A Case Study of a Hardware-Managed TLB in a Multi-Tasking Environment

Chih-Chieh Lee, Richard A. Uhlig, and Trevor N. Mudge

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan
{leecc, uhlig, tnm}@eecs.umich.edu

Abstract

There have been very few performance studies of hardware-managed translation look-aside buffers (TLBs).The major reason is the lack of efficient and accurate analysis tools. Newer operating systems, applications, and the popularity of the client-server model of computation place a greater burden than their predecessors on memory system components such as TLBs. Thus it is becoming more important to measure the performance of memory systems under such workloads. In this work, we implemented a trap-driven simulator in an operating system to emulate a variety of TLBs. Using this tool, we were able to evaluate the performance of a range of TLBs under these newer workloads. The results show that in order to improve the TLB performance, we should carefully map pages into the TLB, append process identifiers to avoid flushing the TLB contents frequently, or reserve part of the TLB for a particular server process.

1 Introduction

It is widely recognized that the selection of an optimal hardware system design, primarily the design of the Central Process Unit (CPU) and the caches, is highly dependent on the software characteristics that the hardware system is to support. For example, main frame systems are designed to support database applications, while vector machines, or super computers, are designed for massive or intensive computing workloads such as matrix computation. Each hardware system should be designed to support the primary software running on and the hardware/software interface is perhaps one of the most crucial performance factors a hardware designer should confront.

Typical laboratory evaluations of the hardware/software interface have understandably leaned toward being efficient both in time and money, but may no longer be adequate. Hardware designers usually take a single and short program as a test on the system they are developing and use the results of these simple experiments to guide their design of future systems. As a consequence, their machines may perform well only under some artificial circumstances yet perform questionably under a real environment. This is especially true for the computer systems designed for general purposes, such as personal computers and workstations. Meanwhile, these general purpose systems actually have dominated the computer market because they are more widely used than are supercomputers in the distributed network environments, that are typical of business, academic, and research environments. Therefore, the performance of the hardware/software interface in today's widely used computers, personal computers and workstations, strongly demands sufficient, and—if possible—efficient examination.

The largest disparity between the simple but artificial working environment and the complicated but real one is that the latter involves multi-tasking. A multi-tasking environment is a computing environment that allows more than one process[i] to simultaneously share a computer's rare resources, such as the CPU and caches. To allow processes to share the resources, some fair strategies must exist to arbitrate and switch the active[ii] process among the competitors. This function of switching processes can introduce unexpected effects that may not be discovered in the traditionally experimental environment. Meanwhile, multi-tasking is becoming increasingly popular, since the current trends in software design, such as object-oriented programs, microkernel operating systems, and client-server models, can generate process switching more frequently. Numerous preliminary studies have indicated that hardware performance can be degraded greatly in the presence of multi-tasking [Nagle93, Chen94, Nagle94]. It is thus becoming important to reconsider the hardware of modern computers in light of these trends.

Hardware designers may already realize the importance of testing machines by putting them under a more realistic environment. However, the difficulty and expensive cost of the experimental methodology restrains designers from doing it. To overcome this problem, an efficient method of evaluating the system performance under a multi-tasking environment has been proposed by Uhlig, *et al.*, which is termed "*trap-driven simulation*" [Uhlig94a, Uhlig94b]. In this study, we extended this new method to a different but even more popular hardware architecture and collected some interesting results.

In order to emphasize the multi-tasking environment, we incorporated the operating system (OS) because the OS is primarily responsible for managing processes. We instrumented the OS and allowed more than one process to run in the system at one time. While these processes were running, we collected some statistics and also did simulations for hardware components of interest. With this experiment, we can examine various hardware design options and suggest a better one for computer designers.

i. When a program is running, we call it a "process".
ii. "active" refers to the process that is using the CPU at the moment.

In particular, we studied a hardware component that is most likely to be affected by the multi-tasking environment: the translation look-aside buffer (TLB). The purpose of the TLB is to speed up the virtual-to-physical address translation for the virtual memory system by storing the most recently used translation information for a process. Because accessing information from this TLB usually takes one clock cycle, if this buffer can capture the majority of the translation information most of the time, the performance of the system will be greatly enhanced. However, since, as indicated above, the hardware designers configure their TLBs according to single-process experiment, the TLBs may not be able to function well as expected in a multi-tasking environment.

The architecture we are working on in this study is the Intel i486 microprocessor, which has an on-chip 32-entry TLB, organized in 4-way set associative [Intel92]. The i486 microprocessor is unique because its TLB is managed by hardware instead of software, as is the case with previous study [Uhlig 94a]. In addition, the Intel i486 based machine has shared a significant portion of personal computer market, where there are a large amount of software available, including multi-tasking programs, such as Linux and Window 95.

Therefore, this work explores the performance of the i486's TLB in supporting the address translation under a multi-tasking environment we built. To measure performance, we implemented a trap-driven TLB simulator in Mach 3.0, a microkernel operating system, on an i486-based personal computer. Moreover, to magnify the importance of address translation, we did not use simple test programs, such as SPEC92, which are usually used to evaluate system performance. Instead, we developed client-server style workloads that stress the TLB by switching between several tasks frequently. This report discusses the implementation of the TLB simulator and presents the results from these test workloads.

The organization of the remainder of this report is as follows: Section 2 outlines related work. Section 3 describes the design of the trap-driven simulator. Section 4 presents the preliminary results and analyses of our experiments. Section 5 presents some concluding remarks and proposes of our future work.

2 Related Work

Because this work is an evaluation of hardware performance under a multi-tasking environment, previous similar studies focusing on hardware performance under such environments deserve review. In addition, because the experimental method used in this work is critical to the success of this kind of evaluation, previous studies involving similar methods are also reviewed here.

2.1 Similar Multi-tasking Evaluations

Only in the last ten years have researchers begun to study hardware performance under multi-tasking environments or comparable software structures. In these recent studies, two kinds of software structures are commonly investigated: (1) a new-generation operating system, such as a microkernel OS and (2) a new application, such as X window and multi-media applications. In these studies, furthermore, the evaluation of hardware performance is primarily focused on memory system performance.

Studies of a microkernel OS, such as Mach 3.0, have consistently demonstrated that cache and TLB misses occur more frequently with the microkernel OS than with traditional OS structures [Chen93a, Nagle93, Nagle94]. A miss is, simply, an event which caches or TLBs do not contain the data requested by central process unit (CPU), and therefore CPU has to spend more clock cycles to get the data from main memory. The more misses are, the worse the cache and TLB performance is degraded because the purpose of the cache and TLB design is to satisfy most of the CPU's data requests to avoid expensive main memory accesses. As an example of TLB performance studies, Nagle found that this higher frequency of misses is primarily due to the division of the microkernel OS into more subsets-or, more address spaces-than are found in traditional OS structures. As a result, the path of invoking an OS service becomes longer, which, in turn, stresses the cache and TLB more than does the traditional OS structure.

Studies of both X window and multi-media applications showed that these multi-tasking workloads degrade TLB performance considerably [Chen93b, Chen94]. This degradation occurs because these new applications consult servers or OS services more frequently than do the traditional benchmarks. Meanwhile, switching process contents between the applications and the servers prompts a purging of the TLB content, to ensure the address translation valid. This purging will disallow the TLB to be fully utilized.

2.2 Similar Experimental Methods

To evaluate hardware performance under multi-tasking environments, we need tools that are capable of monitoring system activities with minimal disturbance to the system under analysis. The most common monitoring tools are code annotation systems such as *pixie* [Smith91]. These are purely software-based because they work by inserting monitoring code directly into executable images of programs. This process of inserting code is called “annotation.” When the annotated program is executed, the inserted code can record program activities into a predetermined file for post-analysis.

In addition to purely software-based tools, hardware-assisted tools for monitoring system activities also exist. For example, Nagle *et al.* [Nagle92] have developed a monitoring tool by combining a logic analyzer, which is a hardware item, with an instrumented OS kernel, which is a software item. This monitoring tool probes the system bus and records the system statistics in its own buffer. Because it directly probes the system bus, this tool is capable of collecting system activities completely. However, once the tool’s buffer becomes full, the system under investigation needs to be stalled so that the buffer contents can be “dumped” to files. This dumping is necessary, on the one hand, because otherwise the system statistics cannot be collected. On the other hand, the stall is detrimental to the experiment because it discontinues the system execution and therefore distorts the system behavior. Unfortunately, the tool’s buffer is usually small, compared to the amount of system statistics collected during program execution; thus, stalls occur frequently and the system under measurement is distorted by the experiment.

The tools mentioned above, both those purely software-based and those that are hardware-assisted, have some important shortcomings. Although it monitors applications adequately, pixie only works well in monitoring single process activity and cannot capture events produced in an OS, because it is very hard to annotate the operating system. Nagle’s tool requires both a large buffer in the monitoring tool and a method of stalling the system completely and correctly. These are serious shortcomings if multi-tasking environments are to be studied. To study the multi-tasking environment, we must be able to both monitor OS activities and keep the system functioning undisturbed (not stalled) as much as possible. A limited sized buffer and, therefore, the necessity of frequent system stalls inevitably changes the system behavior.

To overcome these shortcomings, Uhlig et al. [Uhlig94b] developed a trap-driven simulator, called *Tapeworm*, that can capture events during operating system activity efficiently and correctly. Furthermore, these events can be processed on-the-fly, thereby avoiding the need for buffering and stalling. Tapeworm, moreover, is purely software-based. It does simulation by setting traps on all memory locations in the workload's address space that correspond to the events under study. Therefore, each time any of those memory regions being trapped is accessed, Tapeworm can be aware of it because a trap occurs. Within each trap, Tapeworm may set or clear traps again on the accessed memory place to control the progress of the experiment. Because Tapeworm is capable of capturing multi-tasking and OS kernel activities, we modified it into the monitoring tool for our work.

This work extends previous work using Tapeworm. In the work of Uhlig et al., Tapeworm was used to study instruction caches and software-managed TLBs in MIPS R3000-based systems. Implementing Tapeworm on i486-based machines, which employ hardware-managed TLBs, represents a new area of study that will also (1) demonstrate the portability of the Tapeworm and (2) allow us to compare the performance of Tapeworms on different underlying hardware platforms. In addition to that, as the most popular general purpose machine, i486-based machines support more intensively interactive workloads than do MIPS R3000-based machines. These workloads are requiring more operating system services because they do more input/output activities, but the performance of these workloads on the i486 machines are still unknown. An third contribution of this study, therefore, is to give initial performance evaluation for these interesting and frequently more popular workloads.

3 Experimental Method

We tested our trap-driven, Tapeworm-based TLB simulator on a Gateway 2000 i486-based personal computer with a Mach 3.0 operating system. Using this trap-driven TLB simulator, we can count the number of TLB misses and hence evaluate the TLB performance and design trade-offs in TLB structures under a multi-tasking environment.

In this section, we describe our Tapeworm-based experimental method in detail. First, we describe the software environment of our experiments, which is composed of several user-level programs and the underlying operating system, Mach 3.0. Because Mach 3.0 is the very program which makes the multi-tasking environment in our experiment sophisticated, we focus on explaining its structure in the section below. In particular, we describe its module and data structure that we used for our experiments, namely PMAP and *pv_list*. Second, we discuss the hardware environment of our experiments. We focus primarily on the memory management unit (MMU) of the i486 microprocessor. In particular, we discuss the i486's two-level page table structure and hardware-managed TLB.

After having described our software and hardware experimental environments, we explain the Tapeworm algorithm in detail. Lastly, we mention some problems we encountered when we were implementing Tapeworm on our i486 machine and propose some solutions.

3.1 Mach 3.0 Microkernel

Mach 3.0 represents a new generation of OS, which is called "microkernel," as opposed to the traditional monolithic OS, such as UNIX BSD. In this section, we first describe the structure of a microkernel OS by using Mach 3.0 as an example. Then we describe those module and data structure in Mach 3.0 which we used for our experiment, the PMAP module and the *pv_list* data structure.

3.1.1 Monolithic vs. Microkernel

In the traditional operating system design, all the OS related codes are implemented in a single address space. This way of implementation is rather straight-forward and allows programmers to easily begin writing the codes. However, as the OS is required to provide more and more functions and services, the OS code grows huger and may contain several times the amount of code it initially may have had. This growth makes the OS hard to maintain. Thus, the need for a new implementation of OS codes has become paramount to programmers.

To solve this maintenance problem, OS programmers have tried to make the OS system more structural. One of these attempts is the microkernel operating system. Microkernel not only provides the benefit of lower-cost maintenance of huge operating systems, but it also provides a more powerful protection mechanism and is more suitable for distributed computing environments. However, the related issues of constructing OS in a microkernel is beyond the concern of this work, so we would like to refer readers to some relevant reports rather than to give detailed discussion here. [Rashid 89]

A typical example of the microkernel is Mach 3.0 [Rashid 89]. Mach 3.0 exports and implements a small number of “essential” abstractions that include inter-process communication (IPC), threads, and virtual memory. Mach 3.0 moves higher-level operating system services, like the UNIX server and the MDOS server, to separated address spaces, usually user address spaces. Under this OS structure, a user program running under Mach 3.0 may contact the UNIX server, which is in another user address space, through the Mach kernel's IPC facility.

3.1.2 The module for handling physical mapping—PMAP

To allow for wide use with as many different computer architectures as possible, Mach 3.0’s virtual memory system is partitioned into machine-independent and machine-dependent layers. Most of the virtual memory modules are implemented in the machine-independent layer so that they do not need to be modified while being implemented on different machines. This easy portability makes the OS more marketable, as it can easily be adapted to various hardware architectures.

In contrast, the dependent layer is machine-specific. The dependent layer of the virtual memory system is contained in the PMAP module, which handles all virtual-to-physical address translations [Rashid88]. The PMAP module creates a unique *pmap* data structure for each task[i]. Every *pmap* data structure also has its own unique handle number, which—if it can be detected—can be used to determine the active task address space. The *pmap* data structure carries the virtual-to-physical address translation information for the corresponding task. To manipulate the address translation information, the PMAP module provides the interfaces *pmap_create*, *pmap_destroy,* and *pmap_enter. Pmap_enter* is the only interface through which the page table entries (PTE) can be changed, and all PTE modifications can be intercepted at this interface. In our work, Tapeworm is a separate code module hooked on to *pmap_enter.*

3.1.3 *pv-list* data structure

The PMAP module also provides another useful data structure for Tapeworm, *pv_list*, which records the inverse address translations, namely, the mappings of physical-to-virtual pages. By going through *pv_list*, Tapeworm can easily find all valid PTEs and then set traps on these physical pages.

3.2 Intel i486 Memory Management Unit

Intel’s i486 microprocessor is already a very popular microprocessor used in numerous personal computer models. The i486’s design goal is to easily accommodate single-user, multi-processing computing environments at compatible performance levels. For this goal, it has a built-in memory management unit (MMU) to effectively support the virtual memory system of the OS. In this section, we explain the

i. In Mach 3.0, “task” is synonymous with “process.”

mechanism the i486 provides for virtual memory system, which is called “two-level page table.” Also, we describe an additional microprocessor component the i486’s MMU uses to speed up this mechanism, which is the on-chip, translation-look-aside buffer (TLB).

3.2.1 Two-level page table

The i486 provides a paging mechanism to support virtual memory multi-tasking operating systems. The i486 uses a page table to translate virtual addresses to physical addresses. This page table is hierarchically composed of three components: the *page directory,* the *page tables*, and the *page frames*. Every virtual address requires two-level translation to get its corresponding physical address, i.e. from the *page directory* to the *page subdirectories* and from the *page tables* to the *page frames*. Therefore, this page table is considered two-level page table. (Figure 3.1) All memory-resident elements of this page table are the same size, 4k bytes.

3.2.2 Hardware-managed TLB

To speed up the virtual-to-physical address translation, the i486’s MMU employs a hardware-managed TLB to cache the address translation for the most recently accessed pages. This TLB has 32 entries that are organized as 8 sets with 4-way set associativity. The i486’s TLB does not distinguish code and data address translations; it does not reserve special room for kernel address space; and it does not use process identifiers (PID) to distinguish address spaces. If a TLB miss occurs, the central process unit (CPU) will search for the corresponding mapping entry in the page table and place it directly in the TLB without informing the OS.

The OS, however, may modify a page table associated with an active process as result of either swapping pages, writing pages, or changing page protections. If the OS modifies a page table, it has to flush the TLB to maintain consistency between the TLB and the page table. To flush the TLB, the OS needs to reload a process’ page directory base address into a control register in the CPU (Fig 3.1) to invalidate all TLB entries. Because the OS may modify the page table frequently, this flushing TLB can occur many times during a program execution in the i486.

3.3 The Trap-Driven TLB Simulator—Tapeworm

Since the i486’s TLB is managed by hardware, all TLB misses are viewed as hardware events that are transparent to software. In spite of this, some operations, such as a page fault, can help expose TLB misses to the OS. The basic idea with Tapeworm is to force a page fault happen on each TLB miss so that all TLB misses are visible to the OS kernel. To do this, all pages are first marked invalid except those pages whose PTEs are held in a specific kernel-controlled data structure. We use this specific data structure to simulate the TLB, and can vary the size, set associativity, etc., of this emulated TLB structure for different study purposes.

The i486’s page table entry (PTE) is a one-word item (32 bits) and is organized as shown on Fig. 3.2. The lowest bit of the PTE is the Present (P) bit, which indicates whether this PTE is valid or not. When a TLB miss occurs, hardware automatically searches the page table for the PTE corresponding to this missing virtual address. To recall, a miss is an event which the TLB does not hold the data requested by CPU. If the P bit of this PTE is 1, this PTE is valid, the hardware simply stores this PTE into the TLB and resumes normal execution. All of these actions are invisible to the OS. If the P bit is 0, the PTE is not valid, which indicates the page pointed by this PTE is not residing in the physical memory and a page fault exception needs to be generated. In a page fault exception, the OS is responsible for bringing in the faulting page from

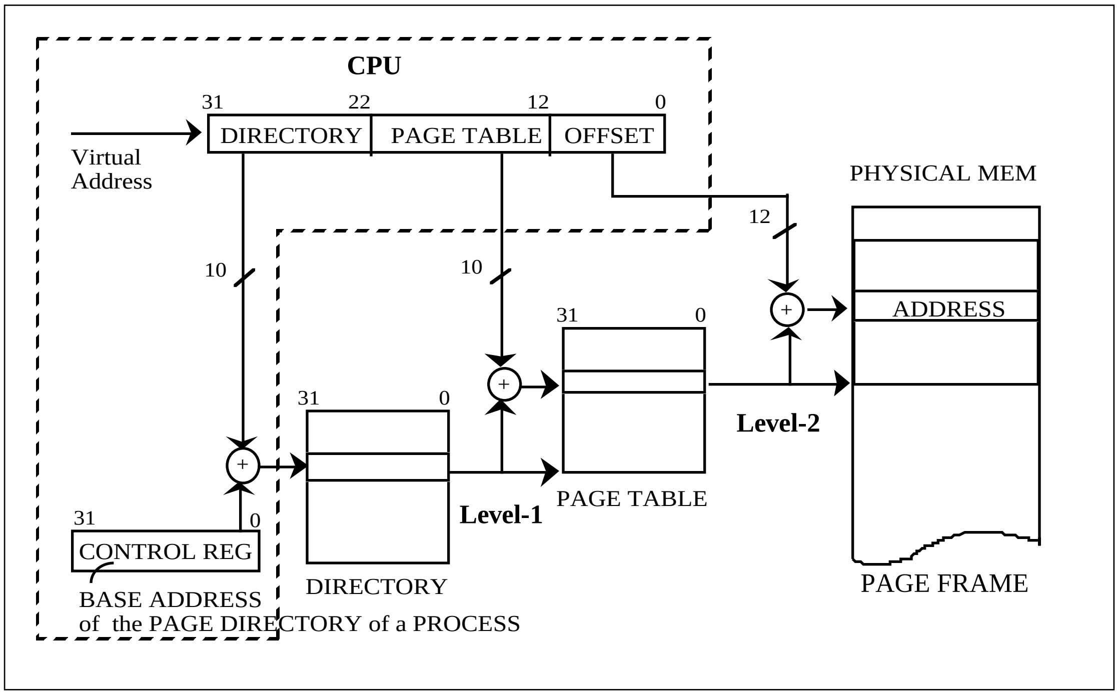


Fig. 3.1 The i486 two-level paging mechanism

31 … 12	11 10 9	8	7	6	5	4	3	2	1	0
PAGE FRAME ADDRESS 31..12	OS RESERVED	0	0	D	A	PCD	PWT	U --- S	R --- W	P

P: Present
R/W: Read/Write Protection
U/S: User/System Protection

PWT: Page Write Through
PCD: Page Cache Disable
A: Access **D**: Dirty

NOTE: The Bit 9 is used as the TapeWorm (**TW**) Bit

Fig. 3.2 The i486's page table entry (PTE) format

the "backing storage," which is usually a disk, and setting the P bit of the corresponding PTE to 1. OS may also be required to evict a page to make room for the incoming page. In a word, the P bit serves as a coordinating point for the hardware and the operating system in the management of the virtual memory system. (Fig. 3.3)

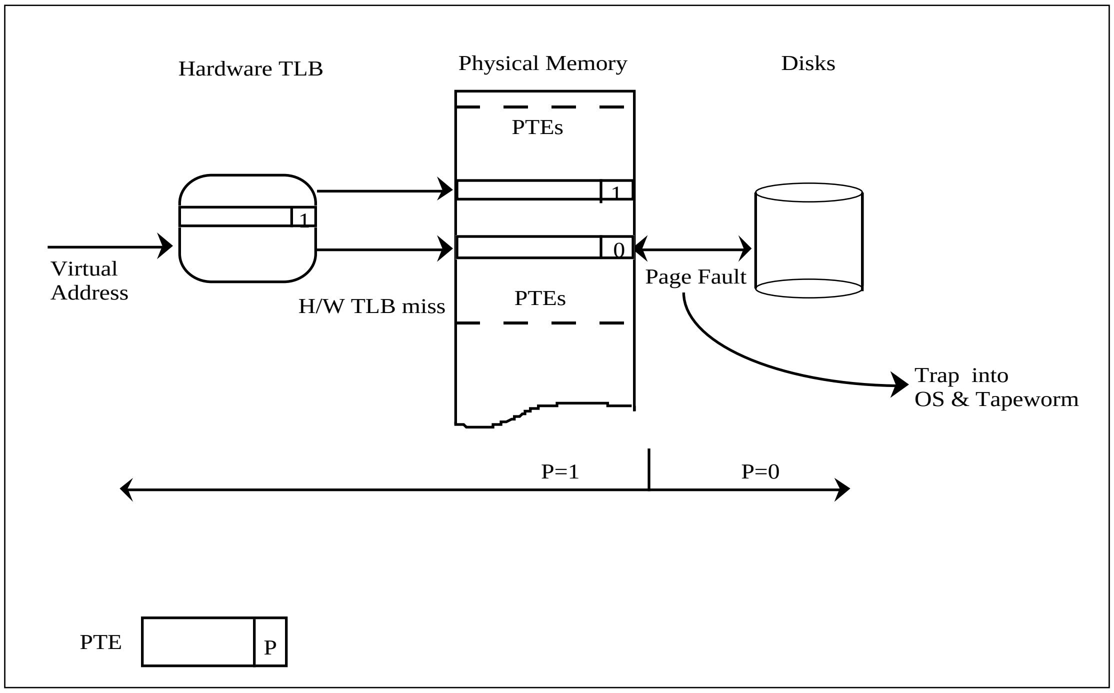


Fig. 3.3 The i486’s support for virtual memory

To implement Tapeworm, however, we need two bits in the PTE for coordination because three entities work together now, which are the hardware, OS, and Tapeworm. P bit is one of these two coordinating bits; the other one can be obtained from one of the three OS reserved bits as shown in Figure. 3.2. We will refer to it as the TW bit in this work.

The P bit is now used to indicate if the PTE is valid in the emulated TLB, while the TW bit is used to indicate if the PTE is valid in OS. Table 1 is a summary of all the possible statuses of a PTE and its page. In

TW Bit	P Bit	Status	Description
0	0	Invalid	PTE is invalid both in Tapeworm and in the OS. Its page is in the backing storage, and not present in physical memory
0	1	Forbidden	This PTE is not allowed because its page is in the emulated TLB but not present in physical memory, which is not realistic.
1	0	OS_valid	PTE is invalid in the Tapeworm, but it is valid in the OS. Its page is in physical memory, but not present in the emulated TLB
1	1	TW_valid	PTE is valid both in Tapeworm and in the OS. Its page is both in the emulated TLB and in physical memory

Table 1: Page status in Tapeworm

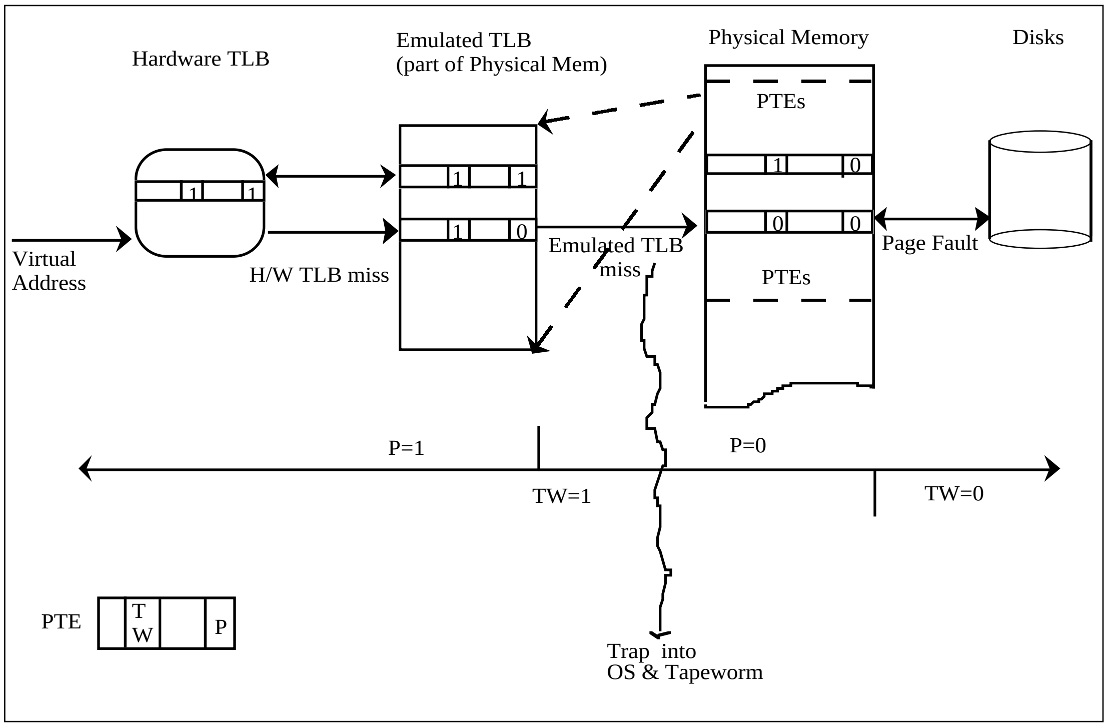


Fig. 3.4 Tapeworm on the i486

such a scheme, when a miss occurs in the hardware TLB, i486 hardware controller will first examine the emulated TLB for the missing PTE. If this PTE is absent in the emulated TLB, its P bit must be 0. It means this page status must be Invalid or OS_valid. Since P is 0, this missing PTE will cause a page fault exception into OS. Tapeworm will be invoked at this moment and check the TW bit of the missing PTE. If its TW bit is 1, it means the page status is OS_valid, and Tapeworm can bring this PTE into the emulated TLB directly from the physical memory and do some replacement if necessary. Otherwise Tapeworm passes this faulting event to the OS kernel for handling because it is a true page fault. (Fig. 3.4)

When implementing Tapeworm, it is necessary to keep track of the number of TW_valid PTEs in the entire system for the emulated TLB. As mentioned in Section 3.1, we can achieve this by monitoring *pmap_enter* activities. If *pmap_enter* validates a new PTE, Tapeworm must put this new PTE in the emulated TLB and possibly replace some other TW_valid PTE, if this is necessary to create space.

3.4 Some Problems with Implementing Tapeworm on i486 Machines

We encountered three problems listed below when we implemented Tapeworm on an i486 machine. We solved some of these problems, and have some suggestions for solving those remaining.

- Simulating the replacement policy
- Invalidating kernel address pages
- Counting the total instructions executed

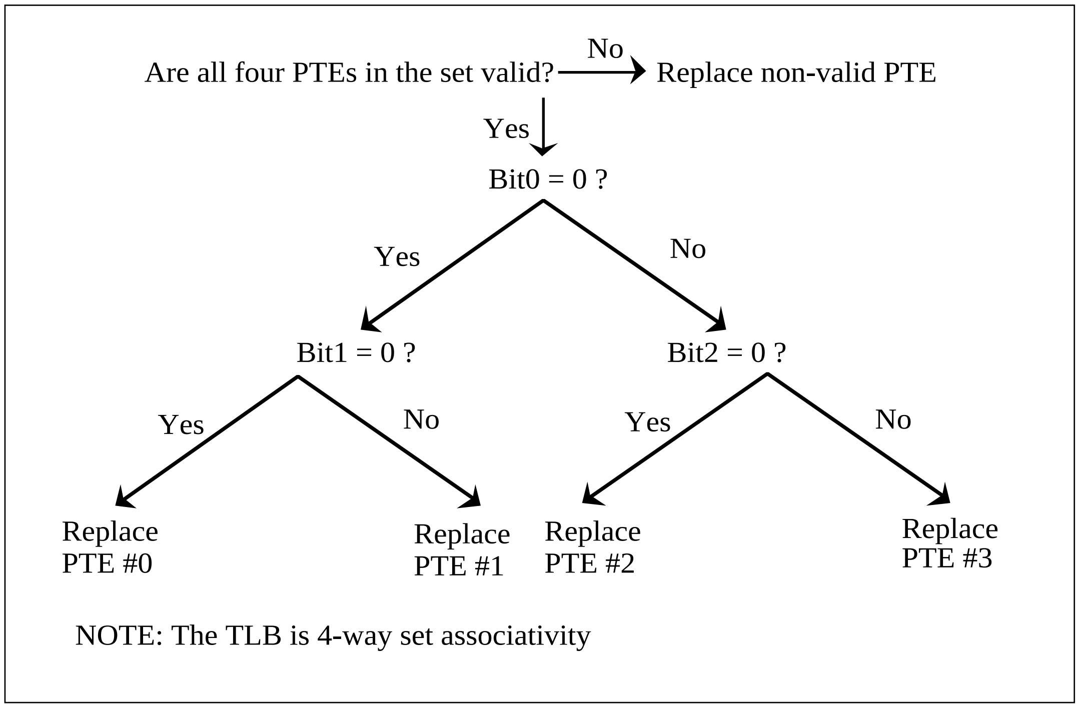


Fig. 3.5 The i486's pseudo LRU replacement policy

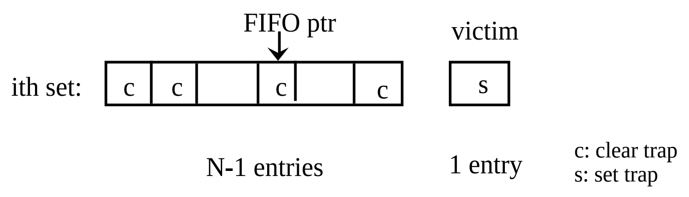
3.4.1 Simulating the replacement policy

By using underlying hardware, Tapeworm filters out hits in the emulated TLB and processes only the miss events. Because of this filtering capability, Tapeworm is much faster than other approaches [Uhlig94b]. However, because Tapeworm records only the miss events, its record of workloads (tasks processed in the TLB) is incomplete. Thus, it cannot simulate those replacement policies which depend on the full history of references, such as the least-recently-used (LRU) algorithm. Fortunately, most modern microprocessors' TLB designs do not depend on the LRU algorithm but on a pseudo-LRU replacement policy, which Tapeworm can simulate. This replacement policy is simpler and much less expensive for implementation because it does not need the full reference history. Furthermore, this pseudo-LRU policy can still perform well because it does not throw away the most recently used entry.

The i486 adopts the pseudo-LRU as the replacement policy for its TLB and on-chip cache. Specifically, for each set of the TLB it uses only three bits to identify the most recently used entry and the entry to be likely replaced out (Fig. 3.5). (in contrast to an LRU policy, which uses six bits).

To simulate this pseudo-LRU policy in Tapeworm, we chose one of the members in each set as the not-most-recently-used (NMRU) candidate for that set. We labeled this selected member "victim." As long as we guaranteed the victim is a NMRU, we could replace it whenever an entry slot of the emulated TLB needs to be reclaimed. To ensure that the victim was NMRU, we set a trap on the victim. When the victim was referenced, we caught this reference by a fault exception, and then cleared its trap and selected another entry from the same set as the new victim. For example, suppose the emulated TLB is N-way set associativity, one entry of each set is set a trap on while others are not. If a fault occurs on the victim, we exchange any one of

The Emulated TLB has
M sets with
N set associativity



```
Supposed the missing PTE (mi) is mapped into the ith set.

If  FIFO_ptr->PTE is invalid
        FIFO_ptr->PTE = mi

else if  mi = victim
        swap FIFO_ptr->PTE with victim

else
        victim = FIFO_ptr->PTE
        FIFO_ptr = mi
        miss ++
```

Fig. 3.6 Tapeworm's strategy for NMRU replacement policy

the N-1 entries of the same set with that victim. Through this way, we can guarantee that next time when a new PTE is claiming a slot, the victim entry is absolutely not the NMRU (Fig. 3.6).

Another way to emulate the pseudo-LRU replacement policy is as follows. Every valid PTE of the i486 has an Access (A) bit. Whenever the CPU accesses a PTE, it will set the A bit of that PTE. Tapeworm could periodically clear the A bits of all PTEs in the emulated TLB by using the clock interrupts. By determining an appropriate length for the interrupt period, Tapeworm can keep sufficient history of references. Then it can tell which entries of the emulated TLB were not referred during the last period, and, hence, can determine which entry should be replaced out. This method, compared to the one mentioned just above, has fewer fault exceptions because all of the emulated TLB entries are not set traps on. However, it may introduce another overhead, periodically clearing the A bits of all the emulated TLB entries. Moreover, this method may not be applicable for the Tapeworm-based cache simulator if the emulated cache data structure does not provide such an Access bit.

3.4.2 Invalidating the kernel pages

When Tapeworm was implemented, furthermore, some faulting memory addresses were not restartable. Hence some kernel pages cannot be set traps on. At each time when Tapeworm begins, all the page frames in the physical memory should be invalidated first in order that the emulated TLB can work correctly. All page frames in user address spaces can be invalidated without side-effects. However, some portions of the kernel address space cannot be invalidated. This occurred in part because the i486 CPU may generate more than one memory reference in an instruction. In some cases, the second or later memory references are not restartable if a page fault exception occurs in previous memory references. These memory references are mainly touching the hardware data structures residing in the kernel space such as the Task Control Blocks.

So some portions of the kernel space cannot be faulted on. This problem was not discovered when Tapeworm was implemented on MIPS machines because these machines are RISC style, which issues only one memory reference in each instruction.

3.4.3 Counting the total instructions executed

Tapeworm only intercepts the miss event, therefore it neither counts the total number of instructions executed nor obtains the ratio of misses to total instructions, the “miss ratio.” For fair comparison, we only compared the results from the same workload currently. To make an exactly quantitive comparison of TLB performance across benchmarks or machines, we would need to collect the following statistics: the number of total instructions of each workload, the cost of servicing a TLB miss, and the cost of servicing a page fault. Given the number of total instructions, we could calculate the miss ratio and hence we can compare results across benchmarks. After obtaining the cost of servicing a TLB miss, we could measure the portion of the memory stall cycles due to the TLB performance degradation and compare the i486 TLB performance with TLBs on other machines for these test workloads.

The important parameters mentioned above can be obtained through hardware devices, such as Monster developed by Nagle [Nagle92] and timers with high resolution. Some microprocessors even provide on-chip instruction counters to make it easier to count total instructions of a workload. Unfortunately, the i486 does not provide such a counter, so an additional hardware device is necessary. Our present effort is to install Monster on the i486 machine and collect the above mentioned statistics.

4 Experiments and Analyses

In this section, we present our preliminary experimental results to demonstrate the problematic performance of hardware components under a multitasking environment. As mentioned previously, we focus primarily on the i486’s TLB performance. To achieve our experiment’s goal, we built two multitasking environments as tests. One is a DOS server running a LOTUS spread sheet and the other is *mpeg_play*; both are under Mach 3.0 operating system.

4.1 Mach DOS Server (MDOS) and LOTUS

The first multitasking environment we built is the DOS server provided by Mach 3.0 with an application program, the LOTUS spread sheet. The DOS server of Mach 3.0, known as MDOS, is to emulate the Microsoft DOS working environment under Mach 3.0. It is a good example for demonstrating the microkernel OS’ capability of supporting more than one operating system service semantics at a time. Particularly, this DOS server can co-exist with the UNIX server under Mach 3.0 without causing conflicts. We chose LOTUS spread sheets as the test application program because it is widely used in general purpose computers.

To present our experimental results, we first introduce the working model of the MDOS server. Then we show the performance degradation of the i486’s TLB under this environment and discuss the various reasons of degradation. Moreover, the speed of our Tapeworm-based TLB simulation also deserves mention because it is the major advantage of trap-driven simulation over other kinds of simulation tools.

4.1.1 The working model of the MDOS server of Mach 3.0

MDOS server is a user-level server task on top of the Mach 3.0 kernel. It supports DOS system services as a multi-threaded[i] Mach task running in conjunction with the 4.3 BSD UNIX server. The MDOS server provides a truly emulated traditional DOS environment so that those existing DOS applications can run immediately under Mach 3.0 without modification.

To run a DOS application program, MDOS first loads the DOS bootstrap image from a DOS partition of a second storage (disk), initializes the instruction pointer of a special thread (V86 thread) to the bootstrap code, and then resumes the V86 thread’s execution. The i486’s instruction set architecture (ISA) provides a feature which can interpret instructions as an 8086 chip would. This feature is called the Virtual 8086 mode. In this mode the one megabyte address space of a DOS task can be mapped to anywhere in the 4 gigabyte address space provided by the i486, and hence DOS can co-exist easily with other UNIX tasks. The V86 thread utilizes the Virtual 86 mode, maps DOS into the MDOS task's address space, and then begins interpreting DOS commands. This V86 thread is also on behalf of DOS application programs when they are running. In other words, it will read instructions from DOS application programs literally and execute them. This V86 may generate exceptions such as DOS system calls and system interrupts, when it is executing on behalf of DOS programs. The Mach 3.0 exception handling mechanism can catch these exception events and either handle them inside the kernel or pass them back to the MDOS server.

The MDOS task of Mach 3.0 consists of six threads. One of them is the V86 thread, whose function has been mentioned above. Other threads include an MDOS thread which handles the exceptions passed back from kernel, a pager thread which handles the page mapping from conventional DOS memory to virtual memory, and three other threads that are responsible for generating DOS interrupts for keyboard, mouse, and timer, respectively. The V86 and the MDOS threads are the sources for most of the execution time when a DOS application program is running.

4.1.2 Experiment and results

In this work, we used as our test program a LOTUS spread sheet which calculates stressing forces by using a finite-element method. We counted the number of TLB misses for each entire “test cycle.” A typical “test cycle” consists of the following steps in order:

- start the MDOS server
- start the LOTUS program
- open the spread sheet
- vary two particular input parameters
- have LOTUS do calculation
- exit the LOTUS program
- exit the MDOS server

For each TLB structure, we ran our test cycle twice, and we averaged the two data sets because the results may vary from run to run for two major reasons. One is that Tapeworm is intrinsically a dynamic system in which experimental results are non-deterministic; the other is that the test cycle needs some manual inputs.

i. Mach supports a multi-thread environment in which a task can be broken into more than one thread; each thread is an independent executing unit within the address space defined by its task.

A dynamic system means that when the test program is running, there are also some other processes, such as the network daemon, running on the same machine. These processes will come to occupy the CPU alternatively and in a random order. Because their behaviors are unpredictable and their execution may affect the experimental outcomes, the results are not deterministic.

Fig. 4.1 shows the number of misses contributed by the MDOS server against twenty TLB structures for our test cycles. We obtained these results by using a “first-in, first-out” (FIFO) replacement policy for the simulated TLB. As shown in this figure, we can easily find that the miss numbers level off as the size of the simulated TLB is above 64 entries. Specifically, this implies that a TLB of 64 entries can cover the “working set”[i] of this workload most of time. In addition, the miss number for the TLB of 32 entries with 4-way set associativity is much higher than that of the others. The i486 TLB has the same structure, and hence it will perform badly for this workload.

To further investigate the reasons of performance degradation, we studied the following perspectives:

- TLB capacity
- Mapping of the TLB entries for demanded pages
- Instruction and data reference separation

i. “working set” means the range of data requested by programs at a time.

4.1.2.1 TLB Capacity

As the TLB size gets larger, the performance disparity between set associativities and full associativity is less significant because large TLBs can cover the active process's working set and most of the remaining misses are due to compulsory misses[i]. To reduce these remaining misses, one possible way is appending a process identifier (PID) to each TLB entry to avoid flushing the TLB when context switches happen. We examine this suggestion further in the next section. Another way is prefetching wisely. Prefetching is a good technique to enhance the performance of the instruction caches in CPUs. However, compared to caches, the TLB entries have larger granuality since each of them covers a range of a page size. In such a case, the advantage of prefetching used in TLBs is questionable because within a page of 4k size the instructions stream issued from the CPU may have great chances to change the directions due to the quota expiration or mandatory context switches for requesting system services. We have not yet proven our thinking currently, and we plan to do it as our next step.

4.1.2.2 Mapping of the TLB entries for demanded pages

We showed above that when the TLB size is 32 entries the number of misses for 4-way set associativity is much higher than those for higher degrees of set associativity of the same TLB size. We thought this is due to the poor mapping of the TLB entries for pages demanded by programs. Therefore, we made a detailed examination on the TLB of 32 entries.

As shown in the Fig. 4.2, we proved our thinking by comparing two TLB structures: 4-way against 8-way set associativity. We changed the mapping function of the TLB entries for pages by simply changing the TLB’s set associativity. This change reduced the TLB misses significantly.

To realize the reasons of the poor mapping, we have the following observations. From the Fig. 4.1, we found most of the extra misses were from the v86 thread’s references. Moreover, in the Fig. 4.3 we show that most of those extra misses were contributed by data references. We conjectured that the LOTUS spread sheet has poor data placement in memory, since most of the extra misses were from the v86 thread which executed instructions on behalf of LOTUS. However, this conjecture needs to be further verified.

To further support our thinking of poor mapping, we also have tried remapping the pages into the TLB by using a TLB of 28 entries. As also shown from the middle two columns in the Fig. 4.3, we got good benefit from remapping. Fig. 4.4 shows the profile of misses over TLB entries where both TLBs have 4-way set associativity. We can see the mapping is better for the TLB of 28 entries, although there is still a set with peak contention.

4.1.2.3 Instruction and data reference separation in the TLB

We also examined the effectiveness of splitting instruction and data references for TLB. In other words, we suspected if the conflicts between instruction and data references were significant.

The leftmost column in the Fig. 4.3 represents the miss number of a TLB of 36 entries in which 4 entries were allocated for instructions and 32 entries with 4-way set associativity were for data. Apparently, we got little benefit from this TLB structure and splitting instruction and data references cannot help reducing misses. The reason is shown in the Fig. 4.5. Most of the instruction pages were not mapped into the sets with higher contention. In the Fig. 4.3 the rightmost column represents the miss number of a TLB of 32 entries in

i. “compulsory miss” is a miss due to the fist time of data request. after a context switch

-

which 4 entries are for instructions and other 28 entries are for data. Again, we cannot gain benefit from this structure, compared to the unified TLB with 28 entries.

4.1.3 The slowdown

One of the interesting issues for implementing Tapeworm on a i486-based machine is to compare Tapeworm performances on different platforms. Fig. 4.6 shows the slowdown for the MDOS & Lotus test program running with Tapeworm. Note that

Compared with the result in Uhlig’s work [Uhlig94b, Fig. 2], the Tapeworm on i486 introduces almost the same degradation to the system performance as the one on MIPS R3000, although the i486 one performs worse while the emulated TLB is small. It performs worse because the codes have not yet been optimized.

However, the code size of the Tapeworm on i486 is smaller, compared to the Tapeworm on MIPS R3000. One of the reasons is that the Tapeworm on MIPS R3000 needs to ensure no overlaps between contents of the real TLB and its own emulated TLB. On the other hand, the Tapeworm on i486 can ignore it because the

–

real TLB is always a subset of the emulated TLB. Another reason is that the Tapeworm on i486 relies on the hardware to search for entries in its emulated TLB. In contrast, the Tapeworm on MIPS R3000 has to do searching by software in the extended part of the emulated TLB if the emulated TLB is larger than the real one.

4.2 *Mpeg_play* - A Benchmark Interacting with Servers Intensively

4.2.1 The working model of the *mpeg_play* in the X window system

mpeg_play is a software video decoder which decodes and displays frames from MPEG video bitstreams. A previous study on the *mpeg_play* has shown that memory bandwidth is the primary limitation in performance of this decoder instead of the computational complexity [Patel92]. In addition, *mpeg_play* needs intensive supports from the operating system and X-window server. In the Mach 3.0 system, there are three user-level tasks being involved in when *mpeg_play* is running; they are the *mpeg_play* task, the UNIX server, and the X-window server. Therefore, *mpeg_play* is a good candidate to stress the TLB in this study and can help us to understand the demands of the multi-tasking environment.

In this work, *mpeg_play* displayed 148 frames in each trial. Again, every value shown here is an average of two trials and each trial was executed in the same environment.

4.2.2 Experiment and results

Fig. 4.7 shows the misses contributed by each involved task against various TLB structures.

The TLB employed the FIFO replacement policy. We can easily find that when the TLB size is above 128 entries the miss numbers level off.

Because each task here represents a distinct user address space, whenever the active task is changed the emulated TLB will be flushed once to avoid the alias problem. The effect of context switches will hence be eminent in this experiment. Although the TLB size needs to be as large as 128 entries to cover the largest working set, the size of 64 entries has been large enough for the UNIX and X-window servers. In addition, as the TLB gets larger, most of the misses contributed by the UNIX server remain and occupy more fraction of total misses. As will be shown later, those misses are due to the coldstarts of the TLB, i.e., compulsory misses after each context switch. In other words, it implies that the UNIX server suffers from the context switches and UNIX server's program locality is not exploited in the TLB. This problem arises because the UNIX server acts as an event-driven model. It becomes an active task only when user tasks ask for a certain UNIX service. Those services may happen frequently but each runs shortly. This situation is similar to the common kernel behavior which is driven by interrupts from users or hardware.

4.2.2.1 Splitting instruction and data references

To reduce the misses for smaller TLBs, we tried to split the TLB according to the instruction and data references. However, the results were not good. One of the reasons is that, similar to the MDOS server, there is little interference between the data and instruction references. Another reason is that for *mpeg_play* there are three tasks working together and therefore an optimized I/D partitioning for one task may not be suited splitting instruction and data references, we increased the misses of the other two tasks and the overall miss numbers were still high.

4.2.2.2 Appending process identifiers (PIDs)

Although splitting instruction and data references for the TLB does not work well, we may reduce the misses for *mpeg_play* by avoiding flushing the TLB. To not flush the TLB, we appended a PID to each TLB entry such that there is no aliasing problem after context switches occurred. Without being flushed in context switches, the TLB can remember useful translation information across address space boundaries. Especially, X-window applications may invoke mandatory context switches frequently because they need immediate system services, as shown in [Chen94]. Flushing the TLB at context switching will thwart the TLB performance greatly for X-window applications. Fig. 4.8 compared the miss numbers for the TLBs with and without PIDs and showed how badly the TLB performance degraded due to the coldstarts.

4.2.2.3 Isolating the UNIX server entries in TLB

Although appending PIDs to TLB entries is beneficial, it may cause large area of the TLB and also complicate logic design, thus impacting the cycle time.

One way to get around is that, we may allocate some slots of the TLB to be dedicated for a particular address space during the program execution. This can be done either by using the software managed TLBs or adding some hardware supports. The software managed TLBs have been available in some modern microprocessors and the trade-off between its performance and cost has been studied in [Nagle93]. Here, we studied for the hardware managed TLB.

By dedicating TLB slots for the UNIX server, we can preserve the translation information for the frequently invoked service subroutines in the TLB without being interfered by other address spaces. In this experiment we partitioned the TLB into two parts; one for the UNIX server only and the other for *mpeg_play* and the X-window server. Since there was no PIDs appended to TLB entries, when context switches occur between *mpeg_play* and the X-window server, the portion of the TLB allocated for these two processes was flushed.

In our experiments, there are three types of fully-associative TLBs examined, as shown in Table 2, and for

TYPE-1 TLB	appending PIDs, without flushing TLBs
TYPE-2 TLB	a separated TLB for UNIX server
TYPE-3 TLB	conventional unified TLBs without PIDs; flushing TLBs in context switches

Table 2: Three types of TLBs

each type of TLBs three sizes are studied. Fig. 4.9~4.11 present our results. In each figure, the left two columns represent TYPE-1 TLBs, the middle four columns for TYPE-2 TLBs, and the right two columns for TYPE-3 TLBs.

As shown in Figures 4.9~4.11, TYPE-2 TLBs are worse than TYPE-1 TLBs. The reason is that with PIDs appended on, the TYPE-1 TLB can dynamically exploit the entire TLB for the memory reference patterns. On the other hand, although we reserved a subset of the TYPE-2 TLB for the UNIX server, we actually restricted the TLB capacity for each task. As TYPE-2 TLBs are small, the extra capacity misses can offset the benefit from dedicating TLB slots for the UNIX server.

As the TLB size is small, such as 64 entries shown in Fig. 4.9, TYPE-2 TLBs perform worse than TYPE-3 TLBs. However, as the TLB becomes larger (256 entries), most of configurations of the TYPE-2 TLB can outperform TYPE-3 TLBs. When the TLB size is small, the TLB capacity is the most concerned problem, and further partitioning TLBs will worse the problem. When the TLB is large enough to cover the working set, partitioning becomes useful. We can look at the figure where the TLB has 256 entries (Fig. 4.11). The misses of UNIX server are almost neglectable as the subset of a TLB allocated has more than 128 entries. This means the TYPE-2 TLB has remembered most of the UNIX service subroutines. Moreover, the misses of the X server also become fewer, as opposed to its competitor, *mpeg_play.* This is interesting because it suggests that performance of the X server is strongly related to that of the UNIX server.

Finally, for those applications which contact with servers intensively, OS scheduling policy may be irrelevant to the TLB performance because the context switches may occur before quota expires [Chen94]. However, if we preserve a subset of the TLB for the server such that TLB can remember the server-related translation information across context switches, then the scheduling policy may need to be considered.

4.2.2.4 The replacement policy

In this work, we also examined different replacement policies for TLBs. We modified our replacement policy to emulate the pseudo-LRU employed in the i486 TLB. The pseudo-LRU policy is actually a variation of Not Most Recently Used (NMRU) policy mentioned in the section 3. Fig. 4.12 and Fig4.13 present the results. We found that there is little difference between these two policies, especially as the TLB gets larger.

5 Concluding Remarks and Future Work

The structure of the software system has become more modularized and multiple-threaded. In this work, we have examined two examples: one is a multi-threaded task, and the other has three tasks involved; both are under the operating system, Mach 3.0, a micro-kernel working environment.

We have demonstrated that to improve the TLB performance for the two examples examined, carefully mapping pages into the TLB, appending PIDs to the TLB entries, and fixing TLB entries for frequently invoked OS service routines can be beneficial. However, splitting the TLB into instruction and data parts has no advantages, which is similar to the experience in designing caches that miss rates are not improved by splitting caches to reduce conflicts between instructions and data [Hennessy90].

By improving the TLB structures we can enhance the performance of translating addresses under a multi-tasking environment. If software managed TLBs are used, there is almost no cost for adding enhancement since only a small amount of code needs to be added into kernel. However, for hardware managed TLBs, it is not so flexible to make these modifications once they are implemented. Moreover, some of improvement methods can impact on the cycle time. Therefore, it deserves more detailed examination for the improvement methods found in this work.

In addition to the study above, we are also interested in the following future work. Some new microprocessors adopt complicated MMU designs. For example, the Power PC 601 has three TLBs. Two of

–

them form a two-level TLBs and the remaining one is used to support variable-sized blocks. The performance of this new MMU design under a multi-tasking environment is still unknown and is worthwhile of study.

We have examined the performance of some newer workloads, but have not yet studied database systems which are also important in general-purpose computer systems. Database systems may show different reference patterns because they have their own data placement policies and distinctive system structures. We are interested in investigating this to find out its needs for hardware support.

References

[Anderson91] Anderson, T.E., Levy, H.M., Bershad, B.N., et al. *The interaction of architecture and operating system design*, In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, ACM, 108-119, 1991

[Chen93] Chen, B., et al. *The impact of operating system structure on memory system performance*, In Proc. 14th Symposium on Operating System Principles, 1993

[Chen94] Chen, B. *Memory Behavior of an X11 Window System,* In Proc. of the USENIX winter Technical Conference, 1994

[Hennessy90] Hennessy, J., Patterson, D., *Computer Architecture: A Quantitative Approach*, p423~424, Morgan Kaufmann Publishers, INC., 1990

[Intel92] Intel, *Intel486 DX Microprocessor Data Book,* 1992

[Malan92] Malan, G., et al. *DOS as a Mach 3.0 Application*

[Nagle92] Nagle, D., Uhlig, R. and Mudge, T. *Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures.* The University of Michigan. CSE-TR-147-92. 1992

[Nagle93] Nagle, D., Uhlig, R., Stanley. T., Sechrest, S., Mudge, T., Brown, R., *Design tradeoffs for software-managed TLBs*, In the 20th Annual International Symposium on Computer Architecture, San Diego, California, IEEE, 27-38, 1993

[Nagle94] Nagle, D., Uhlig, R., Mudge, T., et al. *Optimal Allocation of On-chip Memory for Multiple-API Operating Systems,* In The 21st International Symposium on Computer Architecture, Chicago, IL, 1994

[Ousterhout89] Ousterhout, J. *Why aren't operating systems getting faster as fast as hardware. WRL Technical Note* (TN-11): 1989

[Patel92] Patel, K., Smith, B. C. and Rowe, L. A., *Performance of a Software MPEG Video Decoder,* University of California, Berkeley, 1992

[Rashid88] Rashid, R., et al. *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures*, IEEE Transactions on Computers, Vol. 37, No. 8, p896-908, Aug, 1988

[Sites92] Sites, R., *Alpha Architecture Reference Manual,* Digital Press, 1992

[Smith91] Smith, M.D. *Tracing with pixie.* Stanford University, Stanford, CA. 1991

[Uhlig94a] Uhlig, R., Nagle, D., Mudge, T., Sechrest, S., *Kernel-based Memory Simulation,* ACM Sigmetrics Conference on Measuring & Modeling of Computer Systems, Vol 22, May, 1994

[Uhlig94b] Uhlig, R., Nagle, D., Mudge, T., Sechrest, S. *Trap-driven Simulation with Tapeworm II,* In the proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, ACM, 132-144, 1994