jpeg_play` $\sigma/\bar{X}$ is less than 0.3%, for `video_play` it is less than 0.8%, for `mpeg_play` $\sigma/\bar{X}$ is less than 1.0%, for `mab` it is less than 1.2%, for `IOzone` it is less than 1.5%, and for `ousterhout` it is less than 4.0%. Since TLB penalties are a relatively small percentage of overall application runtime, it is important that $\sigma/\bar{X}$ be small.

3.3 Prefetching Results

Figure 2 shows the decrease in the number of kernel TLB misses for each of the three types of misses – L1K, L2, and L3. Table 3 gives the counts of kernel TLB misses under unmodified Mach and Mach+PTLB and the percentage of kernel TLB misses eliminated by prefetching.

Prefetching TLB entries on the IPC path eliminates 40% to 50% of the kernel TLB misses for all benchmarks except `IOzone` and `mab`. Neither of these benchmarks is IPC-intensive, and therefore they derive little benefit from the PTLB.

Figure 3 shows kernel TLB miss penalties under unmodified Mach in black. The striped bars represent kernel TLB miss penalties under Mach+PTLB, and the solid gray bars indicate 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).

Table 4 presents the measured kernel TLB penalties, in millions of CPU cycles, for applications under Mach and Mach+PTLB. The last column presents the application speedup due to prefetching; speedup is measured as the decrease in application runtime as a percentage of the application runtime under unmodified Mach. Prefetching results in speedups of up to 3.5% of overall application run time. Kernel TLB misses typically constitute greater than 50% of the overall TLB penalties in an application[1]. Kernel TLB miss penalties constitute 1% to 5% of the overall application time for the benchmarks studied [17]. Thus, prefetching decreases a significant fraction of the kernel TLB penalties.

[1] 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 significantly contribute to overall TLB penalties.

	Mach	DM 1K		DM 4K	
Application	**Total**	**Total**	**Hit rate**	**Total**	**Hit rate**
mpeg_play	245.4	197.5	82.5%	173.8	97.2%
jpeg_play	25.6	20.2	84.4%	17.4	99.8%
video_play	211.0	202.4	70.9%	163.3	100.0%
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 6: **TLB miss counts and hit rates under Mach and STLB.** Total kernel misses in thousands under unmodified Mach 3.0, and the two direct-mapped STLB configurations, DM 1K and DM 4K. Hit rates are included for the STLB configurations. DM 1K is the direct-mapped STLB with 1K entries, while DM 4K is the direct-mapped STLB with 4K entries.

	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 5: **Average STLB penalties.** 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.

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

Table 7: **Nested TLB misses.** Nested TLB misses in thousands under unmodified Mach 3.0, and the two direct-mapped STLB configurations, DM 1K and DM 4K. The nested misses are caused because of the page table hierarchy.

In summary, prefetching benefits all applications and decreases TLB overheads significantly, usually eliminating about 50% of the kernel TLB miss penalties. Although TLB penalties are currently only a few percent of overall application runtime, prefetching will provide increasing benefits as TLB penalties increase in the future.

4 Software Cache of TLB Entries

Architectures with software managed TLBs incur large penalties in TLB miss handling. This is due not only to the use of generic trap handlers but also to cascaded TLB misses resulting from hierarchical page table organizations. We address this problem by providing a large second-level software cache of TLB entries (STLB).

L1K, L2, and L3 entries that have been evicted from the hardware TLB are stored in the STLB. The STLB code branches off the generic trap handler path at the beginning and does a quick lookup in the STLB. On an STLB hit, the entry is inserted into the hardware TLB. On an STLB miss, the code branches back to the generic trap handler path.

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 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.

As with any cache, the STLB can be organized in many different ways – direct-mapped, direct-mapped with a victim cache [14], n-way associative, or fully associative [18]. The organization and the size of the STLB affect its hit rate.

4.1 Implementation

Like the PTLB, the STLB resides in unmapped, cached physical memory, and therefore does not occupy page table entries in the hardware TLB. 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[2].

[2]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.

	TLB penalty (million cycles)			Speedup	
Application	**Mach**	**DM 1K**	**DM 4K**	**DM 1K**	**DM 4K**
mpeg_play	124.6	41.7	21.4	0.27%	1.73%
jpeg_play	13.0	4.2	1.9	0.15%	0.21%
video_play	108.0	60.7	15.7	2.18%	3.53%
IOzone	3.4	0.7	0.6	0.85%	0.59%
ousterhout	24.0	7.9	3.9	0.76%	1.53%
mab	52.0	31.6	13.2	0.76%	0.80%

Table 8: **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. The last two columns present the percentage of application time saved by the STLB configurations.

We studied the following organizations of the STLB: direct-mapped with 1K entries (DM 1K), direct-mapped with 4K entries (DM 4K), 2-way set-associative with 1K and 4K entries, and 4-way set-associative with 1K and 4K entries. The direct-mapped organizations index into the STLB and check only one entry to find a match with the faulting address, whereas the set-associative organizations check more than one entry of the STLB in software to find a match.

4.2 STLB results

Preliminary experiments with direct-mapped organizations showed high hit rates, comparable to the set-associative organizations, but with lower access times because sequential checks in software for matches are avoided in a direct-mapped organization. Therefore, we present only the results for the direct-mapped configurations of the STLB. Table 5 presents the average time, in CPU cycles, that the direct-mapped STLB takes to service TLB misses[3]. An 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. Thus, 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.

The experimental environment and benchmarks for the STLB are the same as those for the PTLB which are described in Section 3.2. Table 6 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% while the hit rates achieved by DM 4K are close to 100%. The larger STLB (DM 4K) services more TLB misses than the smaller DM 1K STLB. This configuration also results in an overall decrease in the number of kernel TLB misses. This is because the cascade of TLB misses to higher levels of the page table hierarchy is eliminated

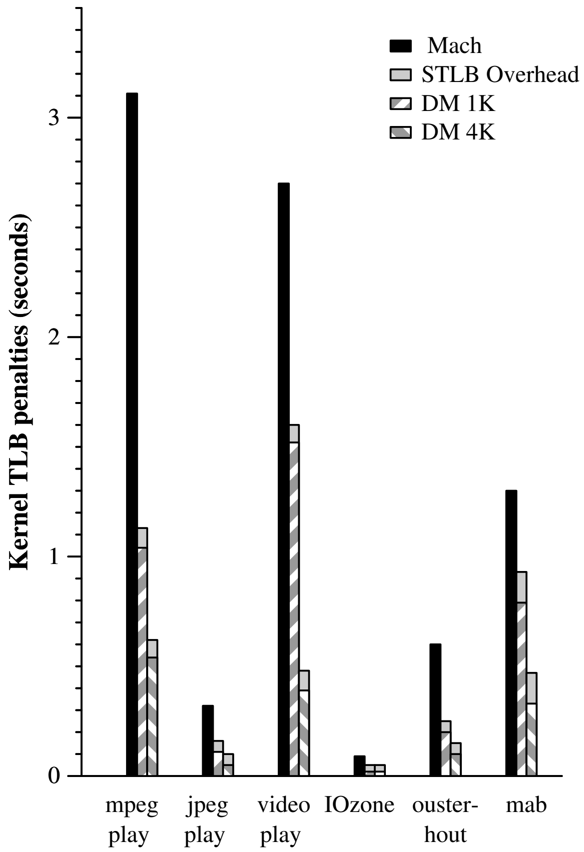

Figure 4: **Total TLB penalties for different STLB configurations.** This figure gives the kernel TLB penalties, in seconds, with the different STLB configurations and compares them with TLB penalties under Mach. The overhead added by the STLB is represented by the solid gray bars. The overhead is the same for both STLB configurations, since it depends on the number of system calls and is independent of the STLB size.

[3]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.

	Kernel TLB Misses			
Application	Mach	PTLB+ STLB	Percent Removed	Hit Rate
mpeg_play	245.4	128.6	47.6%	96.2%
jpeg_play	25.6	13.8	46.2%	99.7%
video_play	211.0	111.8	47.0%	100.0%
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 9: **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 software TLB and prefetching, and the last column gives the hit rate for the STLB.

	Kernel TLB Penalty (million cycles)		
Application	Mach	PTLB+ STLB	Speedup
mpeg_play	124.6	18.4	1.09%
jpeg_play	13.0	1.6	0.27%
video_play	108.0	11.4	3.04%
IOzone	3.4	0.6	0.99%
ousterhout	24.0	3.4	1.65%
mab	52.0	13.6	0.25%

Table 10: **Kernel TLB penalties: Mach, PTLB+STLB.** The kernel TLB penalties are measured in millions of CPU cycles. The last column gives the percentage of application time saved by using the integrated scheme with prefetching and the STLB.

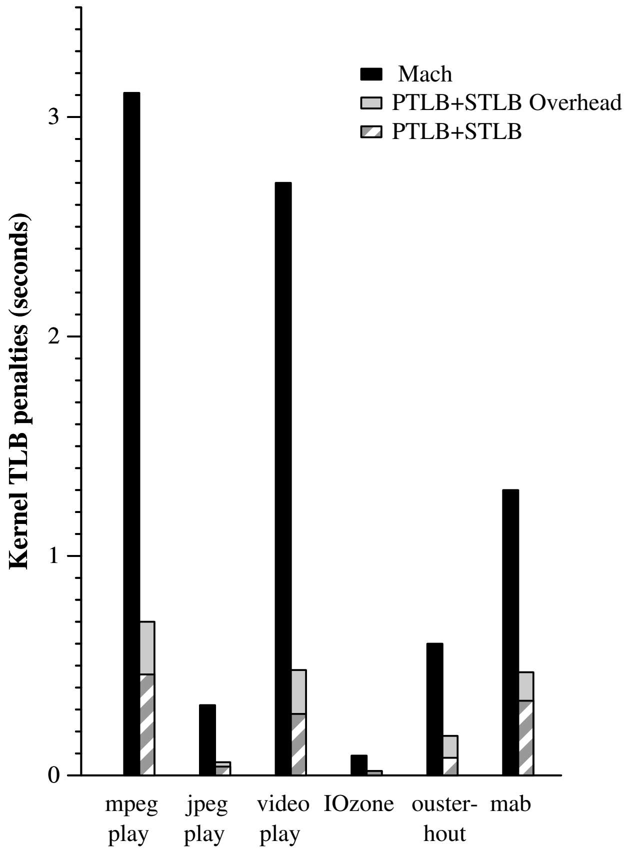

Figure 5: **Costs and benefits of PTLB+STLB.** The measured timing contributions of kernel TLB misses in unmodified Mach and PTLB+STLB. The solid gray bars give the measured overhead added to the IPC path due to prefetching, and the trap handler path due to the STLB.

when a TLB miss hits in the STLB; and as Table 7 shows, under Mach 3.0 nearly 30% of the kernel TLB misses for each application are such nested misses. The larger STLB eliminates almost all of these cascaded misses.

Figure 4 presents the total TLB penalties under the different STLB configurations and unmodified Mach. The gray bars represent the overhead imposed by the STLB on system calls. As is evident from the figure, DM 4K reduces TLB penalties significantly.

Table 8 presents TLB penalties under unmodified Mach and the two STLB configurations, and the percentage of application time saved by using the STLB. The speedups obtained by using the larger STLB, DM 4K, go up to 3.5%, a significant fraction of application kernel TLB penalties. DM 1K does not perform as well for all applications, due to its lower hit rate.

5 Prefetching with the Software TLB

The direct-mapped organization of the PTLB and the STLB make integrating the two schemes fairly straightforward. Using the STLB decreases the number of cascaded TLB traps. Therefore, prefetching L3 entries is not very useful. In fact, it has the negative effect of evicting useful entries from the TLB. Therefore, only L1K and L2 entries are prefetched. Since the DM 4K organization of the STLB performed well, the PTLB+STLB implementation uses a direct-mapped STLB, with 4K entries.

Table 9 gives the counts of kernel TLB misses under unmodified Mach and PTLB+STLB and the percentage of kernel TLB misses eliminated. The STLB hit rates are given in the last column. The hit rates are lower than those obtained for the STLB configuration DM 4K in Table 6. This is because the total number of kernel TLB misses is decreased due to prefetching. Figure 5 shows kernel TLB miss penalties under unmodified Mach in black. The striped bars represent the kernel TLB miss penalties under the PTLB+STLB, and the solid gray bars indicate the overheads added by prefetching and the STLB.

Table 10 presents the measured kernel TLB penalties, in millions of CPU cycles, for applications under Mach and the PTLB+STLB. The last column gives the percentage of overall application time saved due to the integrated scheme. As seen in Figure 5 and Table 10, the integrated scheme performs well, improving application performance by up to 3% of overall application runtime. However, it does not do better than the prefetching scheme alone or the STLB scheme alone for these applications. This is because while prefetching and the STLB are both very effective at reducing TLB penalties, they eliminate opportunities to reduce TLB costs from each other. However, integrating the two schemes adds the overhead of both schemes. Therefore, it does not provide greater benefits than the two schemes do independently for these benchmarks.

6 Discussion

6.1 Multiple Clients and Servers

All the benchmarks presented above involve a single client application communicating though IPC to the UX server and the X server. To study the benefits of our schemes with multiple clients and servers and more frequent IPC, we ran a small synthetic database application that consists of several clients making requests in a loop to a database server. The database server receives a request from each client to query a fixed number of records randomly distributed in the memory-resident database file. The server touches each record requested and returns to the requesting client. This application is IPC-intensive and tests TLB behavior with several communicating processes. Since it includes several clients, it puts significantly more pressure on the TLB than the benchmarks presented earlier.

We ran the benchmark with three clients communicating with one server. Prefetching eliminates about 30% to 35% of the kernel TLB misses. This improves overall application performance by 6%. The STLB exhibits a hit rate of about 90% and eliminates about 20% of the kernel TLB misses by avoiding cascaded TLB misses. Using DM 4K, this improves overall application performance by 10%. The integrated PTLB+STLB scheme achieves an STLB hit rate of about 86%. It eliminates about 40% of kernel TLB misses, and improves application performance by about 12%.

Thus, as the number of clients and servers increases, TLB penalties represent a larger fraction of the runtime of IPC-intensive applications. Our prefetching and caching schemes both provide increasing benefits under these circumstances. This benchmark also demonstrates the usefulness of the integrated scheme. The PTLB+STLB provides greater benefits than either technique used alone.

6.2 Model for Future Architectures

To study the benefits of our schemes on future architectures we modelled trap time as a function of the caching behavior of the TLB miss handler code. We use the following notation: i denotes the number of instructions executed by the trap handler, m denotes the number of cache misses during trap execution, and t denotes the time to service a cache miss. Therefore we have, Trap time $= i + m \times t$.

The time to service a cache miss on the DECstation 5000/240 is t = 24 cycles [8]. On a machine with a clock speed of 200 MHz and the same memory system, the time to service a cache miss will be (200/40) $\times$ 24 = 120 cycles. On such a machine, TLB penalties will become:

$$\begin{aligned} STLB\ fail &= 29 + 2 \times 120 &= 269 \text{ cycles} \\ STLB\ L2,\ path1 &= 61 + 2 \times 120 &= 301 \text{ cycles} \\ STLB\ L2,\ path2 &= 76 + 3 \times 120 &= 436 \text{ cycles} \\ STLB\ L1K,\ L3 &= 52 + 2 \times 120 &= 292 \text{ cycles} \end{aligned}$$

and for unmodified Mach,

$$\begin{aligned} L1K &= 167 + 14 \times 120 &= 1847 \text{ cycles} \\ L2 &= 207 + 14 \times 120 &= 1887 \text{ cycles} \\ KPT &= 161 + 10 \times 120 &= 1361 \text{ cycles} \end{aligned}$$

Thus, TLB miss penalties increase by approximately 1500 cycles, from 500 cycles to nearly 2000 cycles. STLB penalties increase only by about 300 cycles, from 100 cycles to 400 cycles. Referring to Table 5, an L1K miss that hits in the STLB on the current architecture eliminates approximately $(512 - 105) = 407$ cycles. Based on the above model for a 200MHz architecture, an L1K hit in the STLB would eliminate $(1847 - 292) = 1555$ cycles. Prefetching entirely eliminates the penalty of the TLB miss. Thus, prefetching and the STLB should decrease TLB miss handling penalties significantly on faster, newer architectures, and should have an even greater impact on overall application performance.

7 Related Work

Prefetching and caching are two well-known techniques that have been widely applied in computer systems [19, 20], but have not been applied to software management of TLBs. Most of the previous work has focussed on hardware caches.

Huck and Hays [12] 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.

Our work, which has been implemented on a different architecture, complements their work in a number of ways.

Our entirely software-based implementation is in the context of hierarchically-organized forward-mapped page tables and it studies 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. Our 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 approach to decreasing TLB miss handling costs is to provide variable-sized pages that use a single TLB entry to map large contiguous regions of memory. This scheme is used by the PowerPC architecture [13]. Prefetching of TLB entries is still useful for dynamically allocated data structures. As operating systems move to modular organizations supporting finer grained processes interacting with each other, the number of L2 TLB misses will increase. Thus, although each process could be mapped by a single TLB entry with a larger page size, the number of such entries is still large. Since the TLB is a shared resource, contention for the TLB will be high. Therefore, although each entry maps more pages of an address space, prefetching such entries should continue to be useful. Also, in systems supporting garbage collection, ensuring that all important data structures are contiguous in physical memory is not easy. Thus, our schemes can complement the variable-sized pages solution in reducing TLB miss penalties.

8 Conclusions

This paper presents two entirely software-based solutions to decrease TLB penalties. The first scheme, prefetching TLB entries on the IPC path, decreases kernel TLB miss penalties by about 50%. The second scheme uses a large software cache to provide fast path access to entries on a TLB miss. The hit rates achieved are high, ranging from 90% to nearly 100%.

Both prefetching and software caching perform well for all applications. The integration of the two schemes also performs well, though it provides less benefits than any one scheme independently for the benchmarks studied. For a synthetic benchmark that involves several processes communicating frequently, however, using the two techniques together provides greater benefits than either technique alone.

Although TLB penalties are currently only a few percent of overall application runtime, they will probably increase in the future, both because processor speeds are increasing more rapidly than memory speeds, and because of changes in operating system structure that are resulting in more frequent communication. Analysis and experiments indicate that our techniques should improve application runtime by more than 10% in the future.

9 Acknowledgments

Thanks to Wilson Hsieh and Carl Waldspurger for their contribution to early ideas in this work. Special thanks to Carl for his invaluable suggestions, advice, time and support. Thanks also to Don Yeung for our fruitful discussions, Anthony Joseph for his help with X and Rich Uhlig, Alessandro Forin and Mary Thompson for their help with Mach. Thanks to Anthony, Debby Wallach, Don, Eric Brewer, Ulana Legedza, and Wilson for their useful comments on the drafts. Thanks to Paul Leach and the anonymous referees for their helpful comments.

References

[1] T. Anderson, H. Levy, B. Bershad, and E. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the 4th Conference on Architectural Support for Programming Languages and Systems*, pages 108–119, Santa Clara, CA, April 1991.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

[17] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown. Design tradeoffs for software managed TLBs. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 27–38, San Diego, CA, May 1993.

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

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

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

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

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

[23] 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.

[24] 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.