Software Mechanisms for Multiprocessor TLB Consistency

Shin-Yuan Tzou

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

In a shared-memory multiprocessor, a page table entry (PTE) may be replicated in multiple translation lookaside buffers (TLBs), causing an inconsistency problem when the PTE is updated. More generally, this problem exists among virtually-tagged caches, which keep PTE information, such as protection bits, in every cache line. Operating systems and applications that exploit virtual memory remapping must consider the overhead of synchronizing TLBs.

We explore a spectrum of software TLB synchronization algorithms for various consistency semantics and TLB characteristics. We analyze and simulate the performance of the three most general ones: *2-phase*, *optimistic-synchronous*, and *optimistic-asynchronous*. The queueing models for these algorithms do not have product-form solutions because of the interaction among processors (for example, the 2-phase algorithm enforces locking by stalling processors). Instead, we obtain approximations using a computationally efficient iterative analysis method, the accuracy of which is verified by simulation results.

The performance results show that software TLB synchronization algorithms do not scale well with (1) the number of processors, (2) the rate of PTE updates, or (3) the overhead of flushing a TLB entry. Hence TLB synchronization should be avoided in some future architectures (e.g., scalable cache-coherent shared-memory multiprocessors) and under some workloads (e.g., moving high-bandwidth multimedia data to a user address space by virtual memory remapping). To this end, we describe mechanisms for tolerating TLB inconsistency, and classify them according to three fundamental types of tolerable inconsistency: safe, transient and trusted inconsistency. We also discuss how to fit these mechanisms into the software architecture of the virtual memory system.

To my parents, my wife and my two daughters.

Acknowledgements

I would like to thank many people who have helped me finish this dissertation. David Anderson, my research advisor, provided financial support and valuable advice throughout my graduate study at Berkeley. He worked closely with me on the DASH project, on which Chapter 5 is based. We had many inspiring discussions and arguments, through which we learned how to design and build a system. David also helped a lot in the preparation of this dissertation. Domenico Ferrari, who created the DASH project with David, always gave me confidence when I needed it. He also provided valuable comments on the dissertation; his thoroughness greatly improved the quality of it. Charles Stone, the third committee member, taught me BLSS, the tool I used for simulation.

Many others have helped me along the way and deserve thanks. Raj Vaswani implemented the DASH message-passing system. Ramesh Govindan implemented part of the DASH virtual memory system. G. Scott Graham participated in the design of the DASH virtual memory system and the mechanisms for TLB consistency. This research was partly inspired by my CS252 term project on the in-cache address translation mechanism of SPUR. Young-Chul Shim was my partner of the project; David Wood helped a lot in setting up the project. I would also like to thank other members of the DASH group and members of the Tenet group for numerous stimulating technical conversations.

Most of all, I wish to thank my wife. She fully supported my decision to return to school after we had been used to a comfortable life style, and sacrificed her own ambitions to mine. She cared our children and gave me a warm home while I spent most of my time on my own work. Her patience and constant support have made this dissertation possible.

The material presented here is based on research supported in part by the Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871, monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089, the California MICRO program, AT&T Bell Laboratories, Digital Equipment Corporation, International Business Machines Corporation, Olivetti S.p.A, Hitachi, and the International Computer Science Institute.

Contents

Acknowledgements ii
Contents iii
Figures vi
Tables vii

1. Introduction 1
1.1. Motivation 1
1.2. Research Issues 3
1.3. Organization of the Dissertation 3

2. The Problem of TLB Inconsistency in Shared-Memory Multiprocessors 5
2.1. The Machine Model and Terminology 5
2.1.1. Overview of the model 5
2.1.2. TLB parameters 6
2.1.3. Discussion of concurrent PTE updates 7
2.2. Problem Statement 9
2.3. Discussion of In-Cache Address Translation 10
2.3.1. Eliminating the need for separate TLBs 10
2.3.2. Address translation information in every cache line 11
2.3.3. TLB inconsistency without separate TLBs 12
2.4. Classification and Software Causes of TLB Inconsistencies 13
2.4.1. Types of TLB inconsistency 13
2.4.2. OS functions causing TLB inconsistency 15
2.4.2.1. Supporting a single-level store 15
2.4.2.2. Moving data between virtual address spaces 16
2.4.2.3. Supporting database functions 16

3. Software TLB Synchronization Algorithms 17
3.1. Semantics of Consistency 17
3.2. Algorithms for Updating a Page Table Entry Consistently 18
3.2.1. Overview 18
3.2.2. Basic algorithms 20
3.2.3. Variants of basic algorithms 24
3.3. Discussion 25

3.3.1. Disabling interrupts 25
3.3.2. Interprocessor requests and batching 26
3.3.3. Deadlocking 27

4. The Performance of TLB Synchronization Algorithms 28
4.1. The Performance Model 28
4.1.1. Event arrival and execution 28
4.1.2. Interarrival and service times 29
4.1.3. Algorithm-specific parameters 30
4.1.4. Performance measures 30
4.2. Performance Analysis 30
4.2.1. The 2-phase algorithm 30
4.2.1.1. Method of analysis 31
4.2.1.2. Acknowledgement and synchronization time 31
4.2.1.3. Busy periods 33
4.2.1.4. The complete procedure 36
4.2.2. The optimistic-synch algorithm 37
4.2.2.1. Arrival rates 37
4.2.2.2. Acknowledgement and synchronization time 38
4.2.2.3. Busy periods 40
4.2.2.4. The complete procedure 41
4.2.3. The optimistic-async algorithm 41
4.2.3.1. CPU overhead 42
4.2.3.2. Latency 42
4.3. Simulation 44
4.3.1. The simulators 44
4.3.2. Comparison of simulation and analytic results 45
4.3.2.1. The 2-phase algorithm 46
4.3.2.2. The optimistic-synch algorithm 47
4.3.2.3. The optimistic-async algorithm 47
4.4. Performance Results 48
4.4.1. Choosing workloads 50
4.4.2. The effect of the number of processors 52
4.4.3. The effect of the request rate 54
4.4.4. The effect of service time 54
4.5. Summary and Conclusions 58

5. Tolerating TLB Inconsistency 60
5.1. Three Types of Tolerable TLB Inconsistency 60
5.1.1. Safe inconsistency 60
5.1.2. Transient inconsistency 61
5.1.3. Trusted inconsistency 63

5.2. Software Structure of the VM System 64
5.2.1. A portable model 64
5.2.2. Hiding TLB inconsistency inside the machine-dependent part 65
5.2.3. Handling TLB inconsistency outside the machine-dependent part 65
5.3. A Virtual Memory System Design 66
5.3.1. Overview and context 66
5.3.1.1. Object-oriented programming 66
5.3.1.2. Local kernel structure 67
5.3.1.3. The DASH virtual memory system 68
5.3.2. Interface to the machine-dependent part 69
5.3.3. Paging 72
5.3.3.1. Overview: flow and states of physical pages 72
5.3.3.2. The unmapper process 72
5.3.3.3. Launderer processes 74
5.3.3.4. Managing TLB inconsistency 74
5.3.4. Message-passing using virtual-memory remapping 75
5.3.4.1. Software copying vs. VM remapping 75
5.3.4.2. Initial experience of VM remapping on a uniprocessor 76
5.3.4.3. An overview of the DASH message-passing system 77
5.3.4.4. The IPC region of the VM system 80
5.3.4.5. Message-passing operations 82
5.3.4.6. Managing TLB inconsistency 84
5.3.5. Mechanisms not suitable for tolerating TLB inconsistency 85

6. Related Work 87
6.1. Software Mechanisms 87
6.2. Hardware Mechanisms 88
6.3. Performance Evaluation 90

7. Concluding Remarks 91
7.1. Contributions 91
7.2. Summary of Results 91
7.3. Future Work 93

Bibilography 95

Figures

1.1. Kernelization creates extra address space boundaries 2
2.1. The model of a shared–memory multiprocessor 6
2.2. A PTE may be overwritten as a result of setting status bits 9
2.3. Updating a PTE on machines with virtually tagged caches 12
3.1. Consistency semantics 18
3.2. The 2-phase algorithm 22
3.3. The optimistic algorithm 23
4.1. A simplified timing diagram of the 2-phase algorithm 34
4.2. Simulation vs. analysis: the 2-phase algorithm 47
4.3. Simulation vs. analysis: the op-synch algorithm 48
4.4. Simulation vs. analysis: the op-async algorithm 49
4.5. The effect of the number of processors on performance 53
4.6. The effect of the request rate on performance 55
4.7. The effect of service time on performance 57
5.1. Transient TLB inconsistency 62
5.2. A portable model of the virtual memory system 64
5.3. An overview of the DASH VM system 69
5.4. The flow of physical pages 73
5.5. Throughput of message-passing: initial experience 77
5.6. Cost breakdown of message-passing: initial experience 78
5.7. Software architecture of the message-passing system 79
5.8. User object reference tables 83

Tables

2.1. Some real TLBs and their properties 8
2.2. Classification of TLB inconsistency 14
3.1. Summary of algorithms for solving TLB inconsistency 19
5.1. Simplified interface to kernel's message-passing system 68
5.2. Simplified interface to the **VAS_MD** class 70
5.3. Interface to the `IPC_REGION_MGR` class 82

Chapter 1
Introduction

The topic of this dissertation is translation lookaside buffer (TLB) consistency in shared-memory multiprocessor computer systems. We first study and evaluate software TLB synchronization algorithms, and show that the performance of these algorithms does not scale well with the number of processors, the rate of TLB synchronization operations, or the overhead of flushing a TLB entry. We then propose a virtual memory system design that tolerates TLB inconsistency. The design avoids TLB synchronization by exploiting trust relationships and lazy remapping. When TLB synchronization is necessary, the design does it efficiently by exploiting asynchrony.

The research area of this work is Operating Systems. It also overlaps the area of Performance Evaluation because it develops an iterative technique for analyzing the performance of TLB synchronization algorithms, and verifies the analysis with simulation. Additionally, it is related to the area of Computer Architecture because it reviews hardware mechanisms for TLB consistency, and shows the connection between the problem of TLB inconsistency and that of meta-data inconsistency of virtually tagged caches.

The first chapter explains motivations, identifies research issues, and previews the rest of the dissertation.

1.1. Motivation

Most modern machine architectures provide paged virtual memory. Such machines use page tables to define virtual-to-physical memory mappings of virtual address spaces. Operating systems use paged virtual memory for various purposes, such as (1) supporting demand paging, and (2) logically moving virtual pages between virtual address spaces. The second usage (or *virtual memory remapping*) potentially reduces the overhead of data movement by avoiding software data copying. Consequently, it makes the following directions in operating systems research more feasible.

- *Kernelization*, *i.e.*, reducing the size and functionality of an operating system kernel by moving services out of the kernel into user-level processes. Such a system is more extensible and maintainable, and makes it easier for multiple servers to coexist as different user processes or libraries. The V kernel [Che84] and QuickSilver [HMS88] are two representative examples. Mach is also targeting this direction [Ras89]. However, this approach creates additional address space boundaries. Invoking a system service may involve passing control and data across address space boundaries many times, thereby limiting the performance of such a system. Figure 1.1 illustrates this problem.

- *High-performance I/O*. Processor performance has been improved at a much faster pace than that of I/O subsystems. As a result, I/O has become the performance bottleneck in many systems. For example, no existing system can achieve 1 Gbps user-to-user communication throughput, although 100-MIPS processors and 1 Gbps fiber optics are available now [Lei88]. To achieve high I/O performance, many mechanisms in the operating systems and host interface must be drastically improved. Data movement in the operating system is such a mechanism. At very high throughput, an extra data copy could easily double the overhead of protocol processing or the overhead of delivering data to a user process.

Virtual memory remapping can eliminate the data movement bottleneck only if updating a page table entry (PTE) is significantly faster then copying a page. The success of remapping in uniprocessors does not imply the success in shared-memory multiprocessors, as the performance tradeoff is different [Fit86, FiR86]. In most uniprocessors, updating a PTE takes only a few machine cycles. On the other hand, in shared-memory multiprocessors, a PTE may be replicated in multiple TLBs, causing an inconsistency problem when the PTE is updated. A TLB synchronization algorithm usually involves interprocessor interrupts and synchronization, and is potentially expensive. Hence the overhead of updating a PTE, which includes the overhead of managing TLB

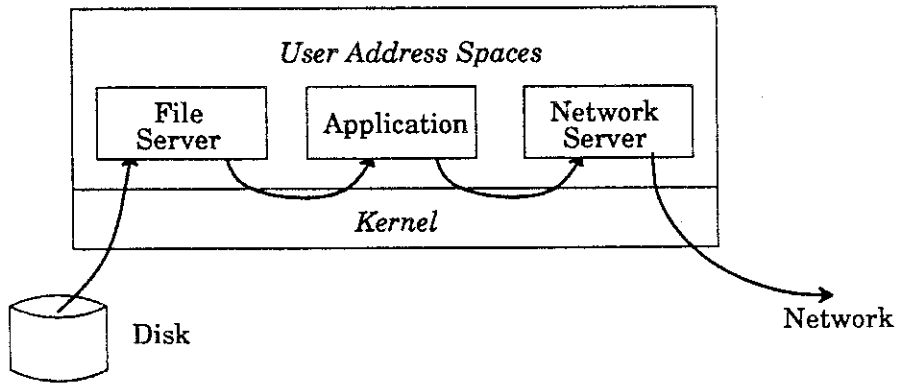


Figure 1.1. Kernelization creates extra address space boundaries. A piece of data is often moved across address space boundaries many times.

inconsistency, could also be high.

The goal of this dissertation is to understand the performance impact of maintaining TLB consistency on virtual memory remapping, and to develop mechanisms for efficiently exploiting virtual memory remapping in shared-memory multiprocessors. This work links the above two operating systems research directions (kernelization and high-performance I/O) with the shared-memory multiprocessor architecture, which is cost-effective, has abundant computing power, and can support parallelism.

This dissertation differs from previous research on TLB consistency in its emphasis on scalability. The performance of TLB synchronization mechanisms is not critical in most existing shared-memory multiprocessors, because they typically have a small number processors and update PTEs at a low rate (primarily for demand paging). On the other hand, we intend to support virtual memory remapping at a high rate for kernelization and high-performance I/O. We also intend to support systems with a large number of processors and other hardware features that make TLB synchronization expensive.

1.2. Research Issues

The major research issues of this dissertation are the following.

(1) Is TLB inconsistency a fundamental problem of shared-memory multiprocessors? Is it still relevant if the hardware eliminates TLBs by using virtually tagged caches?

(2) What is the range of possible software TLB synchronization algorithms? Is hardware support for TLB consistency necessary?

(3) What are the performance tradeoffs among different algorithms? How can the performance of TLB synchronization algorithms be evaluated?

(4) What are the likely workloads of TLB synchronization or virtual memory remapping mechanisms? Do the algorithms perform and scale well under these workloads?

(5) Can we take advantage of high-level OS semantics in achieving scalable TLB consistency mechanisms? Can operating systems reduce the overhead of TLB synchronization by tolerating TLB inconsistency? How?

1.3. Organization of the Dissertation

We address the above issues and carry out the research in four stages: (1) understanding the problem, (2) finding direct solutions to the problem, (3) evaluating these solutions and pointing out their drawbacks, and (4) proposing a system design that overcomes the drawbacks.

The organization of this dissertation reflects these stages. Chapter 2 states the problem of TLB inconsistency in shared-memory processors. It also classifies TLB inconsistency, and summarizes how and when operating systems remap pages. Chapter 3 defines consistency semantics, and presents a set of software TLB synchronization algorithms for various hardware characteristics and consistency semantics. Chapter 4 evaluates the performance of these algorithms. It analyzes three major algorithms using an iterative analytic method, and verifies analytic results with simulation. It also predicts the workload of future systems. Chapter 5 describes principles for tolerating TLB inconsistency, and presents an integrated system design that applies those principles. This chapter also discusses the software structure of the virtual memory system for handling TLB inconsistency.

Finally, Chapter 6 reviews related work on TLB inconsistency, including possible hardware solutions; Chapter 7 summarizes the main results and contributions of this work, and discusses future research directions.

Chapter 2

The Problem of TLB Inconsistency in Shared-Memory Multiprocessors

This chapter describes the problem of TLB inconsistency in shared-memory multiprocessors. It is organized as follows. Section 2.1 presents our model of a shared-memory multiprocessor. Section 2.2 states the problem. Section 2.3 argues that the problem exists even when in-cache address translation eliminates separate TLBs. Section 2.4 classifies TLB inconsistency, and discusses operating system operations that cause inconsistency.

2.1. The Machine Model and Terminology

This section presents our model of a *shared-memory multiprocessor*. The purpose is to specify the target machine, and to define the terminology. We concentrate on address translation, page tables, translation lookaside buffers, and setting dirty and referenced bits. We omit details that are irrelevant to the TLB inconsistency problem. Our model only represents one type of multiprocessors; other types can be found in [Ens74], [Sat80], [GaP85], and [Mey85].

2.1.1. Overview of the model

As shown in Figure 2.1, the system contains N processors sharing a common memory subsystem (or *main memory*) via an interconnection network. Processors run in parallel, and each of them can access any location of the memory subsystem. The hardware supports instructions for serializing concurrent updates to a shared memory location, such as instructions for atomic read-modify-write or atomic fetch&op.

The system supports virtual memory. A *physical address* uniquely identifies a memory location in the memory subsystem. The software, on the other hand, makes a memory reference using a *virtual address* in a *virtual address space (VAS)*. There are multiple virtual address spaces, so a virtual address is not unique[1]. *Address translation* is the procedure that converts a virtual address generated by the software to a physical address used by the main memory; it is done within a processor[2].

[1] Some systems extend a virtual address into a *global virtual address*, say, by prepending a VAS identifier to it. In our discussion, we do not distinguish between a virtual address and a global virtual address unless otherwise noted.

[2] We exclude architectures that translate addresses along the interconnection network or at the memory subsystems, such as in the design proposed by Teller *et al.* (see Section 6.2) [TKS88].

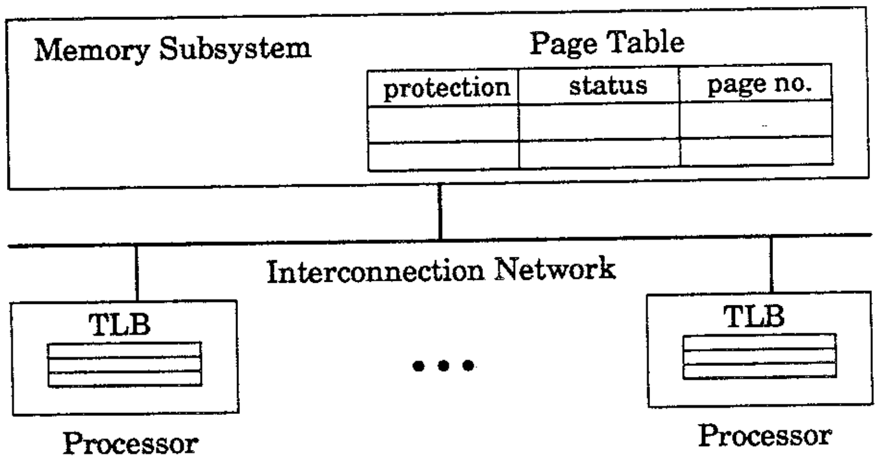


Figure 2.1. The model of a shared-memory multiprocessor.

Page tables store mapping information for address translation. They are located in main memory, and are collections of *page table entries (PTEs)*. The granularity of address translation is a page. Both main memory and VASs are divided into pages (virtual and physical pages respectively). Each PTE stores the mapping information and status of a virtual page. It contains (a) the physical page to which the virtual page is mapped, if any; (b) protection bits defining access rights to the page, such as read-only and read/write; and (c) status of the page, such as the *referenced bit* and *dirty bit*[3]. Page tables are accessible to the hardware; our discussion ignores other forms of mapping information that are used exclusively by the operating system.

Each processor has a *translation-look-aside buffer* (TLB) for caching recently used page table entries [Lee60]. A PTE stored in the TLB (possibly in a different format) is called a *TLB entry*. *TLB hit* means that a needed PTE is in the TLB on address translation; *TLB miss* means the opposite. The hardware supports instructions for flushing TLB entries on a processor, but it does not ensure consistency among a PTE and the corresponding TLB entries. We will discuss more on TLBs in the next section.

2.1.2. TLB parameters

[3] The referenced bit is set if this page has been accessed since the bit was cleared. The dirty bit is set if the page has been modified.

A processor carries out a memory reference as follows[4]. It first loads the corresponding PTE into its TLB, if it is not already in the TLB. It then translates the virtual address using a TLB entry. The reference is rejected if the virtual page is not mapped to a physical page, or the type of access is not allowed. A rejected memory reference generates a fault, a software exception handled by the operating system. If the reference is accepted, the processor accesses the data and, when necessary, adjusts the referenced and dirty status in the TLB entry and the PTE.

Our model allows two variables in the above process:

- A PTE may be loaded into a TLB by hardware, or by software (via a trap).
- The referenced or dirty bit of a PTE in main memory may or may not be set atomically.

The second variable is an issue because the unit of memory writes is a word instead of a bit in most systems. In other words, to set a bit in a PTE, a processor must write a complete word. We consider a status bit being set atomically if the update to the PTE is serializable with other concurrent PTE updates. In general, a processor may set a status bit in five ways (1) not supporting the bit at all[5], (2) generating an exception and letting the software do it, (3) performing an atomic read-modify-write (R-M-W) or fetch&op cycle to update the PTE, (4) performing a non-atomic operation to update the PTE (the opposite of (3)), or (5) skipping the read cycle, and formulating the new PTE using information stored in the TLB. We consider the bits being set non-atomically in case (4) and (5). Table 2.1 lists these two variables for some real systems.

In addition, our model has a third variable for TLBs: whether a TLB can be controlled by only one processor, or by all processors. In most existing systems, a TLB is internal to a processor and cannot be accessed externally. The only known exception is the Motorola MC88200 cache and memory management unit (CMMU), which can be controlled by all MC88100 CPUs, or even by other CMMUs [MMM88a, MMM88b].

Chapter 3 will present TLB synchronization algorithms based on the three variables discussed in this section.

2.1.3. Discussion of concurrent PTE updates

This section discusses the serializability of concurrent PTE updates, which are possible on a multiprocessor. A PTE may be updated in three situations:

(1) for setting the dirty bit on a memory reference,

(2) for setting the referenced bit on a memory reference, and

[4] Without losing generality, this discussion does not consider data caching.

[5] The operating system either does not use the bit, or emulates it with a software mechanism, which is assume to be atomic.

Table 2.1. Some real TLBs and their properties.

TLB	multiprocessor systems	loaded by	set PTE ref bit by	set PTE dirty bit by
VAX	8800	H/W	S/W [1]	non-atomic [2]
NS32332	Encore Multimax	H/W	S/W [1]	non-atomic [2]
SUN3 MMU	none	S/W	S/W [3]	S/W [3]
Intel 386 on chip TLB	Sequent Symmetry	H/W	atomic R-M-W	atomic R-M-W
Intel N10 on chip TLB	Olivetti M110 (being developed)	H/W	atomic R-M-W	S/W [4]
IBM RP3	RP3	S/W[5]	S/W [1]	S/W [1]

[1] The hardware does not set the status bit. The operating system emulates it by invalidating the page (for the referenced bit) or protecting the page as read-only (for the dirty bit).

[2] The bit is set by a non–atomic R–M–W on a TLB miss, and by a write on a TLB hit.

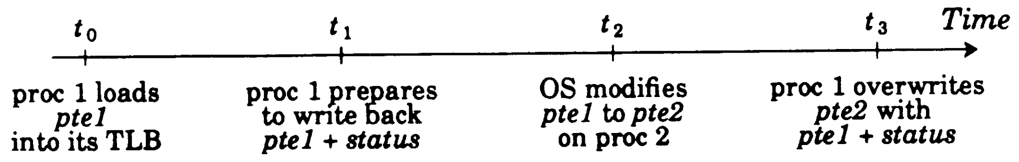
[3] The hardware set status bits only in the MMU. The operating system explicitly reads a TLB entry when it needs a status bit.

[4] The hardware generates a trap on the first write to a page.

[5] RP3 was designed to load TLB entries by hardware, but the RP3 prototype loads TLB entries by a software TLB miss handler [Ros89].

(3) for other software operations.

In our machine model as well as in some real systems, (1) or (2) may be done non-atomically (see Section 2.1.2 and Table 2.1). If not handled properly, this may cause a newly updated PTE to be overwritten, as illustrated in Figure 2.2.



t_0
t_1
t_2
t_3
Time
proc 1 loads *pte1* into its TLB
proc 1 prepares to write back *pte1 + status*
OS modifies *pte1* to *pte2* on proc 2
proc 1 overwrites *pte2* with *pte1 + status*

Figure 2.2. A PTE may be overwritten as a result of setting status bits. The hardware writes a whole word to set a bit in a PTE. The rest of the word may come from a stale TLB entry.

but nothing else.

On the other hand, (2) has undesirable effects on (1). It may cause a dirty bit to be lost. Further, when the dirty bit is set by hardware, this problem cannot be fixed by using software algorithms. Hence, if the operating system relies on dirty bits, it should suppress non-atomic PTE updates caused by setting referenced bits. The operating system can force the hardware not to set the referenced bit by setting the bit in every PTE. When a processor loads a PTE into a TLB entry, it will never attempt to set the referenced bit in the PTE again, because the bit has already been set. The operating system can still obtain page reference information by initially invalidating a PTE to cause a page fault on the first reference, as in VAX/VMS [LeL82]. Wood and Rosenburg also showed software mechanisms for implementing dirty and referenced bits [Ros89, WoK89].

2.2. Problem Statement

In a shared-memory multiprocessor, a PTE may be replicated in multiple TLB entries at the same time. Without hardware consistency support, inconsistency may occur when any of the replicated copies is changed. Here, we concentrate on the inconsistency between a PTE and a TLB entry, although TLB entries may differ among themselves too.

The operating system uses PTEs to define how the memory system should behave, and makes memory management decisions based on status bits stored in PTEs. On the other hand, a processor translates addresses using TLB entries, and writes status bits back to PTEs according to the state of TLB entries. The following problems may occur if a PTE is inconsistent with a corresponding TLB entry.

(1) A memory reference may produce incorrect results, *e.g.*, when a PTE allows read-only access but the corresponding TLB entry

allows read/write access.

(2) A processor may not set status bits properly, *e.g.*, when the dirty bit is zero in a PTE but is one in the corresponding TLB entry.

(3) A processor may overwrite a newly updated PTE with a old TLB entry if it sets a status bit non-atomically (see Figure 2.2).

The above problems can be solved by flushing the corresponding TLB entries after updating a PTE. This operation is straightforward in a uniprocessor, or even in a multiprocessor if a PTE is only cached in one TLB at a time. On the other hand, flushing multiple TLBs is hard because it involves interprocessor requests and synchronization. Further, the overwriting problem (problem (3)) complicates the updating of a PTE.

A multiprocessor system may have a PTE cached in multiple TLBs simultaneously if the operating system (1) supports multiple concurrent threads in one VAS, (2) supports symmetric pageable kernel, or (3) does not flush the entire TLB of a processor on a context switch. The last case is possible when the hardware appends an identifier to each TLB entry, as in MIPS [MMM86, TBJ88]. Many multiprocessor operating systems have some or all of the above features. Without these features, the benefit of having multiple processors is limited, because only disjointed jobs may run in parallel.

2.3. Discussion of In-Cache Address Translation

This section shows that TLB inconsistency is a fundamental problem of shared-memory multiprocessors. It exists even when separate TLBs are eliminated by using virtually tagged caches[6].

2.3.1. Eliminating the need for separate TLBs

Modern architectures use cache memory to improve system performance [Hil87, PHH88, Smi82]. Although cache design is an important research topic, is not the theme of this dissertation. We simply assume caches are correct, coherent, and transparent to our machine model. We are interested only in one issue of cache design—whether a cache is *physically tagged*[7] or *virtually tagged*.

A physically tagged cache is indexed and tagged by physical addresses. To check whether there is a cache hit on a memory reference, the virtual address generated by the software must be translated into a physical address. It is possible to do address translation and cache access in parallel, but the cache size must be less than the page size times the set associativity. This restriction is undesirable for systems that require both a fast instruction cycle time and a large cache size.

[6] To be more precise, in this case the problem should be defined as the inconsistency of PTE information.

[7] Every cache line stores memory data as well as an address tag. On a memory reference, the tags of cache lines are compared with the target address to determine whether there is a hit.

A virtually tagged cache, on the other hand, is indexed and tagged by virtual addresses. Such a cache does not need address translation on a cache hit, in which case the physical address address is not used at all. Thus, it removes the above restriction of physically tagged caches, though it has its own problems, *e.g.*, the difficulty of supporting virtual address synonyms [Goo87, Hil86]. Virtually tagged caches have been used in many real or research systems, such as Sun 3/200 series [SSS85], Intel 860 [II188], SPUR [Hil86], and VMP [CSB86].

The importance of a TLB decreases when address translation is no longer on the critical path on a memory reference. Consequently, some architects have made the tradeoff to eliminate separate TLBs in order to reduce hardware complexity and to use precious chip area for more useful purposes. Without separate TLBs, the hardware directly accesses the PTE to translate an address. Often, the needed PTE is itself cached; accessing it is not a main memory reference but a more efficient cache reference. In this sense, we can view the system as having a "TLB" in the cache. Wood *et al.* called this mechanism *in-cache address translation* [Rit85, WEG86].

In-cache address translation eliminates separate TLBs, and the multiprocessor cache coherency mechanism, assuming there is one, ensures that cache copies of a PTE are consistent. This *seems* to have solved the problem of TLB inconsistency. On the contrary, it has not. The following sections explain why.

2.3.2. Address translation information in every cache line

Every virtually tagged cache line stores meta-data in addition to data. Specifically, every line keeps the protection bits of PTE corresponding to the line. These bits are essential to performance, for without them a cache reference would need an extra PTE reference to check protection. Moreover, in some systems, a cache line even keeps the physical address of the line to reduce the cost of writing back a line. Therefore, every cache line has, at least partially, the functionality of a TLB entry.

With in-cache address translation, the address translation information of a page is stored in:

(1) the PTE,

(2) the cached copies of the PTE, and

(3) the control field of every cache line.

Indeed, in-cache address translation eliminates separate TLBs, and achieves consistency between (1) and (2). However, it also turns every cache line into a TLB entry, and introduces an inconsistency problem between (3) and (1)/(2) (see Figure 2.3).

No existing system provides hardware consistency support between (3) and (1)/(2). In fact, this is difficult for hardware. To do so, the hardware must be able to (a) detect the updating of a PTE, which is a regular memory write; (b) calculate the address of the target page using the address of the PTE; and (c)

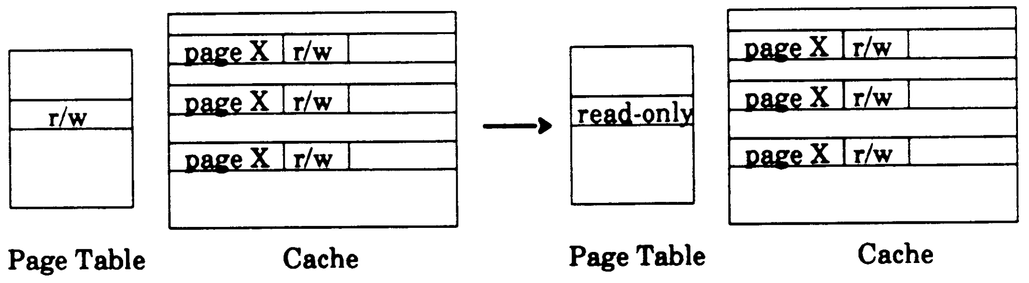


Figure 2.3. Updating a PTE on machines with virtually tagged caches. A writable page with three lines cached (right) is reprotected as read-only (left). A subsequent write to the page may be granted because the three cache lines are inconsistent with the newly updated PTE.

atomically modify/invalidate all cache lines corresponding to the target page on all processors. Most systems, instead, simply provide a command to flush a cache line, and/or all cache lines in a certain address range. It is up to the software to make sure that the virtual address caches are flushed properly when a PTE is modified.

Flushing a virtually tagged cache line may be more complicated then flushing a simple TLB entry. If the cache does not store the physical address of the page, an address translation is needed to write a dirty line back to main memory. The situation would be complex if the PTE for this line has been changed after the line was brought into the cache. In this case, the address translation may fail, generating a page fault.

The total overhead of cache flushing depends on basic hardware costs, software overheads, and, most important, the rate of page table updating. Cheng reported that on a Sun workstation running UNIX, the overhead of cache flushing ranges from 0.13% to 3% of its total CPU time [Che87].

2.3.3. TLB inconsistency without separate TLBs

We now examine the three problems listed in Section 2.2, and show that they exist in machines with virtually tagged caches. First, using a stale cache line is equivalent to using a stale TLB entry. A cache line is brought into the cache based on a PTE. It becomes stale if the PTE is updated later, and may cause a memory reference to produce incorrect results. Second, status bits may not be set properly. For example, if the operating system clears the dirty bit of a page while a writable cache line for the page exists, a subsequent write to the page may not cause the dirty bit to be set in the PTE. Finally, a PTE may be overwritten if (a) the cache does not set status bits in PTEs atomically, or (b) the cache does not store the physical address of the line in the control field. We explain (b) in the next paragraph; (a) is obvious.

To write back a line, the cache needs to perform address translation to obtain the physical address of the line. If the PTE corresponding to the cache line has been updated (say, invalidated) since the line was loaded, the address translation may fail, generating an exception. To avoid data loss, the exception handler has to change the PTE back to its old value to finish the writeback operation. In other words, the cache forces the software to overwrite the PTE. Although the software can restore the PTE later, the old PTE may have been used to load other cache lines by that time.

The basic technique for fixing the above problems is the same as that for fixing TLB inconsistency—flushing stale entries. Other issues such as interprocessor synchronization are also the same for both cases. Hence the algorithms for TLB synchronization work for virtually tagged cache synchronization. The only difference is that flushing a TLB entry is usually faster than flushing all cache lines.

We have shown that the inconsistency between a PTE and TLB entries is equivalent (for problems as well as solutions) to the inconsistency between a PTE and the meta-data stored in virtually tagged lines. In the rest of this dissertation, we will use the term "TLB inconsistency" to refer both. To be more precise, we should define it as the inconsistency among replicated copies of PTE information. We still use "TLB inconsistency" because it is a well-known term.

2.4. Classification and Software Causes of TLB Inconsistencies

This section classifies TLB inconsistencies, and enumerates the operating system functions that cause the various types of TLB inconsistencies.

2.4.1. Types of TLB inconsistency

Table 2.2 lists possible ways that a TLB can differ from the corresponding PTE, and, for each one, gives the potential damage and how it may occur. The goal here is to enumerate all types of TLB inconsistency; Section 2.4.2 discusses how real applications change PTEs.

The table uses the following legends to represent inconsistency.

- For a status bit, condition T0→1 means it is zero in the TLB entry but has the potential of becoming one. Condition T1P1 means it is one in both the PTE and the TLB entry, or can never become one in the TLB entry (*e.g.*, the dirty bit of a read-only TLB entry). Condition T1P0 means it is one in the TLB entry but zero in the PTE.
- The protection bits define multiple access rights, such as kernel write and user write. Condition T=P means the access rights defined by both

Table 2.2. Classification of TLB inconsistency.

How PT and TLB differ		When the OS updates a PTE in main memory to	Possible damage
status bit	protection bits and page #		
T0→1	T<P and EQ	increase rights	overwriting PTE, recoverable rejecting reference, recoverable
T0→1	T>P or NE	reduce rights, or change page number	overwriting PTE, nonrecoverable illegal reference, nonrecoverable
T1P1	T<P and EQ	increase rights	rejecting reference, recoverable
T1P1	T>P or NE	reduce rights, or change page number	illegal reference, nonrecoverable
T1P0	T=P and EQ	clear status bits	wrong PTE status
T1P0	T<P and EQ	clear status bits, and increase rights	rejecting reference, recoverable wrong PTE status
T1P0	T>P or NE	clear status bits, and increase rights or change page number	illegal reference, nonrecoverable wrong PTE status

entries are identical. Condition T<P means the rights defined in the TLB entry are a proper subset of those defined in the PTE. Condition T>P covers the remaining case.

- For the physical page number, condition EQ means both entries have the same number, and NE means the opposite.

The conditions for both status bits may be different, *e.g.*, the referenced bit satisfies T1P1 and the dirty bit satisfies T0. For simplicity, Table 2.2 considers only one status bit. The possible damage listed in the table are the three problems described in Section 2.2, except that they are further divided into recoverable and nonrecoverable. An error is recoverable if it can be detected (*e.g.*, via a page fault) and corrected.

A recoverable error may occur when a TLB entry has less access rights than the corresponding PTE. Such a TLB entry may reject a legal memory reference (*i.e.*, a reference allowed by the correct PTE), causing a page fault. The page fault handler recovers the problem by flushing the stale TLB entry on the faulting processor, and resuming the interrupted memory reference. Such a TLB entry may also be written back to the PTE as a result of setting status bits, and may even be loaded again into other TLBs. We assume that the operating system stores address translation information also in other data structures, such as in machine-independent memory maps; thus it can detect and fix an overwritten PTE on a page fault caused by insufficient access rights.

2.4.2. OS functions causing TLB inconsistency

This section examines operating system functions that update PTEs; Section 4.4.1 discusses their workloads. In these functions, the operating system updates a PTE only in limited ways; it does not update a PTE arbitrarily. Basic PTE updates are:

(1) clearing a referenced bit,
(2) clearing a dirty bit,
(3) validating a page (increasing access rights),
(4) invalidating a page (decreasing access rights),
(5) reprotecting a page to read-only (decreasing access rights), and
(6) reprotecting a page to read/write (increasing access rights).

The operating system does these operations one at a time. It does not combine several updates into a single one. Note that changing the physical page number is not listed, because it is normally done when a PTE is invalid.

Only (4) and (5) in the above list may cause nonrecoverable errors. Clearing status bits may cause a PTE to have wrong status bits in general, but is harmless in the specific cases described in the next section.

2.4.2.1. Supporting a single-level store

This includes both paging and memory mapped file I/O. The operations that change PTEs are paging in, paging out, selecting candidates for paging out, creating a virtual address space (VAS), and deleting a VAS.

For paging in, the operating system allocates a new page, reads it from the backing store if necessary, and validates it. For paging out, the system invalidates a page (probably reprotects it to read-only first), writes it to the backing store if it is dirty, and clears the dirty bit. Only the first step of paging out may cause nonrecoverable errors. Clearing the dirty bit is done when the page is invalid or read-only, in which case the TLB may never write the dirty bit back to the PTE.

For page replacement, the system periodically clears the referenced bit of PTEs. If the bit is not set properly due to TLB inconsistency, the page replacement algorithm may make suboptimal decisions, but no nonrecoverable damage may occur. If the hardware has no referenced bits (*e.g.*, the VAX), the system emulates them by invalidating the page [LeL82]. Although in general invalidating a page is dangerous, it is harmless in this particular case. The page is actually "valid" even though the PTE has been invalidated. Therefore, accessing a page using such a stale TLB entry will get to the correct page.

Creating a VAS or mapping a file to memory does not update PTEs that were originally valid. Deleting a VAS or unmapping a file is equivalent to a sequence of paging out operations.

2.4.2.2. Moving data between virtual address spaces

An operating system moves data between VASs either by copying or by VM remapping. Most systems copy data indirectly, *i.e.*, from a source VAS to a kernel buffer, then to a destination VAS. Direct copying is desirable when the size of data is large. To do so, the operating system maps the buffer from one VAS to another, copies data, and unmaps the buffer again. Mapping is done by validating pages in the buffer, and unmapping is done by invalidating them. If virtual address synonyms is not desirable, the operating system invalidates pages in the original VAS before mapping them into another VAS.

VM Remapping (moving physical page numbers between PTEs) eliminates the need for copying. This is usually done in two ways: (1) unmapping pages from the source VAS and mapping them to the destination VAS, and (2) mapping pages in copy-on-write[8] mode in both VASs. Basic PTE updates for VM remapping include validating a page, invalidating a page, and reprotecting a page to read-only.

Sometimes the hardware only allows I/O in a limited range of virtual addresses, *e.g.*, the DVMA area of Sun 3 workstations [SSS85]. Sun UNIX moves data between I/O buffers and the DVMA area by remapping. Again, the relevant primitives are validating a page, and invalidating it later.

2.4.2.3. Supporting database functions

Some database systems use VM techniques to implement transaction systems [ChM88, Sto85]. In IBM's 801 system, the data manager maps file records to virtual memory pages. When a record is not being updated, its data page is protected as invalid or read-only. The data manager detects the beginning of a transaction by a page fault on the first reference or write to the record. It then takes the necessary actions, such as logging the record, validating the data page, and resuming the faulting process. It reprotects the page again when the transaction commits.

VM techniques are also used to speed up crash recovery [Sul90]. The data manager protects most pages either as invalid or read-only. This simplifies crash recovery because a wildly running process will not destroy a page that is not writable.

The basic PTE updates used in the above two cases are validating a page, invalidating a page, reprotecting a page to read-only, and reprotecting a page to read-write.

[8] Pages are mapped as read-only in both VASs. A write access to such a page causes a page fault. The page fault handler allocates a new page, copies the page, and maps the new page in read/write mode [BBM72].

Chapter 3

Software TLB Synchronization Algorithms

This chapter presents software TLB synchronization algorithms; Section 6.2 reviews hardware solutions and discusses their limitations. Section 3.1 first defines the semantics of consistency. Section 3.2 then gives a set of algorithms for various consistency semantics and TLB parameters (see Section 2.1.2). Finally, Section 3.3 discusses issues such as disabling interrupts, deadlocking, and batching operations together.

We focus on software algorithms that reduce the access rights to a page without causing the TLB inconsistency problem. Besides reducing access rights, the operating system may change a PTE in other ways. They are not discussed here because they do not cause nonrecoverable errors (see Section 2.4.2). Chapter 5 will further discuss whether we can tolerate TLB inconsistency even when reducing access rights.

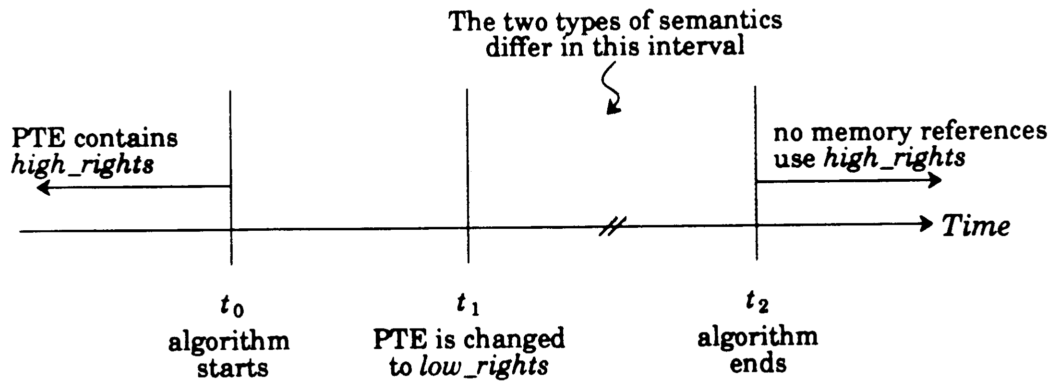
3.1. Semantics of Consistency

The algorithms in this chapter "consistently" reduce the access rights of a PTE from *high_rights* to *low_rights* (including invalidating it). Below, we assume that an algorithm starts at time t_0, reduces the access rights of the PTE at t_1, and ends at t_2.

The term "consistent" has two types of semantics: (1) *always-consistent*, and (2) *consistent-when-done*. The first one guarantees that after t_1, the page will only be accessed using *low_rights*, and the PTE will not be overwritten as a result of setting status bits. The second one provides the same guarantee, but only after t_2. Between t_1 and t_2 (the shaded area in Figure 3.1), the second one allows the page to be accessed by any processors using either *high_rights* or *low_rights*. It also allows the access rights in the PTE to be overwritten with *high_rights*, as long as the rights are changed back before t_2.

The first type of semantics is very strong, but often unnecessary. An operating system executes an algorithm as a single logical step. It considers the access rights of the page reduced only after it finishes that logical step. In other words, it can avoid taking any action on the page, *e.g.*, recycling it, before the algorithm finishes. Therefore, it is logically correct, while the algorithm is in progress, to access the page using *high_rights* by any processor even after the PTE has been changed. We still consider this type of semantics here for comparison.

In addition to consistency semantics, an algorithm is either *synchronous* or *asynchronous*. A call to a synchronous algorithm returns after it is completed,



The two types of semantics differ in this interval
PTE contains *high_rights*
no memory references use *high_rights*
Time
t_0 algorithm starts
t_1 PTE is changed to *low_rights*
t_2 algorithm ends

Figure 3.1. Consistency semantics. Between t_1 and t_2, the page can be accessed using *high_rights* under the *consistent-when-done* semantics, but cannot under the *always-consistent.*

whereas a call to an asynchronous one may return earlier. The caller of an algorithm may have to wait because the algorithm usually involves many processors. The idling time of the caller can be reduced if it returns earlier. In the following discussion, an algorithm is synchronous unless explicitly specified as asynchronous.

3.2. Algorithms for Updating a Page Table Entry Consistently

TLB synchronization algorithms vary with consistency semantics and TLB characteristics. We first summarize the algorithms, then present four basic algorithms, and finally show variants of the basic ones.

3.2.1. Overview

Updating a PTE requires two operations: (1) changing the PTE, and (2) flushing the corresponding TLB entries. The order of execution of these operations is an important issue. In general, if (1) is done first, the newly updated PTE may be overwritten (see Figure 2.2); if (2) is done first, a processor may reload the old PTE into its TLB after having flushed the TLB.

Section 3.2.2 presents four basic algorithms that differ mainly in the order of execution of the above two operations. Algorithm *PTE-first* changes the PTE first; it works only for machines that set status bits atomically. Algorithm *TLB-first* flushes the TLB entries first; it works only for machines that load TLB entries via software. Algorithm *2-phase* and *optimistic* work for machines that have neither of the above properties.

Section 3.2.3 shows variants of the basic algorithms. Algorithm *PTE-first′*, *TLB-first′*, and *optimistic′* are asynchronous versions of the basic ones. Algorithm *2-phase* has no asynchronous version. Algorithms *PTE-first″*, *TLB-first″*, *optimistic″* are for hardware that allows a processor to flush the TLB on other processors. The last three algorithms involve only one processor, so the notion of asynchrony does not exist for them.

Table 3.1 summarizes the above algorithms, and, for each one, lists the consistency semantics and the applicable hardware. Terms related to TLB's, *e.g.*, "atomically", were defined in Section 2.1.2.

All algorithms presented in this chapter assume that TLB's are initially consistent. In other words, when an algorithm starts, TLB's are not in a state that may cause nonrecoverable errors, as listed in Table 2.2. This assumption holds if the page table is always updated using the algorithms presented here. Further, we assume that absence of hardware failures, such as crashes and losses of interprocessor interrupts.

Pseudocode are used to describe algorithms. They omit low-level details, *e.g.*, mutual exclusion, to make ideas clear; some of such details are discussed in

Table 3.1. Summary of algorithms for solving TLB inconsistency.

algorithms	consistency semantics	for what types of TLB (see Section 2.1.2)
PTE-first	consistent when done	status bits set atomically
TLB-first	always consistent	TLB's loaded by software
2-phase	always consistent	any
optimistic	consistent when done	any
PTE-first′	consistent when done asynchronous	status bits set atomically
TLB-first′	always consistent asynchronous	TLB's loaded by software
optimistic′	consistent when done asynchronous	any
PTE-first″	consistent when done	TLB's flushed by other processors, and status bits set atomically
TLB-first″	always consistent	TLB's flushed by other processors, and TLB's loaded by software
optimistic″	consistent when done	TLB's flushed by other processors

Section 3.3. Also, the pseudocodes use the following notations:

PTE :	The PTE to be updated.
new_PTE :	The target value to be written to *PTE* .
old_PTE :	The old value of *PTE* .
hot_TLBEs :	TLB entries corresponding to *PTE* .
requester :	The processor that starts the algorithm. Usually, it sends interprocessor requests to other processors.
replier :	A processor that responds to an interprocessor request.
target_set	The set of processors that contain *hot_TLBEs* .

3.2.2. Basic algorithms

Algorithm *PTE-first* updates the PTE first, then flushes the corresponding TLB entries. It works only if the newly updated PTE can never be overwritten as a result of setting status bits (see Section 2.2 and Figure 2.2). This condition holds if the status bits are set atomically. This algorithm provides the *consistent-when-done* semantics, because TLB entries may still contain *high_rights* after the PTE has been changed.

```
/* PTE-first */
Requester()
{
        PTE = new_PTE;
        if (requester in target_set)
                flush hot_TLBEs locally;
        for each processor in (target_set - requester)
                send an interprocessor request;
                /* see Section 3.3.2 */
        for each processor in (target_set - requester)
                wait for an acknowledgement;
}

Replier()
{
        flush hot_TLBEs locally;
        acknowledge;
}
```

Algorithm *TLB-first* flushes TLB entries before updating the PTE. It works only if TLB entries are loaded via software traps. Otherwise, the hardware may load *old_PTE* into a TLB after the algorithm has flushed it. It provides the *always-consistent* semantics, because the PTE is changed after all old TLB entries have been flushed.

```
/* TLB-first */
Requester()
```

```
{
     changing_PTE_flag = true;
     tentative_PTE = new_PTE;
     if (requester in target_set)
          flush hot_TLBEs locally;
     for each processor in (target_set - requester)
          send an interprocessor request;
          /* see Section 3.3.2 */
     for each processor in (target_set - requester)
          wait for an acknowledgement;
     PTE = new_PTE;
     changing_PTE_flag = false;
}

Replier()
{
     flush hot_TLBEs locally;
     acknowledge;
}

Load_TLB_trap_handler()
{
     if (changing_PTE_flag)
          load tentative_PTE into TLB;
     else
          load PTE into TLB;
}
```

Algorithm *2-phase* works for any combination of TLB parameters, and supports the *always-consistent* semantics. This algorithm is used in Mach [BRG89]; Figure 3.2 illustrates it. To ensure that the newly updated PTE will not be overwritten as a result of setting status bits, the algorithm stalls all processor that may access the page. In the first phase, the algorithm flushes all *hot_TLBEs*, and stops all active processors except the requester itself. In the second phase, it updates the PTE and then resumes all stopped processors. Both the requester and replier have to block (see the shaded intervals in Figure 3.2). Furthermore, they have to wait for the slowest processor. The blocking time could be long, especially when there is a replier that disables interrupts.

```
/* 2-phase */
Requester()
{
     if (requester in target_set)
          flush hot_TLBEs locally;
     for each processor in (target_set - requester)
          send an interprocessor request;
          /* see Section 3.3.2 */
```

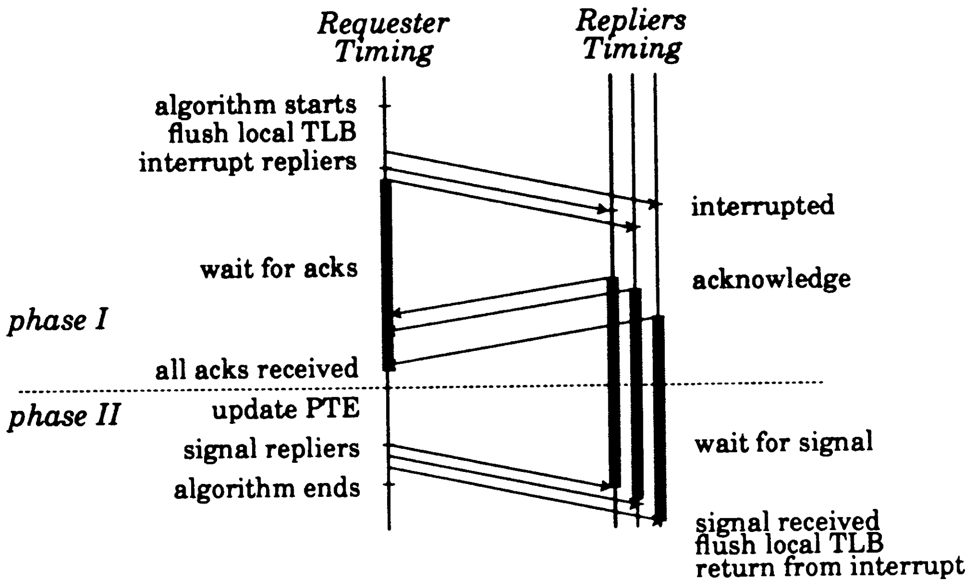


Figure 3.2. The 2-phase algorithm. Processors block in the shaded intervals. Every processor must wait for the slowest replier.

```
     for each processor in (target_set - starter)
          wait for an acknowledgement;
     PTE = new_PTE;
     for each processor in (target_set - starter)
          send a continuing signal;
}

Replier()
{
     acknowledge;
     wait for the continuing signal;
     flush hot_TLBEs locally;
}
```

Algorithm *optimistic* (Figure 3.3) is intended to avoid the blocking of repliers. It proceeds in the same way as algorithm *PTE-first*, even when status bits are not set atomically. Therefore, the newly updated PTE may be overwritten as a result of setting status bits. However, the algorithm can detect this and, when necessary, repeat the updating. It provides the *consistent-when-done* semantics, which allows the PTE to be overwritten while the algorithm is in progress.

The idea of this algorithm is similar to that of optimistic currency control in database systems [BeG81, BHG87]. It does not employ a complex mechanism, as in algorithm *2-phase*, for a problem that happens only rarely. Instead, it proceeds as if the problem does not exist, and fixes it when it does happen. Its performance is close to that of algorithm *PTE-first* if the probability that a PTE is overwritten is low.

The requester code shown below contains a loop, which may make the worst-case time unbounded. This loop can be easily removed, if desirable, by executing the 2-phase algorithm when the test fails. However, this has little effect on overall performance if the test fails rarely.

```
/* optimistic */
Requester()
{
```

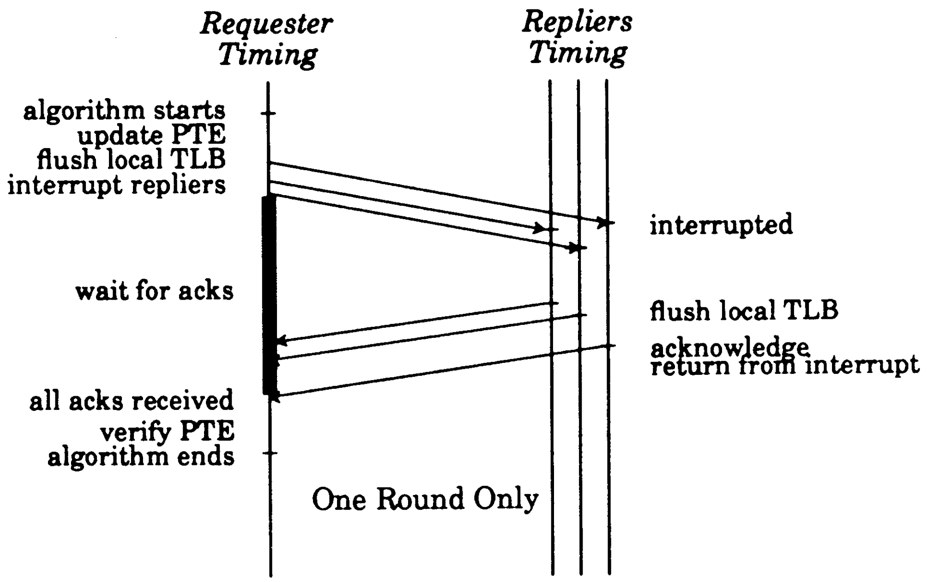


Figure 3.3. The optimistic algorithm. Only the requester blocks in the shaded interval.

```
    do {
         PTE = new_PTE;
         if (requester in target_set)
              flush hot_TLBEs locally;
         for each processor in (target_set - requester)
              send an interprocessor request;
              /* see Section 3.3.2 */
         for each processor in (target_set - requester)
              wait for an acknowledgement;
    } while (PTE.access_rights != new_PTE.access_rights);
}

Repliers()
{
     flush hot_TLBEs locally;
     acknowledge;
}
```

3.2.3. Variants of basic algorithms

Algorithm *PTE-first'*, *TLB-first'*, and *optimistic'* are asynchronous versions of the corresponding basic algorithms. Algorithm *2-phase* has no asynchronous version because the request cannot update the PTE until all repliers have acknowledged. Since the modifications are similar for the three algorithms, only algorithm *optimistic'* is shown here. The requester initializes a counter to the number of repliers. Each replier, after flushing its TLB, decreases the counter by one. The last replier signals the end of the algorithm.

```
/* optimistic' */
Requester()
{
     PTE = new_PTE;
     if (requester in target_set)
          flush hot_TLBEs locally;
     count = sizeof (target_set - requester);
     for each processor in (target_set - requester)
          send an interprocessor request;
          /* see Section 3.3.2 */
     /* do not wait here */
}

Repliers()
{
     flush hot_TLBEs locally;
     count = count - 1;
     if (count == 0)
          if (PTE.access_rights != new_PTE.access_rights)
               Requester();    /* repeat the algorithm */
          else
```

```
                    signal end of algorithm;
    }
```

Algorithm *PTE-first″*, *TLB-first″*, and *optimistic″* are for hardware that allows a processor to flush the TLBs of other processors. These algorithms can be completed solely by one processor, so interprocessor requests are not necessary. Algorithm *2-phase* cannot take advantage of this hardware feature because it requires that all active processors be stopped while the PTE is being updated. Again, we only show *optimistic″* here for the sake of brevity.

```
/* optimistic″ */
Requester()
{
        do {
                PTE = new_PTE;
                for each processor in (target_set)
                        flush hot_TLBEs on that processor.
        } while (PTE.access_rights != new_PTE.access_rights);
}

/* no replier code */
```

3.3. Discussion

The section discusses some details that we omitted in presenting the above algorithms.

3.3.1. Disabling interrupts

Interrupts, when not disabled, may affect the correctness of a TLB synchronization algorithm. An interrupt handler may access the target page in the middle of a TLB synchronization algorithm. This may affect the PTE and the corresponding TLB entries, which are either being updated or being flushed. Below, we will show that interrupts, if not disabled, affect the correctness only of the *2-phase* algorithm among the ten algorithms described above. In addition to correctness, handling an interrupt slows down a processor, possibly delaying every processor involved in a TLB synchronization algorithm.

The key idea of algorithm *2-phase* is to stop accessing the target page on all processors until the TLB entries have been flushed on that processor, and the PTE has been updated. Therefore, the algorithm must disable any interrupt that may cause the target page to be accessed.

Algorithm *PTE-first* and its variants are used when status bits are set atomically. Therefore, accessing the target page cannot destroy the newly updated PTE. Furthermore, accessing the page may load only *new_PTE* into a TLB because the PTE has been updated at the beginning. Hence, interrupts are not a problem for them.

Algorithm *TLB-first* and its variants are used when TLB entries are loaded by software. Hence accessing the target page may load only *new_PTE* into a TLB. The PTE is updated when no TLB's contains *old_PTE*, so the new PTE cannot be overwritten by *old_PTE* due to accessing the target page. Again, accessing the target page in the middle of these algorithms causes no problems.

In algorithm *optimistic* and its variants, servicing an interrupt may overwrite the newly updated PTE, or load *old_PTE* into a TLB if the PTE has already been overwritten. However, these do not violate the *consistent-when-done* semantics, and can be detected and fixed before the algorithm terminates.

3.3.2. Interprocessor requests and batching

An interprocessor request is carried out in two steps: preparing the request in main memory, and optionally notifying the replier. The first step is typically done by enqueueing the request to a per-processor queue. The second step is done by issuing an interprocessor interrupt. Acknowledging a request is easier. The replier sets a shared variable, and the requester spins on it.

An interprocessor interrupt is necessary only when the response time of the request is important, for example, when the requester blocks for the acknowledgement of the request. It is potentially expensive because the interrupt may not be handled immediately if the replier has interrupts masked. This will make the requester wait longer for an acknowledgement, and increase the total overhead.

On the other hand, a processor does not have to be notified for every request. It can check its request queue and execute the requests at a time that is convenient to it, such as on a context switch or on a clock interrupt. This saves the overhead of servicing an interrupt for each request. Additionally, this has the effect of batching requests together, and can further reduce the average overhead per request. Algorithm *PTE-first'*, *TLB-first'*, and *optimistic'*, which are already asynchronous, may take advantage of this feature.

Some machines, such as MIPS [MMM86, TBJ88], allow the TLB to contain entries for multiple virtual address spaces (VAS) at the same time. However, a TLB entry may never be used until the the processor switches to the VAS containing it. Therefore, all algorithms that issue interprocessor requests should be modified for these machines as follow:

(1) Still enqueue an interprocessor request to every processor in *target_set*.

(2) Interrupt only processors running in the VAS that contains *PTE*.

(3) Before switching to a VAS, finish processing all requests for that VAS.

3.3.3. Deadlocking

Deadlocking is a problem only for algorithm *2-phase*, which must disable interrupts during the algorithm. In this algorithm, the requester sends an interprocessor request to all repliers and waits for an acknowledgement from each one. Since the algorithm disables interrupts, deadlocks may happen. For example, a deadlock happens when two processors both enter the requester code, disable interrupts, interrupt the other one, and wait for an acknowledgement.

The deadlocking problem can be solved by not waiting for the acknowledgement of a request, but waiting for the replier to start executing the algorithm (possibly due to another request). The requests for a processor are put in a queue in main memory, and the processor will finish all requests before leaving the algorithm. The problem can also be solved by checking the request queue and servicing requests while waiting for acknowledgements. Black *et al.* and Rosenburg discussed more on the deadlocking problem [BRG89, Ros89].

Chapter 4

The Performance of TLB Synchronization Algorithms

This chapter evaluates the performance of the TLB synchronization algorithms described in Chapter 3. The goals are (1) to compare the performance of different algorithms, and (2) to see whether the performance of these algorithms scales well with the number of processors and with the rate of TLB synchronization. Section 4.1 describes the performance model, assumptions, parameters, and measures. Section 4.2 analyzes the algorithms using an iterative method. Section 4.3 verifies the analysis with simulation. Section 4.4 shows the important performance results. Section 4.5 concludes this chapter and suggests the need for tolerating TLB inconsistency, which is the theme of Chapter 5.

4.1. The Performance Model

The model we use for performance analysis (Section 4.2) and simulation (Section 4.3) consists of N identical processors running independently except for the interactions due to TLB synchronization. Each one services three classes of events:

(1) *requester events*, which initiate new TLB synchronization operations,

(2) *replier events*, which respond to TLB synchronization operations initiated by other processors, and

(3) *interrupt-disabling events*, which disable interrupts for reasons other than TLB synchronization.

We consider a processor idle when it is not executing these events. Regular jobs, *e.g.*, user programs, are preempted by these three classes of events[1], and are not of interest here.

4.1.1. Event arrival and execution

Requester events are generated by a processor when it remaps pages. We assume that a processor does not remap pages while it is busy with TLB synchronization, or while it is servicing an interrupt-disabling event. Therefore, each processor is a closed system in terms of requester events. This assumption is realistic, particularly when the system is saturated, but complicates the

[1] Except for asynchronous algorithms; see Section 4.2.3

analysis[2].

Replier events are triggered by requester events executed on other processors. Each TLB synchronization operation involves M processors, where M is a constant. A requester randomly selects $M-1$ repliers and causes for each one a replier event. Replier events are delivered to repliers by interprocessor interrupts. The interaction between requesters and repliers varies with the algorithms to be evaluated.

Interrupt-disabling events correspond to clock interrupts, I/O interrupts, traps, critical sections, etc. We assume that their arrival is independent of the state of the system. In other words, interrupt-disabling events (*e.g.*, a clock interrupt) may arrive at a processor even while it is busy with TLB synchronization.

The service of events is nonpreemptive. If an event cannot be serviced immediately, it will be queued and serviced eventually. Events in the same class are serviced in FIFO order. Events in different classes are serviced on a priority basis. Both requester events or replier events have a higher priority than interrupt-disabling events. Since a requester event may never arrive when there are replier events pending (because of the closed system assumption described above), differences in priority between requester events and replier event are unimportant.

4.1.2. Interarrival and service times

The interarrival times and service times of events are assumed to be independent random variables. For tractability, we also assume that all of them are exponentially distributed. The following list summarizes their mean values:

$\frac{1}{\lambda_t}$:	mean time for each processor to generate a requester event
$\frac{1}{\lambda_d}$:	mean interarrival time of interrupt-disabling events on each processor
$X_{req_1}+(M-1)X_{req_2}$:	mean amount of computation of requester events (excluding blocking time)
X_{reply}:	mean amount of computation of replier events (excluding blocking time)
X_d:	mean service time of interrupt-disabling events
X_i:	mean overhead to dispatch an interrupt

The mean service time of requester events has multiple components and depends on $M-1$. For simplicity, we still assume that the service time as a whole is exponentially distributed, though this may not be true in practice. Further, for

[2] Contrarily, others often assume independent request arrivals in the analysis of synchronization and updating of replicated data [Gar81, Lee80].

convenience, ρ_d denotes $\lambda_d X_d$.

4.1.3. Algorithm-specific parameters

There are two algorithm-specific assumptions. For optimistic algorithms, during each round of the algorithm, the probability that a processor overwrites a PTE as a result of setting the dirty or referenced bit is p_0. For asynchronous algorithms, replier events do not preempt user programs. Instead, a processor checks its queue of replier events after executing user programs for every time slice of X_j seconds. (X_j is a constant in this model.)

4.1.4. Performance measures

From the solution of the model, we will obtain the following performance measures.

(1) *CPU overhead*: the mean percentage of total CPU time spent in TLB synchronization.

(2) *Latency*: the mean delay between the start and the completion of a TLB synchronization algorithm.

(3) *Throughput*: the mean number of TLB synchronization operations completed per processor per second.

(4) *Space overhead*: the mean number of pages frozen per processor due to TLB synchronization. A physical page cannot be reused for other purposes before all TLB synchronization operations involving it are completed.

4.2. Performance Analysis

This section analyzes three TLB synchronization algorithms: the 2-phase algorithms, the optimistic-synch algorithm, and the optimistic-asynch algorithm. The first two ones are the same as those described in Section 3.2.2. The third one differs from the optimistic-synch algorithm in two ways: (1) the requester does not block, and (2) interprocessor requests are batched and checked in the background (see Section 3.3.2).

The performance of other algorithms listed in Chapter 3 can be analyzed using similar methods, or can be estimated using the performance results obtained for these three algorithms. For example, by setting p_0 to zero, the optimistic-synch algorithm reduces to an one-round algorithm, and its performance is similar to that of the PTE-first or TLB-first algorithm. The optimistic-asynch algorithm does not involve interprocessor interrupts. By further setting X_{reply} to zero, its CPU overhead is similar to that of the PTE-first'' or TLB-first'' algorithm.

4.2.1. The 2-phase algorithm

We use an iterative method to obtain approximate performance results of the 2-phase algorithm. We do not analyze the model as a queueing network, because in this algorithm processors block on each other, making a product form queueing network solution difficult. Section 4.2.1.1 outlines our method. Section 4.2.1.2 and Section 4.2.1.3 derive equations. Section 4.2.1.4 summarizes the method by giving a step-by-step procedure.

4.2.1.1. Method of analysis

Our technique for analyzing the performance of the 2-phase algorithm is iterative. In each iteration, we assume that the algorithm's CPU overhead, ρ_t, is known. Using ρ_t, we derive some performance measures, such as the acknowledgement time of an operation, the synchronization time of an operation, and the duration of busy periods. We then recompute the CPU overhead, ρ_t'. This process is repeated until the difference between ρ_t' and ρ_t is negligible.

For tractability, we make two simplifications during the calculations. First, we assume that different processors acknowledge a TLB synchronization request independently. Second, although we adjust the arrival rates of requester and replier events using ρ_t and ρ_d (see the closed system assumption in Section 4.1.1), we assume that the arrivals are independent processes. Furthermore, for one performance measure, we only calculate its bounds, because its exact solution is too complicated to find. Therefore, the results of this analysis are approximate bounds. In normal operating conditions, these bounds are close to each other, and fairly close also to simulation results (Section 4.3).

4.2.1.2. Acknowledgement and synchronization time

In the synchronization phase, the requesting processor interrupts $M-1$ replying processors, and waits until all of them have entered the 2-phase algorithm, *i.e.*, have stopped accessing the target page. We assume that a replying processor sets a flag to indicate that it is executing the 2-phase algorithm immediately after the interprocessor interrupt has been dispatched. We define *acknowledgement time* as the time it takes for a replying processor to turn on this flag after being interrupted. For each replying processor, the acknowledgement time depends on its state when it is interrupted. Possible states are:

($S1$) the processor is already executing the 2-phase algorithm due to other TLB operations,

($S2$) the processor is not executing the 2-phase algorithm, and interprocessor interrupts are enabled, and

($S3$) the processor is in the middle of an interrupt disabling event.

Let $ack(t)$ denote the pdf (probability density function) of acknowledgment time, and $ack(t \mid C)$ denote the pdf of acknowledgment time under condition C. For state $S1$, the acknowledgement time is zero, so $ack(t \mid S1) = \delta(t)$, a unit impulse function at $t=0$. For state $S2$, the acknowledgement time is X_i, the fixed overhead of dispatching an interprocessor interrupt. Hence, $ack(t \mid S2) = \delta(t-X_i)$. For state $S3$, the acknowledgement time is X_i plus the residual execution time of an interrupt disabling event. The execution time of interrupt disabling events is exponentially distributed with mean X_d. Since this distribution is memoryless, the residual execution has the same distribution, *i.e.*, $ack(t \mid S3) = \frac{1}{X_d} e^{-(t-X_i)/X_d} u(t-X_i)$, where $u(t-X_i)$ is a unit step function at X_i. Combining the three $ack(t \mid C)$'s, we have

$$ack(t) = \text{Prob}[S1]\, ack(t \mid S1) + \text{Prob}[S2]\, ack(t \mid S2) + \text{Prob}[S3]\, ack(t \mid S3)$$

$$= \rho_t \delta(t) + (1 - \rho_d - \rho_t)\delta(t-X_i) + \rho_d \frac{1}{X_d} e^{-(t-X_i)/X_d} u(t-X_i)$$

$$= \begin{cases} \rho_t\ \delta(0) & \text{if } t=0 \\ 0 & \text{if } 0<t<X_i \\ (1-\rho_d-\rho_t)\ \delta(0) & \text{if } t=X_i \\ \lambda_d e^{-(t-X_i)/X_d} & \text{if } t>X_i \end{cases} \qquad (4.1)$$

Integrating Eq. (4.1) from 0 to t, we have the PDF (probability distribution function) of acknowledgement time,

$$ACK(t) = \rho_t u(t) + (1 - \rho_t - \rho_d e^{-(t-X_i)/X_d})\, u(t-X_i)$$

$$= \begin{cases} \rho_t & \text{if } 0 \le t < X_i \\ 1 - \rho_d e^{-(t-X_i)/X_d} & \text{if } t \ge X_i \end{cases} \qquad (4.2)$$

We define *synchronization time*, $\mathbf{s}$, the time it takes for all replying processors to acknowledge. In other words, $\mathbf{s} = \max(\mathbf{a}_i\,; i=1,\ldots,M-1)$, where $\mathbf{a}_i$ is the acknowledgement time of the ith replying processor. Let $s(t)$ and $S(t)$ denote the pdf and PDF of $\mathbf{s}$ respectively. In Section 4.1, we assumed that processors run independently, except for the interaction due to TLB synchronization. Hence it is reasonable to treat the $\mathbf{a}_i$'s as independent in calculating $S(t)$:

$$S(t) = \text{Prob}[max(\mathbf{a}_i\,; i=1,\ldots, M-1) \le t]$$

$$= \text{Prob}[\mathbf{a}_1 \le t]\, \text{Prob}[\mathbf{a}_2 \le t] \cdots \text{Prob}[\mathbf{a}_{M-1} \le t]$$

$$= [ACK(t)]^{M-1} \qquad (4.3)$$

Except for the two discontinuous points at $t=0$ and $t=X_i$, we have

$$s(t) = \frac{d}{dt} S(t) = (M-1)[ACK(t)]^{M-2} ack(t). \qquad (4.4)$$

$s(t)$ at $t=0$ and $t=X_i$ are calculated as follows. The synchronization time is zero only when all replying processors are already in the 2-phase algorithm. The probability of this is ρ_t^{M-1}. The synchronization time is X_i when no replying processor is executing an interrupt disabling event, and at least one replying processor is not executing the 2-phase algorithm. The probability of this is $(1-\rho_d)^{M-1} - \rho_t^{M-1}$. Therefore we have

$$s(t) = \begin{cases} \rho_t^{M-1}\,\delta(0) & \text{if } t = 0 \\ 0 & \text{if } 0 < t < X_i \\ [(1-\rho_d)^{M-1} - \rho_t^{M-1}]\,\delta(0) & \text{if } t = X_i \\ (M-1)\,\lambda_d\,(1-\rho_d e^{-(t-X_i)/X_d})^{M-2}\, e^{-(t-X_i)/X_d} & \text{if } t > X_i \end{cases} \tag{4.5}$$

The mean synchronization time, $SYNC$, is obtained by integrating $s(t)t$,

$$SYNC = \text{Prob}[\mathbf{s} = X_i]\,X_i + \int_{X_i}^{\infty} (M-1)[ACK(t)]^{M-2} ack(t)\, t\; dt$$

$$= X_i\,[(1-\rho_d)^{M-1} - \rho_t^{M-1}] \; +$$

$$(M-1)\,\rho_d X_d \sum_{i=0}^{M-2} \begin{bmatrix} M-2 \\ i \end{bmatrix} (-\rho_d)^i \, \frac{1+(i+1)X_i/X_d}{(i+1)^2} \tag{4.6}$$

4.2.1.3. Busy periods

This section uses mean-value analysis to calculate the duration of *busy periods*, which will be used to compute the latency of a TLB operation, and the CPU overhead of the algorithm. A processor is said to be *busy* when it is executing the 2-phase algorithm. A busy period is a continuous period during which a processor is busy. It may consist of several TLB operations in a row, because replier events may still come in while a processor is busy.

We divide busy periods into two types according to the two classes of TLB synchronization events described in Section 4.1. A *requester busy period* is a busy period starting with a requester event; a *replier busy period* is a period starting with a replier event.

The following notation is used in the calculations. λ_{req} and λ_{reply} are respectively the mean number of requester and replier events per processor per second. T_{req} is the mean duration of requester events. T_{reply} is the mean duration of replier events that initiate a replier busy period; whereas T'_{reply} is the duration of replier events that arrive in the middle of a busy period. B_{req} and B_{reply} are the mean duration of requester and replier busy periods respectively. N_{req} is the mean number of replier events executed during B_{req}. N_{reply} is the mean number of replier events executed during B_{reply} other than the one that starts the period.

Section 4.1.2 assumes that the mean amounts of computation of a requester and a replier event are X_{req_1}+$(M-1)X_{req_2}$ and X_{reply} respectively. The execution of an event is divided into several stages by interprocessor interactions, such as interrupts and acknowledgements (see Figure 3.2). For tractability, in our analysis, we assume that all the computation of an event is carried out in a single stage, as showned in Figure 4.1.

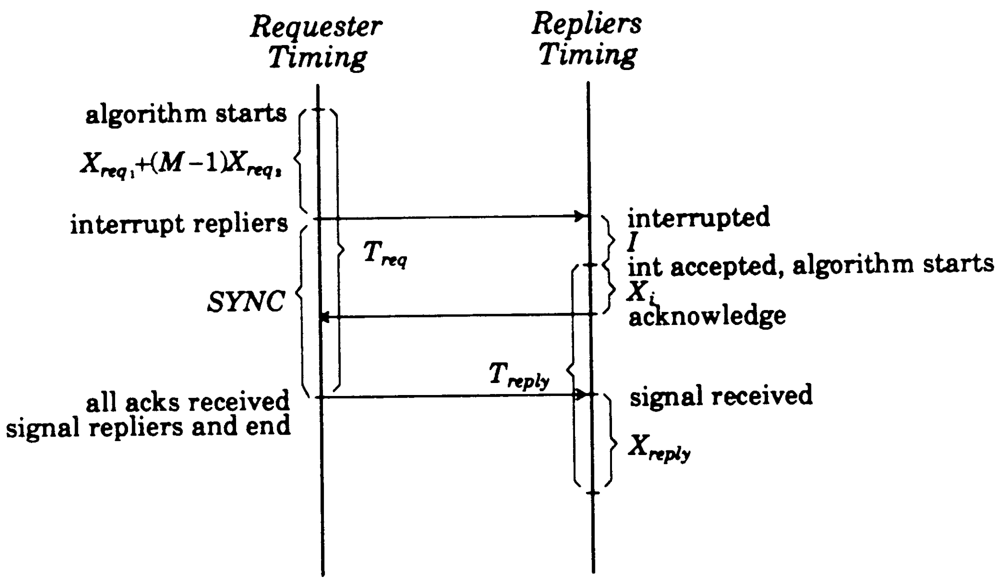


Figure 4.1. A simplified timing diagram of the 2-phase algorithm. I is the initial delay due to other interrupt-disabling events; X_i is the overhead of dispatching an interrupt.

By assumption (Section 4.1), a processor generates TLB requests only when it is not busy and not executing an interrupt disabling event. Hence

$$\lambda_{req} = (1 - \rho_d - \rho_t)\,\lambda_t \tag{4.7}$$

There are N processors generating requester events, and each requester event causes $M-1$ replier events among N processors. Thus

$$\lambda_{reply} = N\,\lambda_{req}\,\frac{M-1}{N} = \lambda_{req}\,(M-1) \tag{4.8}$$

By definition,

$$B_{req} = T_{req} + N_{req}\,T'_{reply} \tag{4.9}$$

T_{req} is the mean amount of computation of a requester event plus synchronization time,

$$T_{req} = X_{req_1} + (M-1)X_{req_2} + SYNC \tag{4.10}$$

As stated in Section 4.2.1.1, we assume that the arrival of replier events is an independent process for tractability. This is not generally true for a closed system, but is a good approximation when the mean interarrival time is much larger than the mean service time. Under this assumption, the mean number of replier events arrived during a period is the mean duration of that period times the arrival rate. So we have

$$N_{req} = \lambda_{reply} B_{req} \tag{4.11}$$

Substituting Eq. (4.11) into (4.9), we have

$$B_{req} = \frac{T_{req}}{1 - \lambda_{reply} T'_{reply}} \tag{4.12}$$

Again by definition,

$$B_{reply} = T_{reply} + N_{reply} T'_{reply} \tag{4.13}$$

The calculations for T_{reply} and N_{reply} are slightly different from that for T_{req} and N_{req}, mainly because of the initial delay due to disabling interrupts. For a replier event that starts a busy period, the mean initial delay equals the probability that a processor is in an interrupt disabling event times the mean residual time of that event:

$$I = \rho_d X_d \tag{4.14}$$

As shown in Figure 4.1, the replier is not executing the 2-phase algorithm during the initial delay. Hence, this delay is not charged to T_{reply}, and we have

$$T_{reply} = SYNC - I + X_{reply} \tag{4.15}$$

On the other hand, all replier events that arrive during the initial delay I are included in the busy period. Similarly to Eq. (4.11), we have

$$N_{reply} = (I + B_{reply}) \lambda_{reply} \tag{4.16}$$

Substituting Eq. (4.16) into (4.13), we have

$$B_{reply} = \frac{T_{reply} + I \lambda_{reply} T'_{reply}}{1 - \lambda_{reply} T'_{reply}} \tag{4.17}$$

We now recompute the CPU overhead. For each processor, there are λ_{req} requester events and λ_{reply} replier events per second. The number of requester busy periods per second is simply λ_{req}. The number of replier busy periods per second, on the other hand, is less than λ_{reply}. This is because $\lambda_{req} N_{req}$ replier events are executed in requester busy periods, and each replier busy periods has $N_{reply} + 1$ replier events (the extra one is for the event that starts the busy period). Therefore,

$$\begin{aligned} \rho_t' &= B_{req} \lambda_{req} + B_{reply} \frac{\lambda_{reply} - \lambda_{req} N_{req}}{1 + N_{reply}} \\ &= B_{req} \lambda_{req} + B_{reply} \lambda_{reply} \frac{1 - \lambda_{req} B_{req}}{1 + \lambda_{reply} (B_{reply} + I)} \end{aligned} \tag{4.18}$$

So far the only unknown is T'_{reply}, whose upper and lower bounds are estimated below. Its exact solution is difficult to find because of the concurrent nature of the 2-phase algorithm. In this algorithm, a processor may scan the replier event queue while waiting for synchronization. Therefore, a processor may effectively handle multiple TLB operations in parallel (although a processor cannot process for multiple operations at the same time, it can "wait" for them concurrently). In the best case, a replier event may be completely overlapped with the synchronization time of the event that starts a busy period. In the worst case, all events of a busy period are serialized, *i.e.*, one arrives right at the end of the previous one. Hence the lower and upper bounds of T'_{reply} are

$$0 \leq T'_{reply} \leq X_{reply} + SYNC \tag{4.19}$$

T'_{reply} appears in the form of $1 - \lambda_{reply} T'_{reply}$ in Eq. (4.12) and Eq. (4.17). Its effect is insignificant if $\lambda_{reply}(X_{reply}+SYNC)$ is much smaller than 1. $\lambda_{reply}(X_{reply}+SYNC)$, as a rough approximation, is the CPU overhead for handling replier events, and should be low for the system to be usable. In other words, T'_{reply} has a significant effect only when the system is saturated, in which case a precise performance measure is less important.

4.2.1.4. The complete procedure

The following procedure summarizes the analysis in Sections 4.2.1.2 and 4.2.1.3:

P1. Assign an initial value, say, 0.1, to ρ_t.

P2. Compute *SYNC* by Eq. (4.6).

P3. Compute λ_{req} and λ_{reply} by Eqs. (4.7) and (4.8); compute I by Eq. (4.14); compute T_{req} and T_{reply} by Eqs. (4.10) and (4.15).

P4. Compute B_{req} and B_{reply} by Eqs. (4.12) and (4.17).

P5. Compute ρ_t' by Eq. (4.18). If $\rho_t' \geq 1$ or $\rho_t' \leq 0$, output "fail" and stop. If $| \rho_t' - \rho_t | \leq \varepsilon$, output "succeed" and stop.

P6. Assign ρ_t' to ρ_t, and go to P2.

We perform this procedure twice, once with $T'_{reply} = 0$ for lower bounds, and once with $T'_{reply} = X_{reply}+SYNC$ for upper bounds. This procedure terminates rapidly, usually in less than 20 iterations. It converges in most cases. It fails only when the given parameters represent an extremely high load. Under such conditions, overestimating T'_{reply} may cause the computed ρ_t' to become greater than 1.

The performance measures listed in Section 4.1.4 can be immediately obtained from the results of this procedure. The per-processor CPU overhead is ρ_t. The per-processor throughput of TLB operations is λ_{req}. The latency of a TLB request is B_{req} (not T_{req} because a call to the algorithm will not return until the busy period ends). The per-processor space overhead is, by Little's formula [Kle75], the arrival rate of requester events times the mean duration of a requester event, or $\lambda_{req} T_{req}$ pages.

4.2.2. The optimistic-synch algorithm

The technique for analyzing the performance of the optimistic-synch algorithm is similar to that used in Section 4.2.1, except for the calculations of arrival rates and acknowledgement time. Here, we consider the effect of failure (*i.e.*, p_0, defined in Section 4.1.3) in calculating arrival rates, and use a nonpreemptive queueing model in calculating acknowledgement time.

The simplifications for tractability are also similar. We obtain approximate results by assuming that (1) processors acknowledge independently, and (2) TLB events obey Poisson arrival processes, though their arrival rates are adjusted using ρ_t, ρ_d, p_0, etc.

Finally, we use the same the terminology and notation as in Section 4.2.1, unless otherwise stated.

4.2.2.1. Arrival rates

In the optimistic-synch algorithm, we break a TLB operation into a sequence of *rounds*, each of which is a request-and-reply-by-all cycle as in the 2-phase algorithm. We call each round of a requester event a *primitive requester event*, and denote its duration as T^0_{req} and its per-processor arrival rate as λ^0_{req}.

A TLB operation may need more than one round because the algorithm may fail at the end of a round. The probability that a processor overwrites a PTE as a result of setting the dirty or referenced bit during each round is p_0. For each TLB operation, $M-1$ other processors have the target page mapped and may overwrite the target PTE (the one that initiates the operation never overwrites the PTE). Therefore, the probability that a TLB operation fails at the end of a round is

$$p = 1 - (1-p_0)^{M-1} \tag{4.20}$$

The mean number of rounds per TLB operation is

$$1 + p + p^2 + p^3 + \cdots = \frac{1}{1-p} = \frac{1}{(1-p_0)^{M-1}}$$

Eq. (4.7) is still valid for requester events. Dividing it by $(1-p_0)^{M-1}$, we obtain the arrival rate of primitive requester events:

$$\lambda^0_{req} = \frac{\lambda_{req}}{(1-p_0)^{M-1}} = (1-\rho_d-\rho_t)\frac{\lambda_t}{(1-p_0)^{M-1}} \tag{4.21}$$

Similarly to Eq. (4.8), each round of a TLB operation generates $M-1$ replier events. Hence

$$\lambda_{reply} = \lambda^0_{req}\,(M-1) = (1-\rho_d-\rho_t)\,\lambda_t\frac{M-1}{(1-p_0)^{M-1}} \tag{4.22}$$

4.2.2.2. Acknowledgement and synchronization time

An acknowledgment serves a different purpose in the optimistic-synch algorithm than in the 2-phase algorithm. In the 2-phase algorithm, acknowledgements guarantee that repliers have stopped using the page being updated so that the requester can safely change the page table entry. In the optimistic-synch algorithm, acknowledgements indicate that repliers have completed their work so that the requester can check whether the operation fails. Therefore, the definition of acknowledgement time is different for the two algorithms. For the optimistic-synch algorithm we define *acknowledgement time* as the time it takes for a replier to finish a replier event after being interrupted for that event. *Synchronization time* is still the maximum of $M-1$ acknowledgement times.

Several factors delay the acknowledgement of a replier event:

(1) The interrupt disabling event being executed, if any, must be finished.

(2) The primitive requester event being processed, if any, must be executed until it enters the synchronization stage. Before this stage, the requester will not check incoming replier events.

(3) All replier events that have arrived before this event must be finished.

Note that, in (2), the waiting time of a requester event does not affect the acknowledgement time, because a processor can serve other replier events while waiting for synchronization. This feature simplifies the analysis of acknowledgement time greatly. First, we need not deal with "waiting in parallel" (see Section 4.2.1.3) because the waiting time is not counted at all. Second, we do not need the total execution time of a requester event in order to calculate the delay caused by it.

We represent this problem by a nonpreemptive queueing model. In the model, each processor serves customers belonging to 3 priority groups, indexed by $p=1,2,3$, where 3 is the highest priority. Replier events belong to priority group 3, the highest one. Interrupt disabling events belong to priority group 1. Customers of priority 2 are the first half of primitive requester events, *i.e.*, request events without synchronization. The queueing discipline is nonpreemptive priority, with FCFS service within a priority group. This queueing discipline effectively takes into account the three factors listed above, *i.e.*, a customer of priority group 3 must wait for the completion of any lower-priority customer that is already in service, and for the completion of any same-priority customers that arrived before it. Therefore, the acknowledgement time we want to find is the priority-3 group's time in the system in this model.

Kleinrock gives the general solution for a P-priority system [Kle76]. He assumes that, for each group p, the arrival process is Poisson with mean rate λ_p; the mean service time is $\bar{x}_p$; the Laplace transform of the service time is $B_p^*(s)$. The Laplace transform of the priority-p group's waiting time is

$$W_p^*(s) = \frac{(1-\rho)[s+\lambda_H-\lambda_H G_H^*(s)]+\lambda_L[1-B_L^*(s+\lambda_H-\lambda_H G_H^*(s))]}{s-\lambda_p+\lambda_p B_p^*(s+\lambda_H-\lambda_H G_H^*(s))} \qquad (4.23)$$

where

$$\rho = \sum_{i=1}^{P} \lambda_i \bar{x}_i$$

$$\lambda_H = \sum_{i=p+1}^{P} \lambda_i$$

$$\lambda_L = \sum_{i=1}^{p-1} \lambda_i$$

$$B_H^*(s) = \sum_{i=p+1}^{P} \frac{\lambda_i}{\lambda_H} B_i^*(s)$$

$$B_L^*(s) = \sum_{i=1}^{p-1} \frac{\lambda_i}{\lambda_L} B_i^*(s)$$

$$G_H^* = B_H^*(s+\lambda_H-\lambda_H G_H^*(s))$$

And the Laplace transform for the priority-p group's system time is

$$S_p^*(s) = W_p^*(s) B_p^*(s) \qquad (4.24)$$

In order to apply Eqs. (4.23) and (4.24), we assume in the calculation that the arrivals of primitive requester events and replier events are Poisson processes with the rates given in Eqs. (4.21) and (4.22). This assumption is not completely true, but is a good approximation of reality, as we will see in Section 4.3.2. Let μ_p denote the service rate of priority-p group. Then the parameters of the model are:

$$\begin{cases} \lambda_1 = \lambda_d & \mu_1 = 1 / X_d \\ \lambda_2 = (1-\rho_d-\rho_t)\lambda_t / (1-p_0)^{M-1} & \mu_2 = 1 / (X_{req_1} + (M-1)X_{req_2}) \\ \lambda_3 = \lambda_2(M-1) & \mu_3 = 1 / X_{reply} \end{cases} \qquad (4.25)$$

Note that μ_2 does not include the waiting time of a requester event, as explained earlier. Also note that we do not include X_i in μ_3 for simplicity. X_i is needed only when a replier event starts a busy period.

The Laplace transforms of service times are $B_p^*(s)=\frac{\mu_p}{s+\mu_p}$, for $p=1,2,3$, because service times are exponentially distributed (Section 4.1). In addition, when P, the number of priorities, is 3, and $p=3$, $\lambda_H=B_H^*(s)=0$, and, $\lambda_L=\lambda_1+\lambda_2$. Substituting these into Eqs. (4.23) and (4.24), we have the Laplace transform of the acknowledgement time:

$$ACK^*(s) = S_3^*(s) = \frac{(1-\frac{\lambda_1}{\mu_1}-\frac{\lambda_2}{\mu_2}-\frac{\lambda_3}{\mu_3})s+\lambda_1+\lambda_2-\frac{\lambda_1\mu_1}{s+\mu_1}-\frac{\lambda_2\mu_2}{s+\mu_2}}{s-\lambda_3+\lambda_3\frac{\mu_3}{s+\mu_3}}\frac{\mu_3}{s+\mu_3}$$

$$= c_1\frac{\mu_1}{s+\mu_1} + c_2\frac{\mu_2}{s+\mu_2} + (1-c_1-c_2)\frac{\mu_3-\lambda_3}{s+\mu_3-\lambda_3} \qquad (4.26)$$

where

$$c_1 = \frac{\lambda_1}{\mu_1}\frac{\mu_3}{\mu_3-\lambda_3-\mu_1} \qquad (4.27)$$

$$c_2 = \frac{\lambda_2}{\mu_2}\frac{\mu_3}{\mu_3-\lambda_3-\mu_2} \qquad (4.28)$$

The pdf of acknowledgement time is the inverse of Eq. (4.26):

$$ack(t) = c_1\mu_1 e^{-\mu_1 t} + c_2\mu_2 e^{-\mu_2 t} + (1-c_1-c_2)(\mu_3-\lambda_3)e^{-(\mu_3-\lambda_3)t} \qquad (4.29)$$

This is a hyper-exponential distribution. Note that $c_1 \approx \lambda_1/\mu_1$ and $c_2 \approx \lambda_2/\mu_2$ when $\mu_3 \gg \lambda_3$, $\mu_3 \gg \mu_1$, and $\mu_3 \gg \mu_2$. In other words, c_1 and c_2 are approximately the probabilities that the server is serving a customer of priority 1 and 2 respectively. Moreover, $\mu_1 e^{-\mu_1 t}$ and $\mu_2 e^{-\mu_2 t}$ are exactly the residual service time densities of customers in priority group 1 and 2, and $(\mu_3-\lambda_3)e^{-(\mu_3-\lambda_3)t}$ corresponds to the waiting time density of an M/M/1 queue. Therefore, the three terms of $ack(t)$ approximately correspond to the three factors discussed at the beginning of this section.

The PDF of the acknowledgement time is

$$ACK(t) = \int_0^t ack(x)dx = 1 - c_1 e^{-\mu_1 t} - c_2 e^{-\mu_2 t} - (1-c_1-c_2)e^{-(\mu_3-\lambda_3)t} \qquad (4.30)$$

Eqs. (4.3) and (4.4) still hold for the optimistic-synch algorithm. Hence the mean synchronization time for a primitive requester event is

$$SYNC = \int_0^\infty (M-1)[ACK(t)]^{M-2}\, ack(t)\, t\, dt \qquad (4.31)$$

4.2.2.3. Busy periods

The analysis of busy periods is similar to that in Section 4.2.1.3, except for the calculations of T_{req}, T_{reply} and I. For the mean duration of requester events, Eq. (4.10) now holds for a single round, *i.e.*,

$$T_{req}^0 = X_{req_1} + (M-1)X_{req_2} + SYNC \qquad (4.32)$$

Since each requester event has on the average $1/(1-p_0)^{M-1}$ rounds, the mean duration of requester events is

$$T_{req} = \frac{T_{req}^0}{(1-p_0)^{M-1}} = \frac{X_{req_1} + (M-1)X_{req_2} + SYNC}{(1-p_0)^{M-1}} \qquad (4.33)$$

The mean initial delay for a replier event now includes the residual service time of interrupt disabling events as well as the residual service time of primitive requester events:

$$I = \rho_d X_d + \lambda_{req}^0 [X_{req_1} + (M-1)X_{req_2}]^2 \qquad (4.34)$$

The mean duration of a replier event that starts a busy period is the overhead of dispatching an interrupt plus the execution time of the event:

$$T_{reply} = X_i + X_{reply} \qquad (4.35)$$

Besides these, the equations for busy periods (Eqs. (4.12) and (4.17)) and CPU overhead (Eq. (4.18)) are still valid.
The bounds of T_{reply}' are also derived similarly. The only difference is that a replier does not wait in the optimistic-synch algorithm. Hence the upper bound of T_{reply}' does not include $SYNC$:

$$0 \le T_{reply}' \le X_{reply} \qquad (4.36)$$

Note that these bounds are much narrower than those in Eq. (4.19), because $SYNC$ is usually much larger than X_{reply}.

4.2.2.4. The complete procedure

The iterative procedure for analyzing the optimistic-synch algorithm is the same as that for the 2-phase algorithm (Section 4.2.1.4), so it is not repeated here. Note that some performance measures are now computed using different equations: $SYNC$, λ_{reply}', I, T_{req}, and T_{reply} are calculated using Eqs. (4.31), (4.22), (4.34), (4.33) and (4.35) respectively. Also, Eqs. (4.25), (4.27), (4.28), (4.29), and (4.30) are needed for calculating $SYNC$.

Again, this procedure is performed twice, once with $T_{reply}' = 0$ for lower bounds, and once with $T_{reply}' = X_{reply}$ for upper bounds. Its results immediately give the performance measures we want to find, in the same way as in Section 4.2.1.4.

4.2.3. The optimistic-async algorithm

The analysis of the performance of the optimistic-async algorithm is much easier than those presented in the previous sections. The CPU overhead is computed using simple algebra; the latency is calculated using a non-priority queueing model. Again, we used the notation and terminology defined in Sections 4.2.1 and 4.2.2.

4.2.3.1. CPU overhead

The optimistic-async algorithm also carries out a TLB operation in one or multiple rounds, but it differs from the optimistic-synch algorithm in three ways. First, the processor that initiates the operation does not wait for the completion of the operation. In other words, the CPU overhead of the algorithm does not include the synchronization time as in other algorithms. This greatly simplifies the analysis of the CPU overhead. Second, replier events are not executed via interprocessor interrupts, but are examined and executed between slices of regular jobs. Hence, the overhead of dispatching an interrupt (X_i) is not included in the overhead of a replier event. Third, at the end of a round, it is the last replier instead of the requester that checks the results and repeats the operation in case of failure. Hence, different rounds of a TLB operation may be initiated by different processors. However, as far as event rates are concerned, Eqs. (4.21) and (4.22) are still valid for λ_{req}^{0} and λ_{reply}.

Directly from input parameters, the mean overhead of a primitive requester event and a replier event can be obtained as $X_{req_1}+(M-1)X_{req_2}$ and X_{reply}, respectively. Multiplying them by their rates and summing up, we obtain the total CPU overhead

$$\rho_t = (1-\rho_d-\rho_t)\,\frac{\lambda_t}{(1-p_0)^{M-1}}\,[X_{req_1}+(M-1)(X_{req_2}+X_{reply})] \tag{4.37}$$

Solving Eq. (4.37) for ρ_t, we have

$$\rho_t = \frac{(1-\rho_d)\,\lambda_t\,[X_{req_1}+(M-1)(X_{req_2}+X_{reply})]}{(1-p_0)^{M-1}+\lambda_t\,[X_{req_1}+(M-1)(X_{req_2}+X_{reply})]} \tag{4.38}$$

Note that we ignore the overhead of checking whether the replier event queue is empty on every context switch.

4.2.3.2. Latency

For the optimistic-async algorithm, the latency of a TLB operation is defined to be the time it takes to complete the operation (Section 4.1.4). A TLB operation may consist of more than one rounds. We first compute the mean latency of a round, then multiply the result by the mean number of rounds per operation.

The mean latency of a round has two components: the overhead of the requester ($X_{req_1}+(M-1)X_{req_2}$), and the maximum delay of all repliers. Two major factors affect the delay of a replier event. First, if the replier is executing a slice of a job, it must finish the current time slice. Second, the other replier events that arrived before this event must be finished first.

In Section 4.2.2.2, we considered two other factors, *i.e.*, the execution of interrupt disabling events and the execution TLB requester events. We ignore them here for simplicity. This is reasonable because the durations of these events are much shorter than a time slice. Furthermore, we assume, as in UNIX, that the overhead of handling interrupts is charged to the current time slice. Hence, these events affect job scheduling only when their execution extends beyond the end of the current time slice.

We use a non-preemptive queueing model to find the time it takes to complete a replier event. In the model, each processor serves customers of two priority classes. High-priority ($p=2$) customers correspond to replier events. Arrivals follow a Poisson process with mean rate $\lambda_2 = \lambda_{reply}$; the service time is exponentially distributed with mean $1/\mu_2 = X_{reply}$. Low-priority ($p=1$) customers correspond to job slices. Their service time is a constant X_j; this arrival rate satisfies $\frac{\lambda_1}{\mu_1} + \frac{\lambda_2}{\mu_2} = 1$. The equation for λ_1 is a consequence of the assumption that the CPU is never idle (the CPU runs, say, the garbage collection job when it is otherwise idle). With this model, the time it takes to complete a replier event is simply the system time of high-priority customers.

Using the notation of Eq. (4.23), we have $P=p=2$, $B_1(s)=e^{-sX_j}$, $B_2(s)=\frac{\mu_2}{s+\mu_2}$, $\lambda_L=\lambda_1$, and $\lambda_H=B_H^*(s)=0$. Let $D^*(s)$ denote the Laplace transform for the completion time of a replier event. Then, by Eqs. (4.23) and (4.24),

$$D^*(s) = \frac{\lambda_1(1-e^{-s}X_j)}{s - \lambda_2 + \lambda_2 \frac{\mu_2}{s+\mu_2}} \frac{\mu_2}{s+\mu_2}$$

$$= \frac{(\mu_2-\lambda_2)}{X_j} \frac{1-e^{-s}X_j}{s\,(s+\mu_2-\lambda_2)} \tag{4.39}$$

The pdf of the completion time of a replier event, $d(t)$, is the inverse of Eq. (4.39)

$$d(t) = \begin{cases} \frac{1}{X_j}(1-e^{-(\mu_2-\lambda_2)t}), & \text{if } t \le X_j \\ \frac{1}{X_j}(1-e^{-(\mu_2-\lambda_2)X_j})\, e^{-(\mu_2-\lambda_2)(t-X_j)} & \text{if } t > X_j \end{cases} \tag{4.40}$$

Integrating Eq. (4.40) over t, we have the PDF of the completion time of a replier event,

$$D(t) = \begin{cases} \frac{t}{X_j} - \frac{1-e^{-(\mu_2-\lambda_2)t}}{(\mu_2-\lambda_2)X_j}, & \text{if } t \le X_j \\ 1 - \frac{1-e^{-(\mu_2-\lambda_2)X_j}}{(\mu_2-\lambda_2)X_j}\, e^{-(\mu_2-\lambda_2)(t-X_j)}, & \text{if } t > X_j \end{cases} \tag{4.41}$$

Eq. (4.4) is still true for the maximum of $M-1$ completion times. Hence, the mean latency of one round of a TLB operation is

$$T_{req}^{0} = X_{req_1} + (M-1)X_{req_2} + \int_0^{\infty}(M-1)[D(t)]^{M-2}\, d(t)\, t\, dt \tag{4.42}$$

Since each TLB operation has on the average $1/(1-p_0)^{M-1}$ rounds, its mean latency is

$$T_{req} = \frac{T_{req}^{0}}{(1-p_0)^{M-1}} \tag{4.43}$$

This completes the analysis of the performance of the optimistic-async algorithm. To summarize, the per-processor CPU overhead of this algorithm is ρ_t (Eq. (4.38)), the latency is T_{req} (Eq. (4.43)), the per-processor throughput of TLB operations is λ_{req} (Eq. (4.7)), and the per-processor space overhead is, again by Little's formula [Kle75], $\lambda_{req} T_{req}$ pages.

4.3. Simulation

To validate our analytic results, we simulate the performance of the three TLB synchronization algorithms analyzed in Section 4.2. Particularly, we want to see whether the iterative procedures (Sections 4.2.1.4 and 4.2.2.4) converge to correct values, and whether the simplifications we made in the analysis are reasonable.

Mistakes in performance analysis or simulation are hard to detect, because they often cause no other symptoms but wrong performance numbers. Whereas in building a real system, bugs cause the system to function incorrectly and are hence easier to detect. We are confident in the correctness of our results only if we can obtain similar results using completely different methods.

Below, we briefly describe the simulators, estimate their computation time, and compare their results with those obtained by analysis.

4.3.1. The simulators

We built an event-driven simulator for each of the three algorithms. The simulators use the model and assumptions in Section 4.1, but do not make the extra simplifications we made in the analysis. Events are generated stochastically according to the workload assumptions in Section 4.1, and recursively as a result of executing events. The simulators essentially implement the exact TLB synchronization algorithms, *i.e.*, they have code corresponding to every step of the algorithms. The simulators maintain state information for each processor, and execute code by changing the states of the processors. Events are executed in the order of their simulated times. For each event, the simulators carry out the algorithms, update simulated time, collect and output statistics, and generate more events if necessary.

We decoupled the simulators into two parts: commands of the *B*erke*L*ey Interactive *S*tatistics *S*ystem (BLSS) [AbR88], and C programs (which are also executed as BLSS commands). BLSS has a rich command set for data manipulation and visualization. We used BLSS commands to generate datasets consisting of raw input events. The C programs, which are engines driven by events, take raw input data and produce raw output data. We again use BLSS commands to extract statistics out of the raw output data, and to visualize the results. This approach reduces the amount of programming, because we need not write C code that deals with probability and data manipulation. Moreover, we can easily change the workload and plot different graphs without modifying the C programs. Note that we did not use a standard simulation tool, such as GPSS [BKP76, Sch74], SIMSCRIPT II.5 [MKV87], SLAM [Pri86], INSIGHT [Rob83], and SIMAN [Peg82]. In our problem, processors interact and synchronize with each other. Moreover, we have to deal with details such as deadlock avoidance. We felt that C is more flexible and less restrictive in handling these issues.

The computation time of the simulators is estimated below. There are N processors each generating TLB operations approximately at a rate λ_t. Each operation generates an event for M processors. Determining the next due event, which is the earliest of all pending events, can be done either by scanning the state of N processors, or by maintaining a sorted list of pending events. Both methods take order $O(N)$ time. Hence the computation time of the simulators is of order $O(N^2 M \lambda_t T)$, where T is the total simulated time. As a numeric example, for the parameters given in Figure 4.2(a) and T = 2000 ms, the data point corresponding to M=40 takes 137.7 seconds of CPU time on a VAX 8600 when N=M, and 2275.4 seconds when N=4M. On the other hand, the analytic methods need little computation, and, most importantly, the computation time is more or less constant. It takes only about 20 ms of CPU time on a VAX 8600 to generate the same data point using the analytic procedure described in Section 4.2.1.4.

4.3.2. Comparison of simulation and analytic results

We computed performance measures for a set of representative parameters using both the simulation and analytic techniques. This section focuses on the differences between analytic and simulation results; interpretations of the results are deferred untilSection 4.4.

The performance measures compared are CPU overhead and latency; the parameter values used are λ_t = 25/sec., λ_d = 100/sec., X_d = 0.5 ms, $X_{req_1} = X_i = X_{req_2}$ = 0.04 ms X_{reply} = 0.04 ms, X_j = 25 ms, p_0 = 0.0002, and variable M. We will present more performance measures (throughput and space overhead) using the same parameter values in Section 4.4. Moreover, Section 4.4 plots results using various parameters (mean arrival rates and mean service times in addition to M) as the X axis.

We verified our analytic results only for parameter values that we are interested in. The total number of possible parameter values is infinite.

Moreover, simulation is computationally expensive (of order $O(N^2 M \lambda_t T)$), or even infeasible, when N or λ_t is large. Therefore, we did not compare all performance measures for all possible parameter values.

4.3.2.1. The 2-phase algorithm

The analysis of the 2-phase algorithm made two additional simplifications with respect to those made by the simulation. We assumed that processors acknowledge TLB requests independently. In reality, processors are more or less coordinated by the 2-phase algorithm. The algorithm causes them to stall and resume at about the same time, and hence delays and synchronizes the execution of other events, *e.g.*, interrupt disabling events. However, we believe that the effect of this is insignificant because the duration of the algorithm is relatively short.

We also assumed in the analysis that TLB events arrive independently of the state of the system. In fact, all processors involved in the algorithm are temporarily stalled and cannot generate new TLB events. Hence the number of processors that can independently generate TLB events is at most $N-M$ (total number of processors minus the number of processors involved in each operation) when the algorithm is in progress. Nevertheless, the effect of N has been eliminated in the analysis because of the above simplification. We examine this here by simulating the algorithm with $N=M$ and $N=4M$. Note that in Eqs. (4.12) and (4.17), the arrival rate λ_{reply} appears in the expression $1 - \lambda_{reply} T'_{reply}$. Hence, the extreme case (*i.e.*, $\lambda_{reply} = 0$, when no new TLB events can arrive during the algorithm) can be covered by the lower bounds, which are based on $T_{reply} = 0$.

Figure 4.2(a) shows the CPU overhead of the 2-phase algorithm. The lower and upper bounds are reasonably close to each other, and agree fairly well with the simulation results. Also, the effect of N is insignificant: the CPU overhead is only slightly higher with $N=4M$ than with $N=M$. Therefore, we conclude that the simplifications we made are reasonable for calculating CPU overhead. In fact, it is generally true that the mean utilization of a closed system is not very sensitive to the approximations made in the analysis [Agr85, R.S83].

Figure 4.2(b) shows the latency of the 2-phase algorithm. The simulated latencies for $N=M$ and $N=4M$ are pretty close, so the effect of N on latency is also insignificant. The lower bounds, upper bounds and simulation results agree well only when $M < 20$, which approximately corresponds to $\rho_t < 20\%$ according to Figure 4.2(a). When M is larger, the lower bounds are still close to the simulation results, but the upper bounds increase more rapidly. In other words, the method we used to calculate the upper bounds works well only when the system is far from saturation (*i.e.*, the CPU overhead is low). This outcome justifies our decision not to assume in the first place that TLB events are executed serially (this is how the upper bounds were derived). Note that Lee and Garcia-Molina have assumed this in evaluating the performance of distributed databases [Gar81, Lee80]. Additionally, they have assumed that events are generated

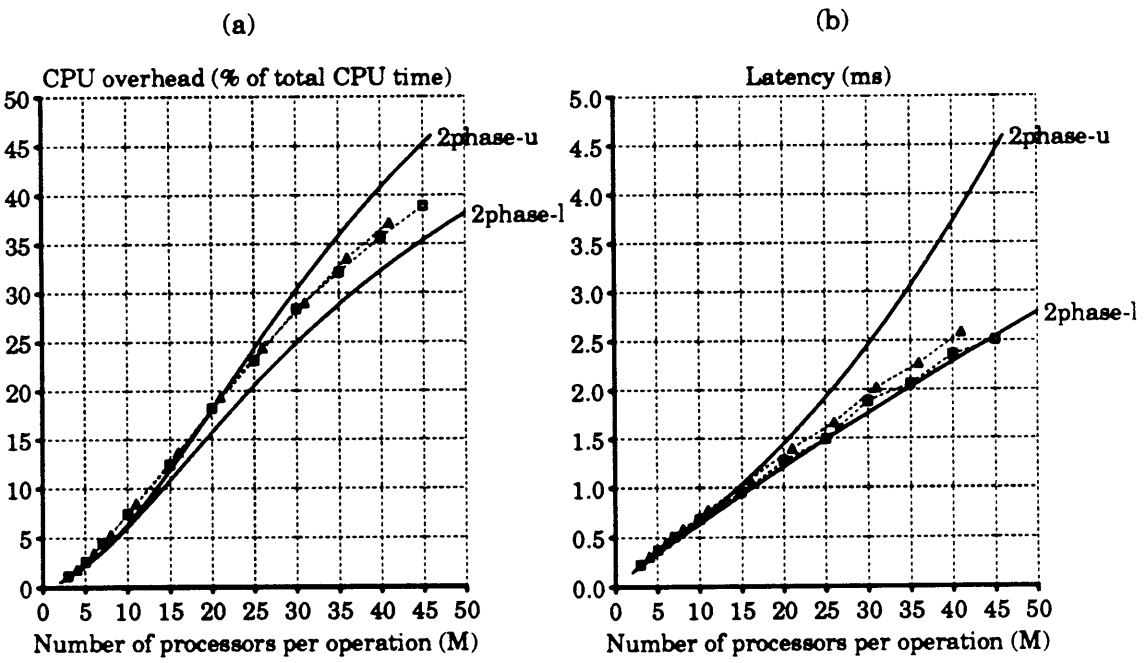


Figure 4.2. Simulation vs. analysis: the 2-phase algorithm. Parameters are $\lambda_t = 25/\text{sec.}$, $\lambda_d = 100/\text{sec.}$, $X_d = 0.5$ ms, and $X_i = X_{req_1} = X_{req_2} = X_{reply} = 0.04$ ms. Solid lines are analytic bounds. Triangles are simulation results for $N=4M$. Squares are simulation results for $N=M$.

externally, *i.e.*, their arrival rate does not decrease when the CPU is saturated; this could make the estimation of latency even more inaccurate.

4.3.2.2. The optimistic-synch algorithm

Figures 4.3(a) and 4.3(b) show the CPU overhead and latency of the optimistic-synch algorithm. For both performance measures there is no noticeable difference among the upper bounds, the lower bounds, and the simulation results. In addition, the effect of N is insignificant. We made similar simplifications for analyzing the optimistic-synch algorithm as for the 2-phase algorithm. The results are more accurate here because we have tighter bounds (*SYNC* exists in Eq. (4.19) but not in (4.36)). We thus conclude that the analysis for this algorithm yields accurate results, at least for the parameter values we chose in our comparison.

4.3.2.3. The optimistic-async algorithm

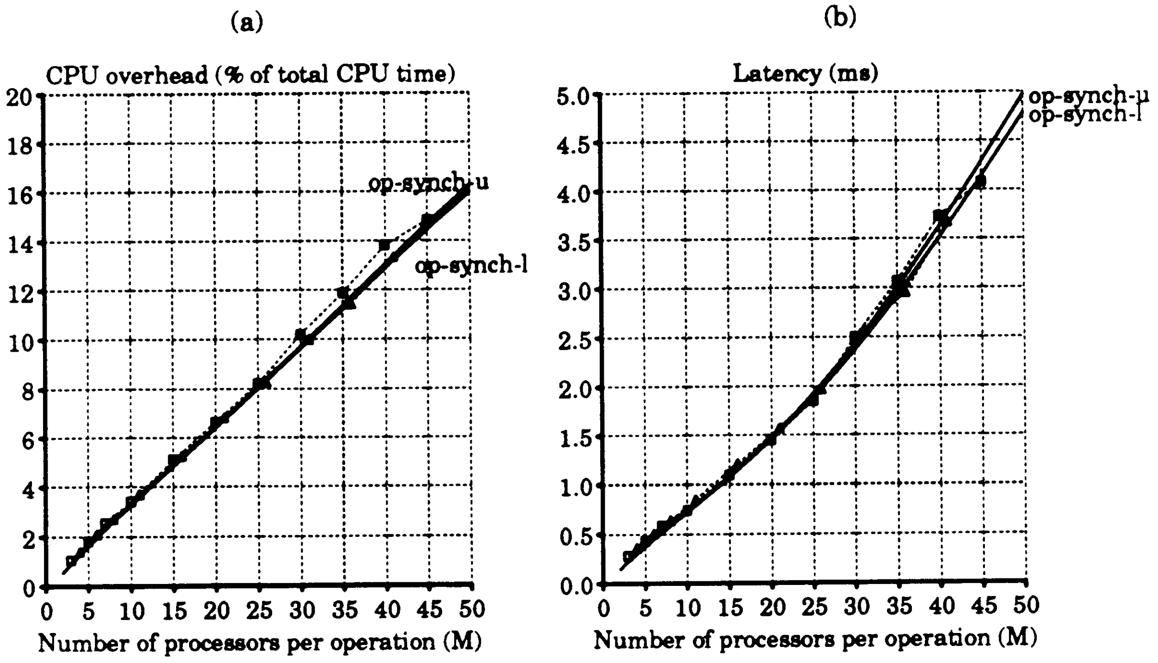


Figure 4.3. Simulation vs. analysis: the op-synch algorithm. Parameters are $\lambda_t = 25/\text{sec.}$, $\lambda_d = 100/\text{sec.}$, $X_d = 0.5$ ms, $X_i = X_{req_1} = X_{req_2} = X_{reply} = 0.04$ ms, and $p_0 = 0.0002$. Solid lines are analytic bounds. Triangles are simulation results for $N=4M$. Squares are simulation results for $N=M$.

Figures 4.4(a) and 4.4(b) show the CPU overhead and latency of the optimistic-async algorithm. Again, the simulation results and the analytic results are in excellent agreement. This implies that the simplifications we made in the analysis are very reasonable. Those simplifications are: (1) processors acknowledge TLB events independently, and (2) background job execution is the dominant factor of delay. The discussion for (1) is the same as in Section 4.3.2.1. The shape of the curve in Figure 4.4(b) supports point (2): As M increases, the delay increases rapidly up to the length of a time slice (X_j) and them increases much more slowly.

4.4. Performance Results

This section presents and discusses performance results for the three TLB synchronization algorithms. The performance measures shown below include CPU overhead, latency, space overhead, and throughput (Section 4.1.4). We first explain how the results were obtained and plotted, and then show graphs of the results.

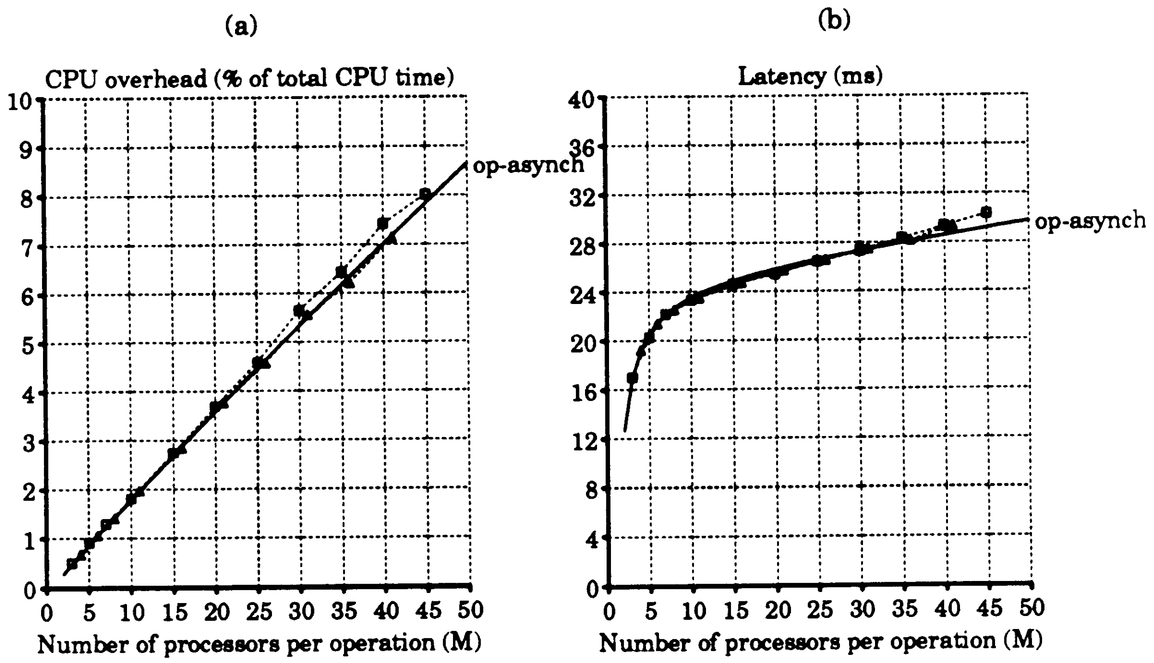


Figure 4.4. Simulation vs. analysis: the op-async algorithm. Parameters are $\lambda_t = 25/\text{sec.}$, $\lambda_d = 100/\text{sec.}$, $X_d = 0.5$ ms, $X_i = X_{req_1} = X_{req_2} = X_{reply} = 0.04$ ms, $p_0 = 0.0002$, and $X_j = 25$ ms. Solid line is analytic results. Triangles are simulation results for $N{=}4M$. Squares are simulation results for $N{=}M$.

The numbers shown here are pure analytic results. We use only analytic results because (1) they are computationally inexpensive to obtain, and (2) they seem to agree well with simulation results (Section 4.3.2).

The four performance measures are obtained as follows. We first calculate CPU overhead and latency using the equations and procedures given in Section 4.2. We show a single value for each case, even if the analysis gives only bounds. For the latency of the 2-phase algorithm, we use lower bounds because they are reasonably close to simulation results, whereas upper bounds do not (Section 4.3.2.1). In other cases, the bounds are close, so we arbitrarily choose their arithmetic means. We then derive the throughput of TLB operations from CPU overhead using Eq. (4.7). Space overhead is defined as the mean number of pages that are being unmapped and are thus unavailable for other uses. It is, by Little's formula [Kle75], simply equal to the product of throughput times latency.

4.4.1. Choosing workloads

Since there are many parameters that can vary independently, it is impractical to compute or to interpret performance results for all possible combinations of parameters. We hence focus our attention on the key issue: *whether the TLB synchronization algorithms scale well.* We start with a canonical set of parameter values. Then for each factor that we are interested in scaling, we plot performance graphs by varying the corresponding parameter, taking the other values from the canonical set.

The canonical set of parameter values is just a starting point. It represents a reasonable workload under the current technology, though not necessarily the workload of a specific application on a specific machine. In other words, we use it as a basis for seeing trends, rather than for evaluating the performance of a particular system. The canonical values are $M = 16$, $\lambda_t = 25$/sec., $\lambda_d = 100$/sec., $X_d = 0.5$ ms, $X_i = X_{req_1} = X_{req_2} = X_{reply} = 0.04$ ms, $X_j = 25$ ms, and $p_0 = 0.0002$ (see Section 4.1 for the definition of these parameters). Note that we also used these values in comparing the simulation and analytic results (Section 4.3.2). In fact, the six graphs (Figure 4.2(a) to 4.4(b)) in Section 4.3.2 are condensed in Figures 4.5(a) and 4.5(c).

We vary parameter values in the following three directions in which future technology and applications are likely to develop.

- *Number of processors (M).* The number of processors involved in a TLB operation (M) varies from one to the total number of processors in the system (N), depending on the degree of parallelism exploited by software. We discuss N instead of M here because M scales with N and there are always applications with M=N, *e.g.*, the operating system itself, if it is symmetrical and runs on every processor.
  The number of processors that a shared-memory multiprocessor system can support is mainly limited by the system bus bandwidth [GaP85]. This number is small for a single-bus system, *e.g.*, up to 32 for the Sequent Symmetry system [BKT87, SSS87]. However, it can be greatly increased by using multiple buses and/or multi-level caches, as in the following examples. The Wisconsin Multicube supports more than 1,000 processors [GoW88, GVW89]; Agarwal *et al.* proposed a directory scheme for scalable shared-memory multiprocessors [ASH88]. Hendrik Goosen at Stanford is extending the VMP multiprocessor to thousands of processors (VMP-MC) using a memory hierarchy based on shared caches [CGB89]. All these systems or designs support the shared-memory model, and provide coherent caches.
- *Arrival rate (λ_t).* The rate of TLB operations depends on applications. Paging alone is not likely to generate a high rate of TLB operations as main memory grows larger and larger. On the other hand, several future applications will use virtual memory remapping for special purposes, and can potentially generate a high rate of TLB operations.
  One such example is the XPRS project, which is targeted at one

thousand TP1 transactions per second using a shared-memory multiprocessor [SKP88]. Part of the design is a fast recovery mechanism that protects buffers against software errors [Sul90]. In short, the system always keeps one unwritable copy of all recoverable data in a protected memory region called *stable buffers*. This implies that virtual memory remapping is needed when updating the data. Since the goal of the system is one thousand transactions per second, the total rate of TLB operations could be high.
Another example is high-performance network communication. Next-generation networks will provide host-to-host throughput in the range of 100 Mbps to 1 Gbps [BKN89, Lei88, Ros86]. One goal of current research is to make this performance available to user processes [Lei88]. To achieve this goal, memory remapping must be used for moving data across address space boundaries. This is because at very high bandwidths an extra software copy operation forced by the operating system could easily double the time for delivering a packet to a user process. If the host computer is a shared-memory multiprocessors, as in the VMP/NAB project [KaC88], a TLB synchronization operation may be needed for every page remapped.
Moreover, there is a trend in operating system design towards moving data servers (file servers, transaction managers, etc.) from the kernel to user processes, as in V [Che84], Ridge [Bas85], and QuickSilver [HMS88]. A data access in such an organization usually involves several data movements between virtual address spaces. A file request from a user client process to a user-level file server might be routed through a transaction manager and a network communication manager at each end. As a result, virtual memory remapping becomes an important technique to avoid the negative performance impact of copying large amounts of data between virtual address spaces.

- *Service time* (X_{reply}). X_{reply} includes mainly the overhead of getting a request block from a queue, examining the request, freeing the request block, and flushing a TLB entry. The cost of the last operation may vary widely on different machines. For many existing architectures (*e.g.*, the VAX), flushing a TLB entry takes only a few assembly instructions, and the cost is trivial. But for machines with a virtually tagged cache, which have recently gotten more popular, invalidating the mapping of a page requires flushing all cache lines corresponding to that page (Section 2.3). The cost could be high because (1) the number of lines per page may be large, and (2) flushing a dirty line generates main memory traffic. The second aspect is particularly undesirable, for all processors involved in a TLB synchronization operation flush their TLBs at about the same time. Further, flushing caches would temporarily increase the cache miss rate; this extra overhead should also be considered as part of the service time. Finally, some RISC processors do not have instructions for selective flushing a TLB entry, *e.g.*, the Intel 860 (formerly known as the N10) has only instructions for flushing the entire TLB [III88].

We only vary parameter values in the above three directions. Hence, the performance results shown below do not reflect other expected technological developments, such as increases in processor speeds.

4.4.2. The effect of the number of processors

Figure 4.5 compares the performance of the three algorithms, with a variable number of processor per operation (M). In general, there is a tradeoff between CPU overhead and latency. The 2-phase algorithm has the highest CPU overhead and the lowest latency; the optimistic-asynch algorithm has the lowest CPU overhead and the highest delay; the optimistic-synch algorithm is in the middle for both measures.

Increasing M has two major effects on performance: (1) increasing the number of replier events that a processor has to handle (Eq. (4.8)), and (2) increasing the request synchronization time, which is the maximum of M acknowledgement times. As a result, both CPU overhead and latency increase with M.

We consider a TLB synchronization algorithm unacceptable if its CPU overhead is greater than, say, 10% of the total CPU time. In this sense, the 2-phase algorithm works only when M is small (less than 15 in Figure 4.5(a)). The optimistic-synch algorithm greatly reduces CPU overhead by not requiring repliers to wait for synchronization. Thus it cuts the CPU overhead for handling a replier event, though not for a requester event. The improvement is roughly the area between the top two lines in Figure 4.5(a). Clearly, stalling all repliers is very expensive, especially when M is large. The optimistic-async algorithm further reduces CPU overhead by (1) batching operations to avoid interprocessor interrupts, and (2) not requiring the requester to wait for synchronization. Its CPU overhead is about 10% of total CPU time when M=50 in Figure 4.5(a).

The throughput of the algorithms decreases as M increases (Figure 4.5(b)). In particular, there is a 40% drop for the 2-phase algorithm as M increases to about 50. In our model, a processor cannot generate TLB operations when it is already executing a TLB synchronization algorithm (Section 4.1). Hence, the throughput of an algorithm decreases as its CPU overhead increases. Without this self-regulating effect, CPU overhead would increase more rapidly with M than in Figure 4.5(a), particularly for the 2-phase algorithm.

The latency of the optimistic-async algorithm is much higher than that of the other two (Figure 4.5(c)). In this algorithm, replier events are executed in batches, so the latency is dominated by how often batches are executed (*i.e.*, by X_j). In Figure 4.5(c), the top curve increases rapidly to X_j and then slowly. The other two algorithms both require a requester to wait for synchronization. However, the optimistic-synch algorithm causes longer delays because (1) it may fail and hence need more than one round, and (2) its synchronization time is longer. Section 4.2.2.2 discusses synchronization time in detail. In short, a requester waits until all repliers have stalled in the 2-phase algorithm, but waits until all repliers have finished processing the request in the optimistic-synch algorithm.

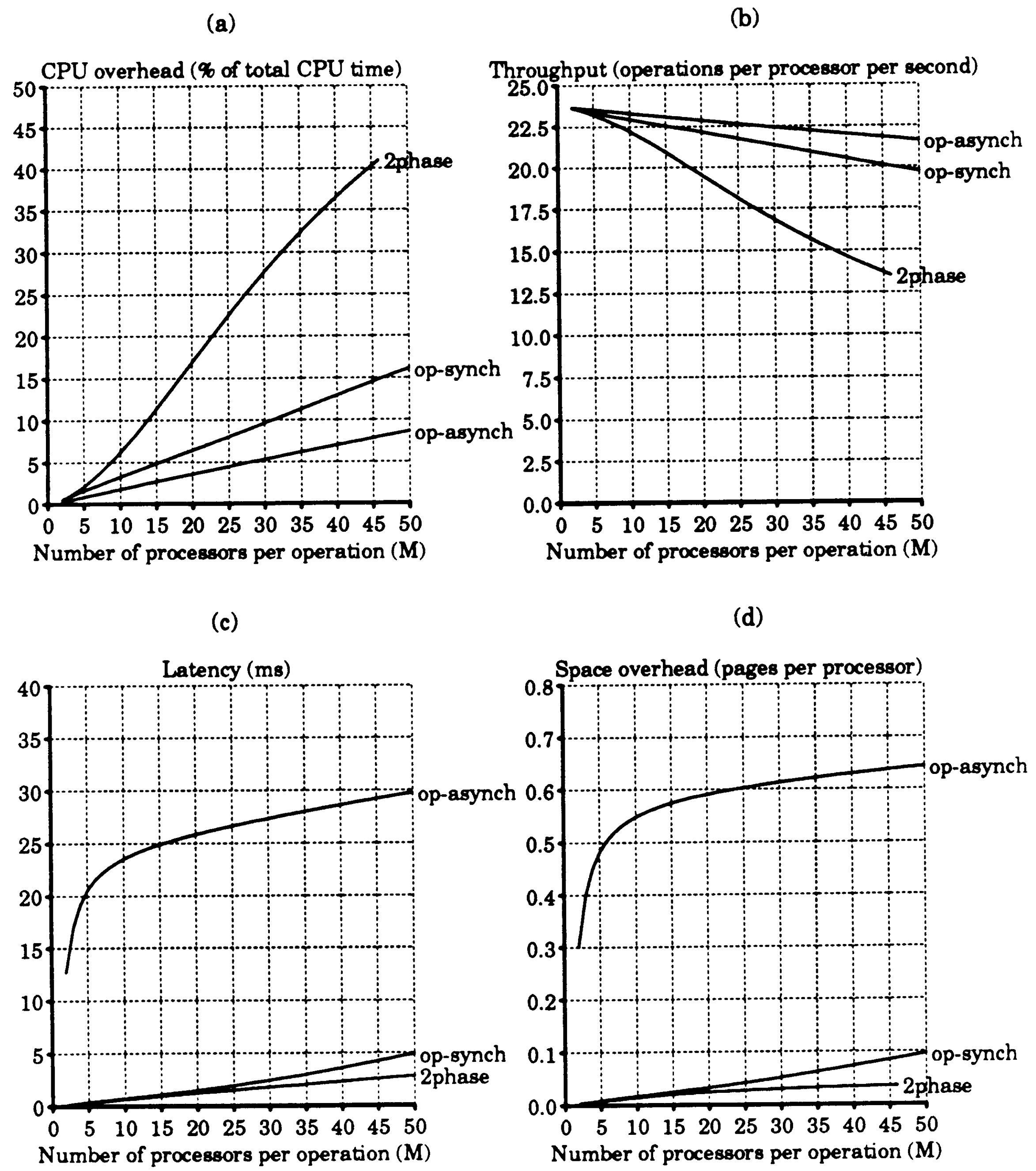


Figure 4.5. The effect of the number of processors on performance. Parameters are $\lambda_t = 25/\text{sec.}$, $\lambda_d = 100/\text{sec.}$, $X_d = 0.5$ ms, $X_i = X_{req_1} = X_{req_2} = X_{reply} = 0.04$ ms, $X_j = 25$ ms, and $p_0 = 0.0002$.

As CPU overhead increases, the chance that a replier has already stalled due to other requests gets higher.

The space overhead shown in Figure 4.5(d) is actually the mean number of outstanding TLB operations. When a page is unmapped from a virtual address space, *e.g.*, for pageout, the physical page cannot be assigned for other uses until the associated TLB operation has completed. Hence a page may be temporarily unavailable when a TLB operation is in progress. The shape of Figure 4.5(d) is similar to that of Figure 4.5(c). Since the absolute values are so low (less than one page for all cases shown), the relative differences among the three lines are not important. Consequently, space overhead should not be a factor to consider when choosing an algorithm.

4.4.3. The effect of the request rate

Figure 4.6 plots performance against the amount of regular computation between two TLB operations (*i.e.*, against $1/\lambda_t$). Reducing $1/\lambda_t$ increases the number of replier events each processor has to handle, and hence increases CPU overhead (Figure 4.6(a)). Reducing $1/\lambda_t$ also increases throughput. However, as shown in Figure 4.6(b), the throughput is less than λ_t because a processor generates TLB operations only when it is not executing a TLB synchronization algorithm. In Figure 4.6(a) and (b), the throughput corresponding to 10% CPU overhead is about 20 operations/processor/s. for the 2-phase algorithm, 40 for the optimistic-synch algorithm, and 80 for the optimistic-async algorithm. Such throughput may be enough for paging, but not for the special applications mentioned in Section 4.4.1.

Unlike M, λ_t has almost no effect on latency (Figure 4.6(c)). Two factors dominate latency: (1) the time for preparing and issuing interprocessor requests, and (2) synchronization time. Both of them are largely determined by M, and have little to do with λ_t. A higher arrival rate may cause a higher queueing delay (and hence a longer synchronization time) for some of the algorithms. However, since system utilization is low (less than 10% if for the system to be usable), the effect of queueing delay is unimportant.

The space overhead for the three algorithms varies widely (Figure 4.6(d)). But the differences are again unimportant because the absolute values are very small (less than 5 pages for the worst case).

4.4.4. The effect of service time

Figure 4.7(a) plots CPU overhead against the cost of processing a replier event (X_{reply}). The three lines all increase with X_{reply}, but at different rates. The rates differ mainly because X_{reply} has a different effect on the synchronization time for each algorithm. The optimistic-asynch algorithm does not require processors to wait, so its synchronization time is always zero. The optimistic-synch algorithms requires a requester to wait until all repliers have finished the request. A longer X_{reply} causes a longer acknowledgement time and hence a

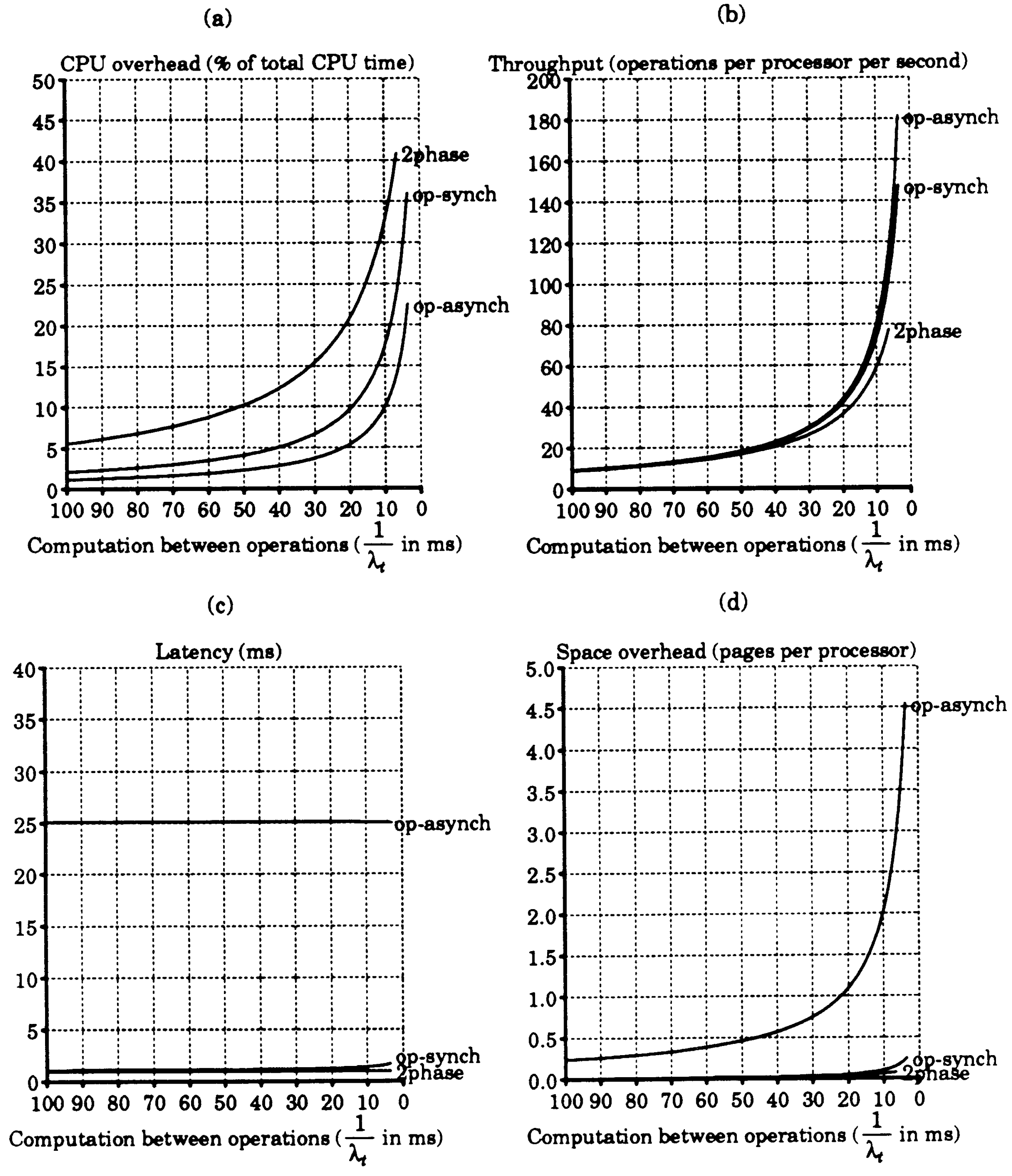


Figure 4.6. The effect of the request rate on performance. Parameters are $M = 16$, $\lambda_d = 100$/sec., $X_d = 0.5$ ms, $X_i = X_{req_1} = X_{req_2} = X_{reply} = 0.04$ ms, $X_j = 25$ ms, and $p_0 = 0.0002$.

longer synchronization time. The 2-phases algorithm requires a requester to wait until all repliers have stalled. As X_{reply} increases, the waiting time becomes shorter because the chance that a replier has already stalled due to other TLB operations is higher.

In this figure, CPU overhead is greater than 10% when X_{reply} is greater than about 250 μs for all three algorithms. We have stated in Section 4.4.1 that X_{reply} is high for machines with virtually tagged caches. We now use SPUR as a numeric example [Hil86, NeO88]. The cost of flushing a cache line ranges from 12 cycles (for reading tags of a nonexistent line) to 37 cycles (for a dirty line). Since a page consists of 128 lines (4096 bytes/page divided by 32 bytes/line), the total cost of flushing a page ranges from 1536 to 4736 cycles. Clearly, consistently changing the virtual memory mapping of a page is very expensive on shared-memory multiprocessors with virtually-tagged caches.

Figure 4.7(b) to (d) show the effect of X_{reply} on the other three measures. The discussion in Section 4.4.3 still applies, so we do not repeat it here.

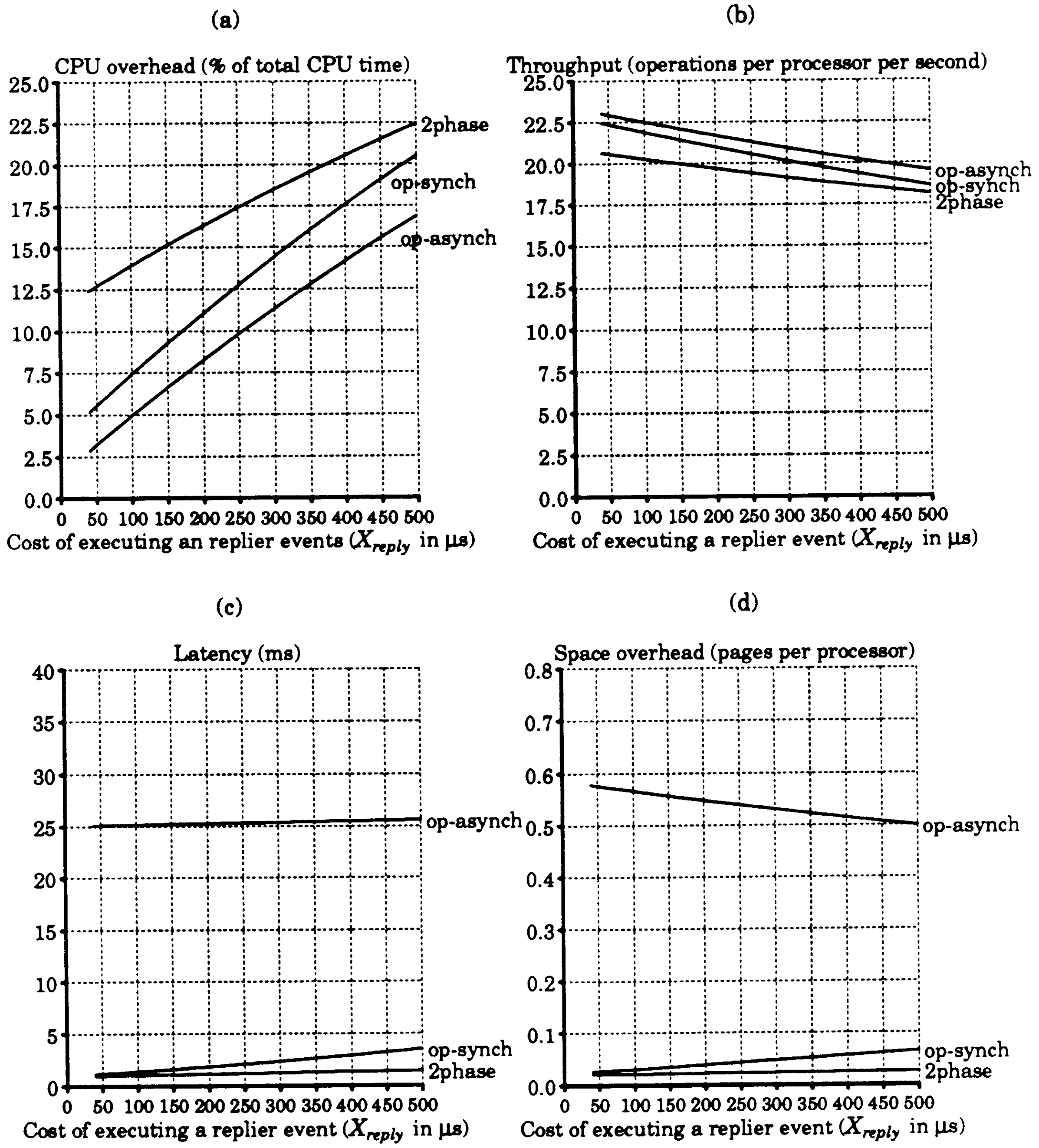


Figure 4.7. The effect of service time on performance. Parameters are $M = 16$, $\lambda_t = 25/\text{sec.}$, $\lambda_d = 100/\text{sec.}$, $X_d = 0.5$ ms, $X_i = X_{req_1} = 0.04$ ms, $X_{req_1} = X_{reply}$, $X_j = 25$ ms, and $p_0 = 0.0002$.

4.5. Summary and Conclusions

We have analyzed and simulated the performance of three TLB synchronization algorithms. This section summarizes our methods and results, and draws conclusions.

We use an iterative method to analyze the algorithms. We do not model and analyze the system as a queueing network. The interaction among processors, such as blocking for acknowledgements, makes product form queueing network solutions impossible. In the iterative method, we fix some performance measures to simplify the model, calculate other measures using the simplified model, and finally recalculate the fixed measures. This procedure is repeated until the fixed measures converge. We apply queueing theory only to some individual steps of this method, not to the problem as a whole. This method is computationally efficient; it converges rapidly for all reasonable parameters we have used.

We verify the analysis using simulation, and obtain results in good agreement with those produced by the analysis. We do not use a standard simulation language or tool, such as GPSS [Sch74]. Instead, we use the recently developed Berkeley Interactive Statistical System (BLSS) plus our own simulation engines written in C [AbR88]. In general, simulation is CPU intensive, particularly when the number of processors is large, and when the rate of TLB synchronization is high. Therefore, simulation is not an efficient method for studying scalability, which is one of our primary interests. On the other hand, the computation overhead of our analysis method has little to do with the values of the parameters. Hence, this method is good for studying scalability.

Performance results depend on the choice of parameter values. Our results do not reflect the performance of a specific algorithm on a specific machine; this study is not meant to be used to find optimal design points in the first place. On the other hand, our results show trends and are sufficient to support the following points:

- There is a tradeoff between CPU overhead and latency. For example, the 2-phase algorithms has the highest CPU overhead, but the lowest latency.
- All algorithms are CPU-bound under high workloads. Consequently, under such conditions, CPU overhead is the major consideration in choosing an algorithms.
- The CPU overhead of the 2-phase algorithm is much higher that that of the optimistic-synch and optimistic-asynch algorithms. The ratio varies with the values of the parameters, but is usually greater than 2:1.
- None of the these three algorithms scale well with the number of processors, the rate of TLB synchronization operations, or the overhead of flushing a TLB entry.

Our performance results can also be used to argue that some other TLB synchronization algorithms (see Chapter 3) do not scale well either. The optimistic-synch algorithm becomes a one-round algorithm when p_0 is zero. Since we used a very small value for p_0, for the performance of of the optimistic-synch algorithm should be close to that of the *PTE-first* and *TLB-first* algorithms for the parameter values we used. Similarly, the performance results of the optimistic-asynch algorithm should be close to that of the *PTE-first'*, *TLB-first'*, and *optimistic'* algorithms for the parameter values we used.

Therefore, we conclude that we need better ways to handle TLB inconsistency than the algorithms we have evaluated. To make virtual memory remapping scalable (*e.g.*, to allow its use for supporting high-performance I/O), we must reduce the CPU overhead of TLB synchronization. This usually involves exploiting asynchrony and weaker consistency semantics (see Chapter 3). More important, the operating system should avoid TLB synchronization by tolerating TLB inconsistency. The next chapter presents principles and a virtual memory system design based on this conclusion.

Chapter 5
Tolerating TLB Inconsistency

We have drawn two conflicting conclusions. Chapter 4 shows that software TLB synchronization algorithms do not scale well. On the other hand, Section 1.1 and Section 4.4.1 show situations where the rate of virtual memory remapping, the number of processors, or the overhead of flushing a TLB entry could be high. This chapter explains how we resolve this conflict. In short, we reduce the overhead of TLB synchronization by tolerating TLB inconsistency. We avoid the need for TLB synchronization by exploiting trust relationships and lazy remapping. When TLB synchronization is necessary, we do it efficiently by exploiting asynchrony.

This chapter consists of two parts. The first part (Sections 5.1 and 5.2) discusses principles for tolerating TLB inconsistency, omitting details to make the ideas clear. The second part (Section 5.3) presents a virtual memory system design, filling in details and demonstrating that the principles described in the first part can be used coherently. The design is part of the DASH virtual memory system design. We do not show the complete DASH design here, but emphasize the mechanisms for handling TLB inconsistency. Some techniques for tolerating TLB inconsistency have been mentioned earlier (Sections 2.4 and 3.1). The chapter still includes them for completeness.

5.1. Three Types of Tolerable TLB Inconsistency

We identify three fundamental types of tolerable TLB inconsistencies, upon which most mechanisms for tolerating TLB inconsistency are based.

(1) Safe inconsistency.

(2) Transient inconsistency.

(3) Trusted inconsistency.

For each type of inconsistency, we explain its properties, discuss its performance tradeoffs, and outline mechanisms that tolerate it (this section concentrates on high-level ideas; Section 5.3 gives details of the mechanisms).

5.1.1. Safe inconsistency

As explained in Section 2.4, when the operating increases the access rights of a PTE in main memory, the TLB inconsistency it causes is safe. Such an operation may produce stale TLB entries that allow less access rights than the corresponding PTE does. A legal memory reference using a stale TLB entry may thus be rejected, generating a protection fault. However, the page fault handler can fix the problem by flushing the stale TLB entry on the faulting processor. Therefore, using such a stale TLB entry does not hurt the integrity of the memory system, and is transparent to application programs.

Additionally, TLB entries with incorrect referenced bits are also safe. They do not allow a processor to make illegal memory references, though they may affect page replacement algorithms.

The operating system does not need extra code to exploit safe TLB inconsistency, except an extension to the page fault handler. On the other hand, the operating system does need to be aware of whether the TLB inconsistency caused by a virtual memory operation is safe or not. Otherwise, it may perform unnecessary TLB synchronization operations. See Section 2.4.2 for operating systems operations that tolerate safe TLB inconsistency.

Tolerating safe TLB inconsistency is not always free; we now discuss the performance tradeoff for the two cases mentioned above. First, tolerating TLB entries with insufficient access rights is an instance of the *lazy remapping* principle, which delays operations as long as possible. This approach replaces the overhead of synchronizing TLBs with the overhead of handling possible page faults. It may improve the overall performance because (a) a processor may not access the page corresponding to a stale TLB entry, or may have replaced the TLB entry before accessing the page; and (b) handling individual page faults is cheaper than synchronizing TLBs, which may involve interrupting and stalling processors.

Second, tolerating incorrect referenced bits may cause the page replacement algorithm to function suboptimally, particularly for LRU-based algorithms. Consequently, page replacement algorithms that do not rely on referenced bits, *e.g.*, the UNIX clock algorithm [BaJ81], are more suitable for shared-memory multiprocessors. This, however, is not an important issue in the future because, as physical memory becomes larger, paging activity will likely to be reduced significantly.

5.1.2. Transient inconsistency

Transient inconsistency corresponds to the temporary period of TLB inconsistency allowed by the weaker *consistent-when-done* semantics (defined in Section 3.1). Specifically, when the access rights to a page are being reduced from *high_rights* to *low_rights* by an algorithm, old TLB entries containing *high_rights* are considered valid by the algorithm until the algorithm terminates, even after the PTE has been changed to *low_rights*. The shaded area in Figure 5.1 illustrates the period of transient TLB inconsistency.

A transiently inconsistent TLB entry may allow greater access rights to a page than the corresponding PTE does. In other words, a processor may use it to make memory references that are not allowed by the corresponding PTE without generating a protection fault. Consequently, the operating system cannot completely ignore transient TLB inconsistency. It must keep track of all pages that are transiently inconsistent and manage them properly. The operating system is capable of doing this because it knows when transient inconsistency starts and

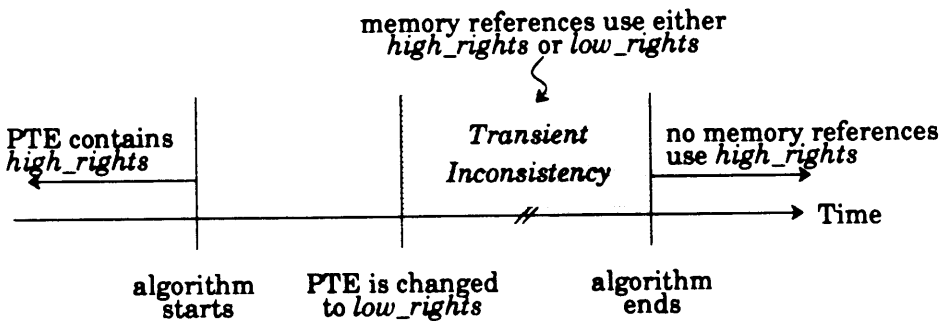

memory references use either *high_rights* or *low_rights*
PTE contains *high_rights*
Transient Inconsistency
no memory references use *high_rights*
Time
algorithm starts
PTE is changed to *low_rights*
algorithm ends

Figure 5.1. Transient TLB inconsistency.

ends; it initiates the VM operation that causes transient inconsistency.

By tolerating transient TLB inconsistency, the operating system can use more CPU-efficient TLB synchronization algorithms, *e.g.*, the optimistic algorithms as opposed to the 2-phase algorithm (see Chapter 3 and Table 3.1). Note that the period of transient TLB inconsistency can be extended over context switches or even longer, allowing asynchronous versions of the optimistic algorithm to be used.

Paging, in which the latency of pageout operations is unimportant, is an ideal situation to tolerate transient TLB inconsistency (*i.e.*, to use the asynchronous optimistic algorithm). In fact, some real systems have already exploited this, although they did not explicitly use the term transient inconsistency [FHM87]. To avoid the overhead of interprocessor interrupts, the operating system batches page invalidation (*i.e.*, pageout) requests together. It keeps track of all to-be-invalidated pages, and dose not recycle a physical page until the corresponding TLBs are synchronized. Before TLBs are synchronized, referencing an to-be-invalidated page is considered legal, and will cause that page to be revalidated.

Message-passing is another case in which transient TLB inconsistency can be tolerated. We assume that the system moves a page of data from the sender's virtual address space (VAS) to the receiver's using virtual memory remapping instead of software copying. We also assume that the semantics of message-passing ensure that, once a page has been received, it is protected from processes running in other VASs, including the sender's. To implement such semantics, the operating system either (1) unmaps the page from the sender's VAS and maps it into the receiver's, or (2) maps the page as read-only in both VASs. Both alternatives reduce the access rights of the page in the sender's VAS, and thus require TLB synchronization. The operating system can reduce the overhead of

TLB synchronization by tolerating transient TLB inconsistency during the time delay between a `send` and the corresponding `receive` operation. If the `send` operation precedes the `receive` operation, this delay is indefinitely long; if, on the other hand, `receive` precedes `send`, this delay is the latency to reschedule and wake up the receiver. Note that tolerating transient inconsistency may allow the sender to modify the message after it has sent it but before the receiver has received it. This, however, does not violate the integrity of the receiver; it is equivalent to modifying the message before sending it.

5.1.3. Trusted inconsistency

A system often has high-level semantic rules that restrict memory references, *e.g.*, a rule forbidding a process to access a virtual page that has been unmapped. These rules, when obeyed, can simplify TLB synchronization, because a stale TLB entry is harmless if the page it represents is never accessed.

Tolerating trusted TLB inconsistency is based on the above idea. The operating system allows user processes to specify trust relationships among themselves, where *trust* means trusting each other to obey high-level semantic rules. The operating system tolerates a stale TLB entry if (1) high-level semantic rules forbid the use of the TLB entry, and (2) processes that may use the stale TLB entry are trusted by processes that may be damaged by the stale TLB entry.

When a user process trusts others, it gives up, at least partly, the virtual memory protection provided by the operating system. The integrity of its VAS may be damaged if the high-level semantic rules are violated. In other words, the cost for better performance is reduced protection. However, this issue is not critical because cooperating processes implicitly trust each other to obey certain rules, *e.g.*, the format of the messages to be exchanged. When the rules are violated, they will not function correctly, in which case protecting their VASs is meaningless. Note that the operating system still fully protects VASs that do not exploit trust (but with performance penalties).

Message-passing is a good application in which trusted TLB inconsistency can be tolerated.. Continuing the example in Section 5.1.2, we assume that, when a page is transferred from one VAS to another, it is either (1) unmapped from the sender's VAS and mapped into the receiver's, or (2) mapped as read-only in both VASs. We define a semantic rule for each case. For case (1), the sender should not access the page after sending it; for case (2), the sender must inform the operating system before modifying the page [1]. Thus, when the access rights to a page are reduced in the VAS of a trusted sender, it is unnecessary to flush stale TLB entries; the semantic rule forbids the sender to use the excess access rights allowed by a stale TLB entry. In real applications, a receiver often trusts a sender. For example, a client may trust a server; a user process trusts

[1] This is similar to *copy-on-write*, except that a page is made writable by a system call rather than by the page fault handler.

the kernel. Also, a server can trust a client when the server does not interpret the received data, as in a file server or network server.

Moreover, the operating system can tolerate trusted TLB inconsistency within the kernel. In many situations, the kernel remaps a page for its own needs, *e.g.*, Sun UNIX remaps a page to a special range of virtual addresses to perform DMA [Che87, SSS85]. Usually, the kernel knows which pages are unsafe to access, and can avoid accessing them accordingly. Therefore, it is unnecessary to synchronize TLBs after remapping these pages; the kernel trusts itself not to use stale TLB entries.

5.2. Software Structure of the VM System

This section discusses OS organization and layering issues. It discusses only principles; Section 5.3 gives detailed designs. We first present a portable model of the virtual memory system, and then examine how mechanisms for handling TLB inconsistency fit into this model.

5.2.1. A portable model

As hardware becomes more diversified, many operating systems emphasize portability in structuring the virtual memory system [ATG88, GMS87, GMS87, OCD88, RTY88]. A popular model is to divide the virtual memory system into a large machine-independent part and a small machine-dependent part (Figure 5.2). The interface between the two parts is a key element of this model. It must be able to support various high-level VM constructs, and yet be easy to implement on different VM architectures. This

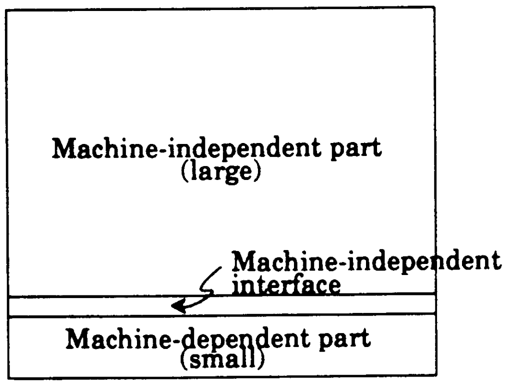


Figure 5.2. A portable model of the virtual memory system.

interface, which is itself machine-independent, defines a virtual machine for higher layers of the operating system.

5.2.2. Hiding TLB inconsistency inside the machine-dependent part

Given the portable VM model, one way of handling TLB inconsistency is to treat it as a machine-specific difficulty, *i.e.*, to hide it completely inside the machine-dependent part. When reducing the access rights to a page, the machine-independent part does not distinguish whether the hardware is a uniprocessor or a multiprocessor. It always calls the same function of the interface, and the machine-dependent part always uses a uniform mechanism to synchronize TLBs, if necessary.

This scheme, although straightforward, is inflexible in tolerating TLB inconsistency. Tolerating transient and trusted TLB inconsistency requires interaction between the machine-dependent and machine-independent parts. For example, the machine-independent part must keep track of pages that are transiently inconsistent (see Section 5.1.2). The machine-independent part must also explicitly define trust relationship in order to exploit trusted inconsistency (see Section 5.1.3).

Black and *et al.* argue that such inflexibility is not an issue, for the overhead of synchronizing TLBs is low according to their measurements [BRG89]. The system they measured had 16 processors and 96 MB of physical memory. The applications they measured were parallel transaction processing, kernel compilation, theorem proving and shortest path searching. These applications do not actively remap pages between VASs; they cause less than 0.5 TLB synchronization operations per second per processor, primarily due to paging.

However, there are cases where the number of processors is high or the rate of TLB synchronization is high (Section 4.4.1). We have shown that TLB synchronization algorithms do not scale well under these conditions (Chapter 4). Consequently, this scheme (hiding TLB inconsistency inside the machine-dependent part) is not a scalable solution.

5.2.3. Handling TLB inconsistency outside the machine-dependent part

To be scalable, the operating system should tolerate all three types of tolerable TLB inconsistency. In other words, TLB inconsistency should be made visible outside the machine-dependent part. We would like to do this in an abstract way, *i.e.*, the system should provide high-level software enough handles and hooks but avoid specifying detailed TLB characteristics.

For example, it is sufficient that the interface between the two parts of the VM systems (1) accepts performance hints on remapping operations, and (2) supports asynchronous remapping operations. Hints, *e.g.*, whether the latency of a remapping operation is critical, allow the machine-dependent part to make performance tradeoffs on a remapping operation and to pick the most suitable TLB synchronization algorithm, possibly tolerating transient TLB inconsistency. Asynchronous algorithms, *e.g.*, optimal-asynch, need proper support, such as completion notification, for bookkeeping and synchronization at higher layers.

Beyond the VM system, a subsystem that uses VM remapping should (1) define semantic rules, *e.g.*, page ownership, that regulate access to a page; (2) define trust relationships; and (3) allow its clients to specify trust relationships using hints. Based on such information, the subsystem determines whether it needs full protection from the VM system when a page is remapped, and then instructs the VM system accordingly.

5.3. A Virtual Memory System Design

This section presents a VM system design, filling in the details of the mechanisms and principles we have sketched in Sections 5.1 and 5.2. The design is part of the DASH distributed operating system kernel; some ideas for tolerating TLB inconsistency were sketched in an earlier report [TAG87]. This section does not intend to cover the complete DASH design, but focuses on mechanisms for tolerating TLB inconsistency. Section 5.3.1 gives an overview of the DASH kernel and of the VM system, establishing the context for the discussion in the following sections. Section 5.3.2 describes the interface between the machine-dependent and machine-independent part of the VM system. Section 5.3.3 and Section 5.3.4 present designs for tolerating TLB inconsistency in handling paging and message-passing respectively. Finally, Section 5.3.5 discusses copy-on-write, for which tolerating TLB inconsistency is difficult.

5.3.1. Overview and context

The DASH project has defined a communication architecture for a large, high-performance distributed system [AnF86, AFR87, And88, AnF88, ATW89]. The architecture is intended to support interactive multimedia communication, and is based on communication channels with real-time performance guarantees. The DASH kernel was developed from scratch as an experimental testbed. It is implemented in the object-oriented language C++ [Str86]. It is about 30,000 lines long, of which about 10,000 are comments. The DASH kernel currently runs on Sun 3/50 workstations. Although a Sun 3/50 workstation is a uniprocessor, the DASH kernel was designed with multiprocessors in mind, and is being ported to a Sequent shared-memory multiprocessor. As we will see below, its message-passing system and VM system fully address the problem of TLB inconsistency in shared-memory multiprocessors.

5.3.1.1. Object-oriented programming

We now briefly introduce some terms and concepts of object-oriented programming that we will use in presenting the design. Kernel data structures and procedures are represented as objects. An object has private data, public data, and a set of routines (called *member functions* in C++ terminology) that manipulate the object. Private data are invisible and inaccessible externally; public data and member functions together define the *interface* to the object. The definition of an object is specified by its *class*. A *derived* class may *inherit* properties from a *base* class, possibly redefining and adding some properties. Class inheritance allows code sharing. It also allows specialized implementations of a common interface. See [Str86] for a complete description of the object-oriented programming facilities used in DASH.

5.3.1.2. Local kernel structure

The structure of the DASH kernel is described in detail elsewhere [AnT88]. This section describes only several kernel properties that are essential to presenting the internal design of the VM system.

DASH supports multiple virtual address spaces (VASs). A VAS is a unit of protection and resource allocation. There is one kernel VAS and multiple protected user VASs. Each VAS can be populated by any number of processes, all of which have distinct kernel context blocks. DASH can be viewed as a message-passing kernel. User-level processes interact with the kernel (and with processes in other VASs) exclusively by message-passing. System calls, exceptions, and requests to user-level servers are all implemented as message-passing operations. Section 5.3.4 describes the user-level message-passing system; the next paragraph describes message-passing within the kernel VAS[2].

The kernel uses message-passing for interactions between various kernel components. Dynamically, the kernel is organized as a collection of trap handlers and processes (or threads) that share the whole kernel VAS. A trap handler or a kernel process communicates with other kernel processes via message-passing, as well as via the shared kernel VAS. For example, an I/O interrupt handler sends a message to a driver process on an I/O event; a buffer producer process delivers a buffer by sending it as a message, possibly being blocked by the flow control mechanism of the message-passing system. A message-passing operation may be a *pseudo* operation (called *uniprocess mode* in DASH terminology), in which case the send operation is carried out as a procedure call, passing the message as an argument. However, a *pseudo* message-passing operation has exactly the same interface as a regular message-passing operation; the sender cannot distinguish the difference. Table 5.1 shows the simplified interface to the message-passing system.

Finally, the kernel uses preemptive deadline-based process scheduling. Shared kernel objects are guarded by fine-grain locks. Concurrent kernel processes together with fine-grain locking establish a basis for kernel parallelism in a shared-memory multiprocessor.

[2] The overhead of intra-kernel message-passing is much lower than that of inter-VAS message-passing.

Table 5.1. Simplified interface to kernel's message-passing system. Message-passing operations are performed on message-passing objects, or MPOs. Class inheritance allows specialized implementation of a common interface.

Base MPO Class	Derived Class	Member Functions

stream MPO	dual-process mode	`send(message)`
		`receive(message_buffer)`
		`control(options)`

	uni-process mode	`send(message)`

request/reply MPO	dual-process mode	`request_reply(request, reply_buffer)`
		`get_request(request_buffer)`
		`send_reply(reply)`
		`control(options)`

	uni-process mode	`request_reply(request, reply_buffer)`

5.3.1.3. The DASH virtual memory system

Figure 5.3 depicts the overall structure of the DASH VM system [ATG88]. The top part of the picture shows the abstraction, or users' view, provided by the VM system. The system supports multiple VASs, each of which consists of three regions. The *general region* contains data that is private to a VAS, such as stacks and heaps. There is no sharing between VASs in this region. The *shared segment region* contains shared read-only named segments (*e.g.*, programs and libraries). Physical pages contained in these segments are shared between VASs, and may be retained even when no VAS is using them. The *IPC region* contains data to be moved between VASs using VM remapping. Messages to be moved between VASs are created, sent, and received in this region.

The middle and lower part of the picture show the VM implementation, which follows the portable model described in Section 5.2.1. The middle part is independent of hardware architectures and backing store services. It consists of a set of objects and processes, such as the zero-filling process. The lower part is hardware and backing store service dependent. The class VAS_MD encapsulates VM hardware architecture (MD stands for machine-dependent). Each instance of the class represents the virtual memory mapping of a single VAS. The class BACKING_STORE encapsulates backing store services. The implementations of these two classes vary with hardware and the nature of backing store services, but their interfaces do not.

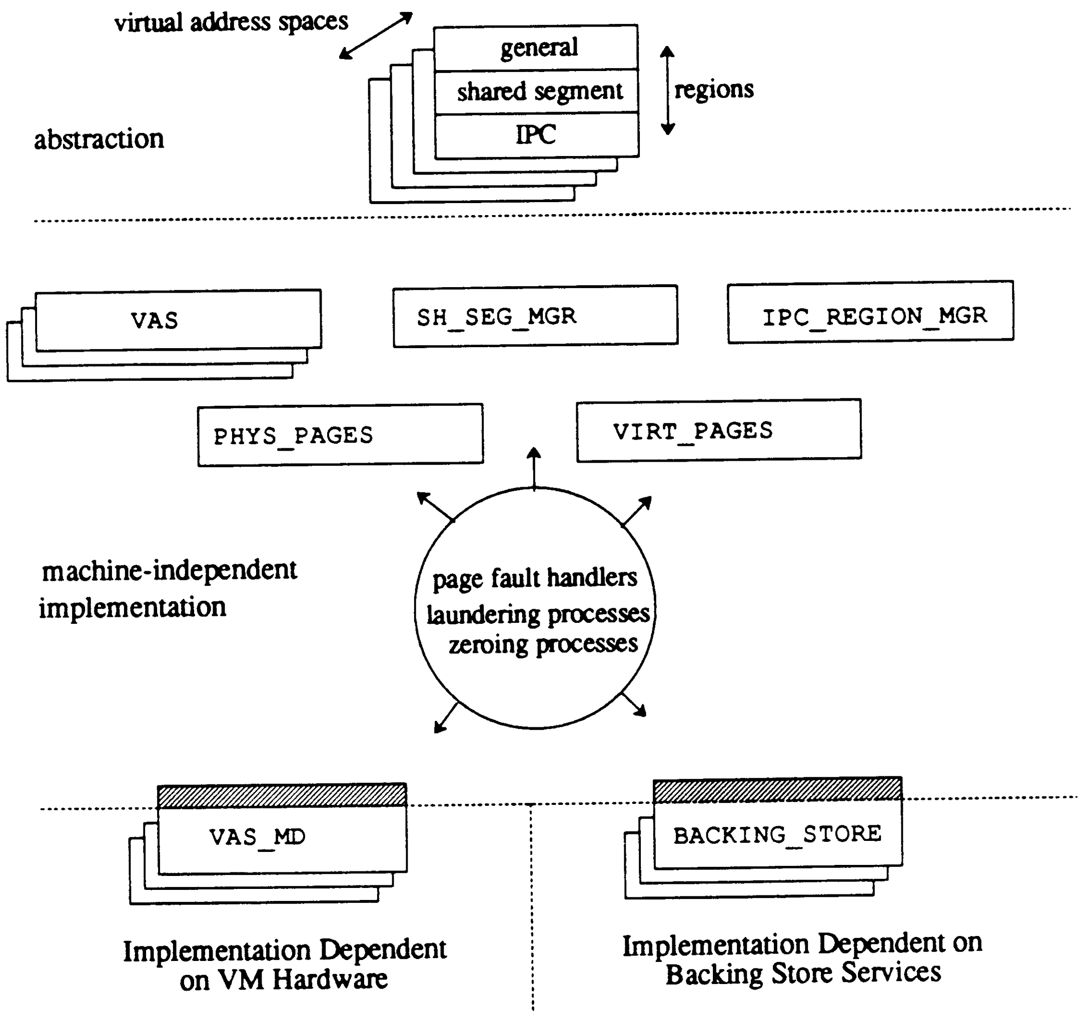


Figure 5.3. An overview of the DASH VM system.

5.3.2. Interface to the machine-dependent part

The interface to the `VAS_MD` class (lower left corner in Figure 5.3) provides a simple logical view of virtual memory mapping. It isolates machine-dependencies, but allows high-level software to exploit features of VM hardware. Sections 5.3.3 and 5.3.4 use this interface in describing paging and message-passing.

Table 5.2 summarizes the member functions of the `VAS_MD` class. We briefly explain each one below. The constructor `VAS_MD()` creates a new instance of the class either for the kernel VAS or for a user VAS, depending on the value of the flag. For the kernel VAS, `root_start` and `root_size` specify the portion of the kernel VAS that must exist in every user VAS. For example, one may choose to put the complete kernel in the lower or upper half of every user VAS, as in UNIX. Alternatively, one may choose to put only basic trap/interrupt handlers in every user VAS, and put the rest of the kernel in a separate map. The `VAS_MD` class does not assume a specific design; the interface is flexible.

The mapping represented by each `VAS_MD` object is specified by a sequence of `map()`, `unmap_synch()`, and `unmap_asynch()` operations on the object. The two `unmap` operations reflect the ideas described in Section 5.2.3. `hint` has two boolean fields: (1) `slow_flag` specifies that the latency of the operation is not critical, allowing the implementation of `VAS_MD` to make tradeoffs; (2) `conditional_flag` instructs the implementation to handle only simple cases, *e.g.*, if the optimistic algorithm fails after on round, it should return instead of starting another round. `unmap_asynch` allows the implementation to use asynchronous algorithms, which are usually more CPU-efficient. When an asynchronous operation ends, the specified message plus a return code will be sent to the specified MPO. If the MPO is a uniprocess mode MPO, the message-passing

Table 5.2. Simplified interface to the `VAS_MD` class. Member functions are invoked on `VAS_MD` objects.

Member Functions	Comments
`VAS_MD(kernel_flag, root_start, root_size)`	create a new `VAS_MD` object
`map(virt_addr, phys_add, access_type)`	map a page
`unmap_synch(virt_addr, readonly_flag, hint)`	unmap a page synchronously (unmapping includes reprotecting it to read-only)
`unmap_asynch(virt_addr, readonly_flag, stream_mpo, message, hint)`	unmap a page asynchronously
`switch_to()`	make this `VAS_MD` the active mapping
`is_dirty(virt_addr)`	is the page dirty?
`clear_dirty(virt_addr)`	clear the dirty bit of the page
`share(sharing_id, seg_start, seg_end)`	declare sharing
`config()`	return machine-dependent parameters
`set_page_fault_hdr(handler)`	setup high-level page fault handler

operation is equivalent to a procedure call (see Section 5.3.1.2 and Table 5.1).

A processor is executing in the context of only one VAS at any time, although there are multiple `VAS_MD` objects. We call this VAS the *active* VAS of the processor. The function `switch_to()` makes the `VAS_MD` object on which this function is performed active on the calling processor.

For demanding paging, `is_dirty()` and `clear_dirty()` check and clear the dirty bit of a page respectively. They are used for determining whether to write a paged out page back to the backing store. On the other hand, the interface does not support referenced bits for two reasons. First, we assume that, because of large physical memory sizes, page replacement will be infrequent. Attempts to make intelligent choices (*e.g.*, to approximate LRU) will not be needed; a random choice will probably be sufficient. It is possible that other readily available information (such as the recent CPU usage of processes in the VAS) may be useful in making a heuristic choice. Also, some page replacement algorithms do not use referenced bits, such as the VMS second-chance FIFO algorithm [LeL82]. Second, eliminating referenced bits simplifies the problem of TLB inconsistency. In general, the hardware may overwrite a PTE as a result of setting a referenced bit. The operating system can force the hardware not to set the referenced bit by setting the bit in every PTE. When a processor loads a PTE into its TLB, it will not attempt to set the reference bit in the PTE again, because the bit has already been set (see Section 2.1.3)[3].

`share()` is used for sharing among multiple VASs. Generally, page-level sharing can be done by mapping the same physical page into different virtual addresses in a VAS or into different VASs; this does not require special support in the interface to the `VAS_MD` class. However, certain VM architectures, *e.g.*, the one in the IBM RT PC [ChM88], have special hardware support for segment-level sharing. `share()` allows high-level software to declare segment-level sharing, thus allowing the implementation to exploit the underlying VM hardware. All `VAS_MD`s that have called `share` with the same ID will share the specified memory block, *i.e.*, a `map()` or an `unmap()` operation on any `VAS_MD` will affect every one of them. When ID is 0, the memory block is shared by all `VAS_MD`s; when ID is -1, sharing is canceled. The DASH shared-segment region uses this facility [Gov89].

Finally, there are two housekeeping functions. `config()` returns the machine-dependent parameters of the virtual memory system, such as the size of a page. `set_page_fault_hdr()` assigns a high-level page fault handler to be called by the machine-dependent part on a page fault. On a page fault, the low-level page fault handler extracts parameters from the stack and calls the machine-independent page fault handler.

[3] Similarly, TLB synchronization would be even simpler if we eliminate dirty bits. The software can either emulate the dirty bit by protecting all pages as read-only first. Or, assuming that the paging rate is low, it can treat all writable pages as dirty when they are paged out. We will in the future remove these two functions (`is_dirty()` and `clear_dirty()`) from the interface if the implementation experience shows that they have little performance benefit.

5.3.3. Paging

This section describes the design of demand paging in the DASH VM system, with emphasis on TLB inconsistency. In short, we reduce the overhead of synchronizing TLBs by tolerating transient TLB inconsistency (or by exploiting asynchrony). Managing transient TLB inconsistency is straightforward in our design because (1) the latency of unmapping a page is not critical, (2) the `VAS_MD` class has an asynchronous interface (see Section 5.3.2), and (3) the kernel is programmed as multiple processes (see Section 5.3.1.2).

5.3.3.1. Overview: flow and states of physical pages

Figure 5.4 summarizes the design by showing the flow of physical pages in the system. Background kernel processes move physical pages from the `in_use_list` to the `clean_list`, and from the `clean_list` to the `zero_list`[4]. In the other direction, a physical page becomes `in_use` when it is reclaimed (*i.e.*, referenced when it has been selected to be paged out but still contains correct data), or explicitly allocated.

The overhead of replenishing the `clean_list` and the `zero_list` is much higher than the overhead of emptying them. To ensure that requests for a physical page are satisfied promptly, the system maintains pools of free pages in the `clean_list` and the `zero_list`. (The BSD UNIX virtual memory system use similar ideas, though it has a different design; see [BaJ81].) This approach not only improves response times, but also removes page unmapping from the critical path of virtual memory management. Consequently, the latency of unmapping a page is not critical.

The state of a physical page can be `in_use`, `clean`, `zeroed`, `being_read`, `being_written`, or `being_unmapped`. The first three correspond to the three lists described above; the last three represent intermediate states in which an action is undergoing. States are manipulated by high-level software. The `VAS_MD` class does not change the state of a page.

Below, we concentrate on page-out operations. Page-in operations cause only safe TLB inconsistency; zero-filling does not cause TLB inconsistency.

5.3.3.2. The unmapper process

Page-out operations are handled by an unmapper process and several launderer processes. These processes interact with each other via a `launderer_mpo` MPO. When the size of the `clean_list` is below a threshold, the unmapper process randomly[5] selects a page from the `in_use` list. It changes the state of the page from `in_use` to `being_unmapped`, and calls (see

[4] The names are self-explanatory: `clean` means a page has not been modified since it was paged in; `zero` means a page is zero-filled.

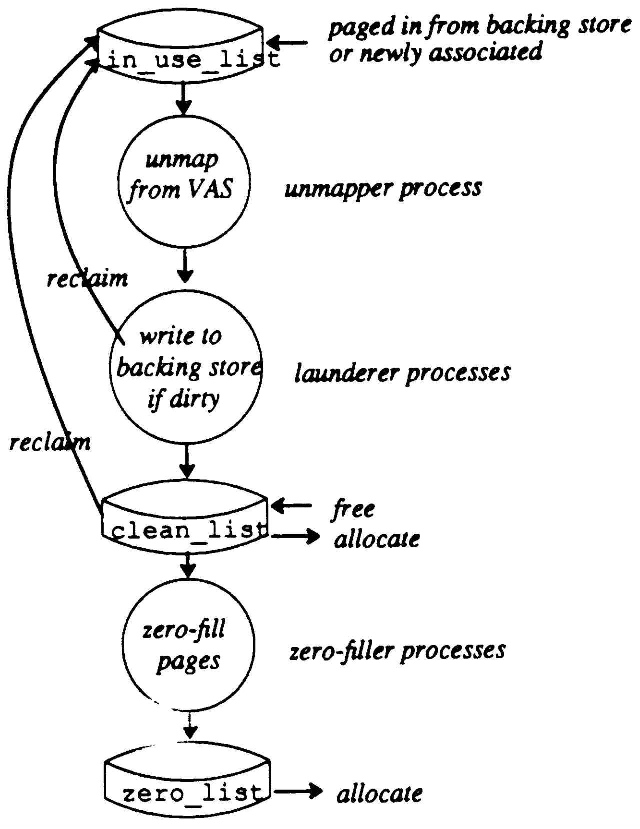


Figure 5.4. The flow of physical pages.

Section 5.3.2)

```
VAS_MD::unmap_asynch(
     virt_addr,
     ~readonly_flag,         // invalidate the page
```

[6] Other algorithms are also possible. The issue here is not page replacement algorithms, but TLB inconsistency.

```
                slow_flag | conditional_flag,
                launderer_mpo,
                return_message        // allocated by the unmapper
            )
```

The `slow_flag` indicates that latency of this operation is not critical, as explained in Section 5.3.3.1. The `condition_flag` instructs the implementation not to handle special conditions, most of which occur only when the page is referenced during the unmap operation. These two flags together instruct the `VAS_MD` object to use the most CPU-efficient TLB synchronization algorithm. Later, when the unmap operation ends, the `return_message` will be delivered asynchronously by the `VAS_MD` object to `launderer_mpo`.

5.3.3.3. Launderer processes

A launderer process writes dirty pages to their backing store using synchronous writes. To allow multiple backing store write operations to proceed concurrently, there are multiple launderer processes waiting on the `launderer_mpo` object.

A launderer process handles a physical page in different ways according to the state of the page and the return code stored in `return_message`. It moves the page back to the `in_use_list` if (1) the state of the page has been changed from `being_unmapped` to `in_use` by the page fault handler; or (2) the unmap operation failed, *e.g.*, the PTE has been overwritten as a result of setting the dirty bit. Otherwise, the launderer process calls `VAS_MD::is_dirty()` to determine whether the page has been modified. It moves clean pages to the `clean_list`, and writes dirty pages to the backing store after changing their state to `being_written`. When the write operation ends, the launderer process moves the page to the `clean_list` if the page has not been referenced during the write operation (*i.e.*, no page faults during this period); else it moves the page to the `in_use_list`.

5.3.3.4. Managing TLB inconsistency

Transient TLB inconsistency may occur when the state of a page is `being_unmapped`. More precisely, `unmap_asynch()` may invalidate a PTE before it has invalidated all corresponding TLB entries. Moreover, stale TLB entries may exist for a long period of time (say, 100 ms) because `unmap_asynch()` usually invalidates them in batches.

Transient TLB inconsistency is harmless in our design. The launderer processes will not take any action on a physical page whose state is `being_unmapped`; a message is delivered to the `launderer_mpo` after the `unmap_asynch()` is completed. In other words, although a physical page is no longer linked to its old virtual page in page tables, the content of the page remains unchanged as long as its state is `being_unmapped`. If a process accesses the old virtual page via a stale TLB entry, it will still get to the same physical page. Therefore, accessing a page using a stale TLB entry yields correct results.

We now enumerate all possibilities of accessing a page using a stale TLB entry. Assume that a physical page `ppage` in state `being_unmapped` was mapped to a virtual page `vpage`. The following cases are possible when a process `p` accesses `vpage` on processor `P`.

- If the TLB entry for `vpage` has been invalidated on `P`, a page fault occurs. The page fault handler calls `VAS_MD::map(vpage, ppage)` on the faulting processor, changes the state of `ppage` to `in_use`, and resumes the faulting process. A launderer process will move this page to the `in_use_list` later (see Section 5.3.3.3).
- Otherwise, the memory reference is granted (accessing `ppage`). If this memory reference overwrites the PTE as a result of setting the status bit, the unmap operation will detect this after one round and return an error code. Again, a launderer process will put this page in the `in_use_list` later.
- Otherwise, if the reference does not overwrite the PTE, it has no effect at all. The page will be unmapped as if the reference had not happened.

In all three cases, accessing `vpage` yields correct results.

5.3.4. Message-passing using virtual-memory remapping

DASH integrates virtual memory and message-passing, using VM remapping to transfer large messages between VASs (small messages are still copied). Our design is intended not only to reduce software copying, but also to reduce the overhead of synchronizing TLBs on shared-memory multiprocessors. Managing TLB inconsistency in message-passing is more challenging than in paging, because the latency of remapping operations is critical. We tolerate TLB inconsistency by exploiting trust relationships and asynchrony.

Section 5.3.4.1 explains why software copying is undesirable. Section 5.3.4.2 shows our initial experience with using remapping on a uniprocessor. Section 5.3.4.3 shows the software architecture of the message-passing/VM system. Section 5.3.4.4 describes the IPC region of the VM system, on top of which the message-passing system is built. Section 5.3.4.5 describes the semantics and interface of message-passing operations, emphasizing VM related aspects. Finally, Section 5.3.4.6 examines the design from the point of view of TLB inconsistency.

5.3.4.1. Software copying vs. VM remapping

Virtual memory remapping is a class of techniques for logically moving or copying a page of data from one VAS to another. Remapping is an attractive alternative to software memory copying because updating a PTE is much faster than copying a page on most machines.

It has long been known that interprocess communication (IPC) systems should avoid unnecessary software copying of memory. Copying may be done in communication protocols for retransmission, in data transfer between user and kernel VASs, and in data transfer between two user VASs on a single host [CHK88, WaM87].

With current technological trends, copying is becoming a more severe bottleneck. Communication technology, particularly fiber optics, is advancing rapidly [BKN89, Lei88]. Gigabit bandwidths exist at the link level, but a variety of bottlenecks prevent user processes from fully exploiting this bandwidth. The system bus of the host computer is often such a bottleneck, since it limits the rate at which data can be moved between the network interface and main memory [Wil87].

Memory copying is especially undesirable because it is bus-intensive. For high-bandwidth data (*e.g.*, real-time video), copying always produces heavy traffic on the system bus, even when the system has cache memory or does I/O directly to or from cache. This traffic slows down DMA devices and the computations of other CPUs.

The copying problem is exacerbated by the trend in operating system design towards moving data servers (file servers, transaction managers, etc.) from the kernel to user VASs. Some examples are V [ChZ83, Che84], Ridge [Bas85], and QuickSilver [HMS88]. A data access in such a system usually involves several data movements between VASs. A file request from a user client process to a user-level file server might be routed through a transaction manager and a network communication manager at each end. If copying is used to move data between VASs, this organization amplifies the negative performance impact of copying.

5.3.4.2. Initial experience of VM remapping on a uniprocessor

Our initial experience shows that VM remapping is advantageous on a uniprocessor. This section gives some performance numbers about DASH running on a Sun 3/50 workstation. (More numbers can be found in [TzA88].) The purposes of presenting these numbers, although they were not obtained on a shared-memory multiprocessor, are (1) to show the relative performance of remapping vs. copying, (2) to show the overhead of other operations associated with remapping, and (3) to preview the design in later sections. Most of the design for tolerating TLB inconsistency has been implemented in the current DASH kernel. Hence we hope that we can achieve similar performance gains on a shared-memory processor as we have on the Sun 3/50 workstation.

Figure 5.5 shows the throughput of data movement from one VAS to another using message-passing on a Sun 3/50 workstation running DASH. The throughput increases with message size, because the weight of the fixed per-message overhead decreases when the number of pages in a message increase. The two horizontal lines represent the throughput of pure software copying (per-message overhead is excluded). In some operating systems, such as UNIX, moving a page between two VASs requires copying it twice (from sender to kernel, and from kernel to receiver). DASH exploits *lazy remapping*, a principle

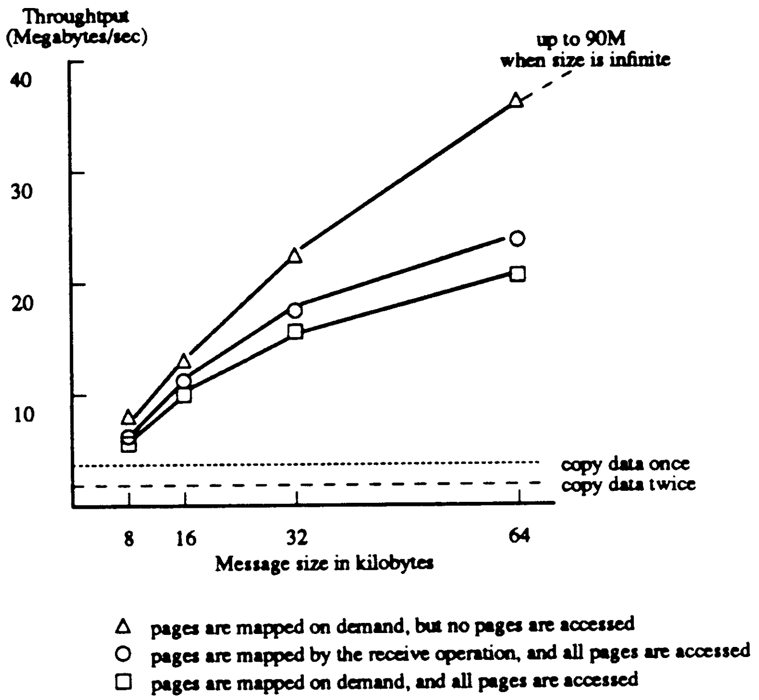


Figure 5.5. Throughput of message-passing: initial experience. The numbers were obtained on a Sun 3/50 workstation running DASH.

that delays operations as long as possible. The throughput depends on whether pages are remapped immediately or on demand, and whether the remapped pages are accessed or not.

Figure 5.6 shows the µs-level cost breakdown of sending a message containing an 8KB page from one user VAS to another on a Sun 3/50 workstation. Individual operations will become clear as we explain the design in later sections; this picture serves as a preview for now. Using VM remapping involves more than just updating page tables. The operating system may have to manage buffers, adjust data representation to reflect address changes, manipulate high-level memory maps, etc. The DASH design intends to reduce these overheads, and has been successful in its Sun 3/50 implementation.

5.3.4.3. An overview of the DASH message-passing system

Figure 5.7 shows the overall software architecture of the message-passing system. It complements Figure 5.3 by showing layers and interfaces of the

- 1231 Total elapsed time
  - 641 User-level send operation
    - 34 trap into and return from kernel mode
    - 49 switch to and from the kernel VAS
    - 15 convert and check user object references
    - 46 check the message header
    - 7 dispatch to the send operation
    - 490 kernel-level send operation
      - 138 processing before message transfer
      - 221 transfer the message
        - 74 copy the message header
        - **19 check a page descriptor in the message**
        - **81 transfer a page between VAS's**
          - **32 look up and check memory maps**
          - **26 invalidate a PTE (needed only if the page has been mapped)**
          - **23 miscellaneous**
        - 47 miscellaneous
      - 131 processing after message transfer
  - 296 User-level receive operation
    - 33 trap into and return from kernel mode
    - 46 switch to and from the kernel VAS
    - 16 convert and check user object references
    - 46 check the header of the receive message
    - 6 dispatch to the receive operation
    - 98 kernel-level receive operation
      - 91 processing before the process is blocked
      - 7 processing after the process is awakened
    - 51 complete the message transfer
      - 24 copy header from temporary buffer to receive buffer, if needed
      - **16 ensure that the page transfer is completed**
      - 11 miscellaneous
  - 120 Context switch between user processes
  - 173 Access the received message
    - 37 find the address of data in a structured message
    - **134 handle a page fault**
      - **24 save states and get parameters of the fault**
      - **55 look up and check memory maps**
      - **32 update a PTE**
      - **23 return from fault**
    - 2 miscellaneous

Figure 5.6. Cost breakdown of message-passing: initial experience. Numbers are elapsed times (in μs) for sending an 8KB message between VASs on a Sun 3/50. Operations in boldface are performed for every page in the message, while the others are independent of the size of the message.

system.

From a user's point of view, the message-passing system consists of two basic elements: (a) messages to be passed, and (b) operations that move messages.

For (a), DASH has a standard message representation used by both the kernel and user programs. A message is not necessarily physically contiguous: it is represented as a header, which contains control information, up to 4KB of data, and optional pointers to separate data pages. When a message is passed between VASs, the header is copied, whereas separate data pages are remapped. The sender and receiver of a message manage the buffer for the header; the kernel manages remapped pages.

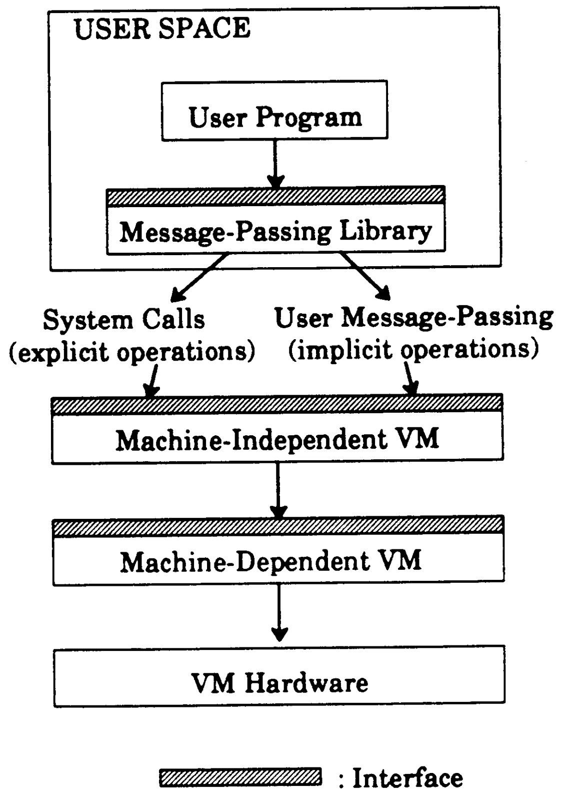


Figure 5.7. Software architecture of the message-passing system.

For (b), DASH is essentially a message-passing kernel. A user process makes kernel requests by trapping into the kernel. The trap handler directly handles only message-passing operations (Section 5.3.4.5 discusses these operations). Other "system calls", *e.g.*, a request to create a process, are invoked by sending a message to a special message-passing object in the kernel. To the trap handler, a system call is just a regular message-passing operation.

A set of library routines provides user processes with a better interface to the message-passing system. For example, some routines encapsulate the message representation as a logical byte array; some prepare parameters and invoke kernel traps.

The message-passing system uses the VM system to move data pages. Hence, a message-passing operation may implicitly invoke a VM operation. Alternatively, a user process may explicitly invoke a VM function by making a system call (see Figure 5.7). The VM system, particularly its IPC region, provides high-level support for the message-passing system. The machine-independent VM system is built on top of the `VAS_MD` class (see Section 5.3.2) for controlling VM hardware.

We will not present the complete message-passing system here (see [AnT88] for details), but will concentrate on aspects related to VM remapping and TLB inconsistency. Specifically, the following sections will describe (1) the IPC region of the VM system, and (2) how message-passing operations accommodate hooks for tolerating TLB inconsistency.

5.3.4.4. The IPC region of the VM system

The *IPC region* contains page-size buffers that can be moved between VASs by VM remapping. It is managed by the `IPC_REGION_MGR` class, a layer between the machine-dependent part of the VM system (the `VAS_MD` class, see Section 5.3.2) and the message-passing system. A virtual page in the IPC region is called an *IPC page*. All data to be moved between VASs without copying must be placed in IPC pages.

For managing TLB inconsistency, the `IPC_REGION_MGR` class supports asynchronous VM remapping functions. These functions are similar to those of the `VAS_MD` class (Section 5.3.2), but are more abstract.

For efficient implementation[6], the `IPC_REGION_MGR` class allows only a restricted form of VM remapping. First, an IPC page must be remapped to the same virtual address in the destination VAS as in the source VAS. This ensures that pointers to IPC pages remain unchanged after remapping, eliminating the need for data representation adjustment. Second, there is a single "meta-level" mapping from IPC pages to real pages, and each VAS sees a subset of this

[6] Design decisions described in this paragraph are orthogonal to tolerating TLB inconsistency. They were made for performance optimization. These decisions improve the efficiency of implementation, but also limit the flexibility of using VM remapping; in our system, VM remapping is only used for data movement.

mapping[7]. Consequently, a virtual IPC page in different VASs will not hold different data, *i.e.*, it will not be mapped to different real pages at the same time. This, together with the first rule, simplifies buffer management. When the kernel needs to remap an IPC page from one VAS to another, the corresponding virtual IPC page in the receiving VAS is guaranteed to be available. In other words, buffer management is straightforward.

Based on the above two rules, we define the notion of *page ownership* for protection and remapping of IPC pages. An ownership has three elements: an owning VAS, a virtual page, and an access right. An IPC page may have multiple ownerships, *i.e.*, it may be owned by multiple VASs, or by one VAS multiple times, or both. The number of ownerships of a page determines the access right to it. When the number is one, the page can be both read and written by its owner (*i.e.*, by processes in the owning VAS); otherwise the page can only be read by its owners. An IPC page may be transferred from a source VAS to a destination VAS; the source loses an instance of ownership and the destination gains it. A VAS may also duplicate its ownership of an IPC page, *e.g.*, before sending the page out.

Page ownership can also be viewed as a high-level VM map maintained by the `IPC_PAGE_MGR` (whereas a `VAS_MD` object represents a low-level VM map). The splitting of a VM map into two parts enables *lazy remapping*, a mechanism that defers VM map changes whenever possible. Updating the low-level VM map is more expensive than updating the high-level map, partly due to TLB synchronization. With lazy remapping, a page is mapped into the low-level map by the page fault handler only when it is referenced. Lazy evaluation saves a pair of low-level `map` and `unmap` operations if a page is mapped into and out of a VAS without being accessed, but incurs the extra overhead of a page fault if the page is accessed. It is beneficial for applications that forward large amounts of data, such as user-level file server and network server.

We now explain member functions of the `IPC_REGION_MGR` class (see Table 5.3 for a summary). `get_ownership()` allocates a new IPC page to a VAS. `release_ownership()` releases an instance of ownership of an IPC page from a VAS.

`start_transfer()` starts transferring an instance of IPC page ownership from `source_vas` to `dest_vas`. It returns after the high-level VM maps have been updated, but not necessarily after the low-level VM maps have been. When `ownership_only_flag` is set, the low-level map is either updated using the most CPU-inexpensive way, or simply ignored. The `IPC_REGION_MGR` keeps track of the status of IPC pages, including what action has been taken on the low-level VM map.

[7] This differs from pure sharing. Each VAS still has a separate VM mapping, so the kernel can enforce protection and security on a VAS-by-VAS basis.

Table 5.3. Interface to the `IPC_REGION_MGR` class.

Member Functions	Comments
`get_ownership(vas, virt_addr_p)`	allocate a new IPC page
`release_ownership(vas, virt_addr)`	free an IPC page
`start_transfer(source_vas, dest_vas, virt_addr, ownership_only_fg)`	start remapping a page
`finish_transfer(source_vas, virt_addr)`	wait for completion of remapping
`duplicate(vas, virt_addr, flags)`	increase page ownership by one
`make_writable(vas, virt_addr, new_addr_p)`	make the page writable
`mapin(vas, virt_addr, access)`	map into the low-level VM map

`finish_transfer()` is the counterpart of `start_transfer()` or `duplicate()` (see the next function). It blocks until `virt_addr` has been completely unmapped (including reprotection) from the low-level VM map of `source_vas`. If the corresponding `start_transfer()` has the `ownership_only_flag`, it synchronously unmaps the page from the low-level VM map.

`duplicate()` increases the ownership of `virt_addr` in `vas` by one. If the access right to the page was read/write, it is changed to read-only. `flags`, same as that in `start_transfer()`, specifies how the reprotection is done. This function is also asynchronous; `finish_transfer()` ensures that the reprotection started by it is completed.

`make_writable()` makes an IPC page writable. If the page has multiple ownership, it is copied to a new page.

`mapin()` maps `virt_addr` into the low-level VM map of `vas` for `access`. By default, `start_transfer()` does not map a page into the low-level VM map of a VAS in order to exploit lazy evaluation. This function is called by the page fault handler on the first reference to `virt_addr`, or to override lazy evaluation on the receive operation.

5.3.4.5. Message-passing operations

User-level message-passing operations are essentially the same as intra-kernel message-passing operations (see Section 5.3.1.2 and Table 5.1), except that they are extended with

- mechanisms allowing user processes to invoke kernel member functions across the user-kernel boundary, and

- semantic rules defining inter-VAS data movement, protection, and trust.

This section describes both extensions (only the second one is related to managing TLB inconsistency).

To invoke a member function on a kernel message-passing object (MPO), a user process uses (1) the *user object reference* (UOR) facility for referencing the kernel MPO, and (2) the trap mechanism for passing control and parameters. Each VAS has an associated set of UORs, small integers that act as capabilities to kernel objects (see Figure 5.8). UORs are related to (but more general than) UNIX file descriptors, Mach port capabilities, and so on. A message-passing operation is started by loading arguments (including a UOR to an MPO) into registers, and executing a trap instruction. Then the trap handler, after collecting and checking arguments, invokes a member function on the target MPO on behalf of the user process.

We now describe the second extension--semantic rules for inter-VAS data movement, protection, and trust. For simplicity, in the following discussion, a *sender* refers to a user process that delivers a message using message-passing,

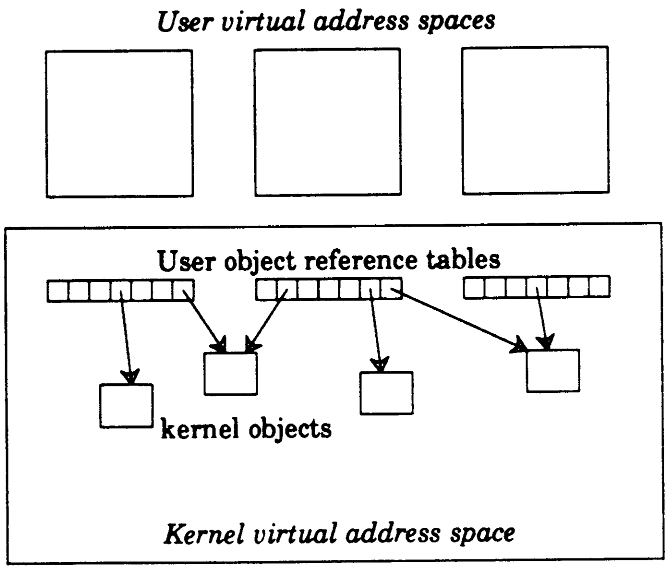

User virtual address spaces
User object reference tables
kernel objects
Kernel virtual address space

Figure 5.8. User object reference tables. User processes use indices to per-VAS UOR tables to reference kernel objects.

and a *receiver* refers to a user process that obtains a message. In some cases, such as in `request_reply()` a process can be the sender of a message and the receiver of another message at the same time. We also use *send operation* and *receive operation* in a similar way in our discussion.

A message may be transferred between VASs in two modes, depending on the `duplicate` flag set by the sender. When this flag is off, the sender loses all data pages in the message and the receiver gains them. In other words, the sender must not access these pages after the message-passing operation. When this flag is on, both the sender and the receiver get a read-only copy of data pages. (This is indicated by flag bits, one per page, in the message header). Either one may then modify its own copy independently, but must inform the kernel before doing so (using `make_writable()` in Table 5.3). If a sender violates the above semantic rules after sending out a message it will (1) generate an exception, (2) damage the content of its own VAS, or (3) read incorrect data.

Note that the second transfer mode described above is functionally equivalent to the copy-on-write mechanism, except that pages are copied via explicit system calls rather than by the page fault handler (see Section 5.3.5 for more discussion). Since user processes access messages via library routines, calling `make_writable()` is transparent to user programs in most cases.

The kernel, in addition to enforcing the above rules, protects every user VAS against memory accesses from processes running in other user VASs. Specifically, after a receiver has received a message (*i.e.*, after the receive operation returns), the kernel guarantees that (1) no processes running in other VASs can modify the message, and (2) no processes running in other VASs can read the message beyond what they had already read. Therefore, if the message is moved using VM remapping, the kernel must invalidate all stale TLB entries that may be used to violate such protection. A receiver may simplify the kernel's job by setting the `trust` flag in the receive operation. This indicates that the receiver trusts the sender not to violate the sender's semantic rules, even when illegal accesses may be granted because of the existence of stale TLB entries.

In addition to the above flags, the message-passing system takes hints. A sender may set the `trust_hint` flag, indicating that it believes the receiver will trust it. This flag is useful when the send operation precedes the corresponding receive operation. If the hint is incorrect, the OS can still do necessary work before the receive operation returns, though the latency of the receive operation will be longer . A receiver may set the `immediate_use` flag (separate bits for read and write), indicating that it will access the received data. The system uses this flag to turn off lazy evaluation (see Section 5.3.4.4) to save a page fault.

5.3.4.6. Managing TLB inconsistency

This section explains how user-level message-passing operations and parameters are translated into functions on the `IPC_REGION_MGR` class, and how TLB inconsistency is handled.

An IPC page is always remapped in two steps. A send operation starts page remapping by calling start_transfer(). The ownership_only_flag of this function is determined by the trust flag of the corresponding receive operation if it precedes the send operation. Otherwise the ownership_only_flag is determined by the trust_hint flag of the send operations. start_transfer() may trigger non-blocking TLB synchronization; the kernel can do other work, such as waking up and rescheduling the corresponding receiver, while the remapping is in progress. Such asynchrony is very useful on a multiprocessor because it allows TLB synchronization and scheduling to be done concurrently.

A receive operation, when its trust flag is false, calls finish_transfer() before it returns to ensure that all IPC pages in the received message have been properly remapped. This call is skipped if the trust flag is true. If the immediate_use flag is true, a receive operation calls mapin() to turn off lazy evaluation.

Our design reduces the needs for TLB synchronization in two ways:

- With receiver's trust, updating the low-level VM map of the sender's VAS becomes unnecessary (reflected by the ownership_only_flag flag). Although the VM system no longer acts as a firewall, the correctness of the system is guided by the semantic rules of the message-system, which forbids a sender to take advantage of stale TLB entries.
- By exploiting lazy evaluation, the kernel avoids as much as possible updating low-level VM maps and consequently avoids synchronizing TLBs. Lazy evaluation causes inconsistency between the high-level and low-level VM maps. Such inconsistency is similar to the *safe TLB inconsistency* discussed in Section 5.1.1, because low-level maps always allow more restricted access rights than the corresponding high-level maps.

When TLB synchronization is necessary, we do it efficiently by exploit asynchrony. We overlap the latency of a TLB synchronization operation with the delay between a send and the corresponding receive operation. Thus, we can either reduce the overall message-passing latency, or use more CPU-efficient TLB synchronization algorithms (such as the optimistic-synch algorithm). Asynchrony may cause transient TLB inconsistency. But as stated in Section 5.1.2, such inconsistency is harmless as long as it is removed before the receive operation returns.

5.3.5. Mechanisms not suitable for tolerating TLB inconsistency

In addition to paging and message-passing, many systems remap memory pages under the copy-on-write mechanism [ABB86, BBM72, RTY88]. This section explains why we do not have a design that tolerates TLB inconsistency under this mechanism, and sketches alternatives.

It is hard to tolerate TLB inconsistency if a VM mechanism requires the operating system to generate a fault on a legal[8] memory reference. Copy-on-write is such an example. Although a write reference to a copy-on-write page is legal, it must generate a protection fault. When a page is reprotected as copy-on-write (*i.e.*, reprotected from read/write to read-only), all stale TLB entries allowing write access to this page must be invalidated. Otherwise, subsequent write references might not generate a page fault, breaking the copy-on-write mechanism. Further, it is difficult to establish semantic rules to prevent a user from making memory references that may use stale TLB entries, because writing to a copy-on-write page is legal.

Therefore, copy-on-write is not necessarily beneficial for shared memory multiprocessors or for machines with virtually tagged caches[9], considering the high overhead of TLB synchronization. Nelson *et al.* also pointed out the high overhead of implementing copy-on-write on machines with virtually tagged caches, and proposed a revised scheme called COR-COW [NeO88].

DASH does not support copy-on-write at all. Instead, it provides other mechanisms to speed up process creation, the most important usage of copy-on-write in existing UNIX-like systems. First, DASH has a different process creation paradigm that reduces the need for copying a whole VAS. It allows multiple processes to run in the same VAS. Most processes are created in an existing VAS, rather than in a separate new VAS containing the same memory image as the parent process. In DASH, a new VAS is usually created as an empty address space. Second, the shared-segment facility supports read-only sharing among different VASs, reducing the overhead of copying program text into every VAS. See [Gov89] for details of this facility.

[8] *Legal* means the reference is allowed by the high-level mapping of the VAS.

[9] A virtually tagged cache can be viewed as a distributed TLB; see Section 2.3.

Chapter 6
Related Work

This chapter reviews related work in three areas: software mechanisms for TLB consistency, hardware mechanisms for TLB consistency, and performance evaluation of TLB synchronization.

6.1. Software Mechanisms

The performance of TLB synchronization mechanisms is not critical in most commercial shared-memory multiprocessors, because they typically have a small number of processors (10 to 20) and remap pages at a low rate. Their operating systems, most of which are UNIX-based, reduce[1] the access rights to a page only on page replacement and virtual address space (VAS) shrinking; they do not use remapping to move data between VASs. The paging rate is low if the system has sufficient physical memory; the frequency of VAS shrinking is also low. More important, the latency of these operations is not critical. Hence most systems achieve TLB consistency with reasonable CPU overhead simply by batching TLB synchronization operations together [FHM87]. The optimistic-asynch algorithm evaluated in Chapter 4 represents this technique.

The MIPS multiprocessor system employs an extra technique to manage TLBIDs, VAS identifiers associated with TLB entries [MMM86, TBJ88]. The operating system does not flush the entire TLB on context switches. It allows a processor to retain TLB entries for VASs other than the currently active one. For determining which processors might contain a stale TLB entry for a target PTE, the operating system keeps track of the migration history of VASs in per-space bitmaps. This technique is useful for machines that support VASs identifiers either in caches or in TLBs. The operating system also manages other details such as the overflow of VAS identifiers. The MIPS system loads TLB entries by software, so the TLB synchronization algorithm is simpler than in the general case (see Chapter 3 and Table 3.1).

Synchronizing virtually tagged caches on page remapping (which is equivalent to synchronizing TLBs) could be expensive. Cheng measured the overhead of flushing the virtually tagged cache on Sun-3 Series 200 workstations running SunOS [Che87]. Depending on the application, the percentage of total CPU time spent on cache flushing ranges from 0.13% to 3%. Nelson and Ousterhout also pointed out the performance problem of flushing virtually tagged caches in implementing copy-on-write, and proposed an improved COW-COR mechanism [NeO88]. The latency of TLB synchronization is important for

[1] Increasing access rights to a page is safe, and is thus ignored here.

copy-on-write[2], so it is not appropriate to defer operations as in page replacement. Section 5.3.5 discusses copy-on-write further.

Mach is the first system that uses the 2-phase TLB synchronization algorithm, or the *TLB shootdown algorithm* in their terminology [BRG89]. Black *et al.* measured the performance of this algorithm on a 16-processor Encore Multimax, and concluded that this algorithm would perform well even for systems containing hundreds of processors. However, all the applications that they have measured caused less than 0.5 TLB synchronization operations per processor per second. Hence our work and theirs address the same problem but for different workloads; we emphasize scalability but they do not. An earlier paper on the Mach VM system also mentioned allowing temporary TLB inconsistency in cases where it does not cause problems (*e.g.*, when protection is being increased) [RTY88]. This technique corresponds to tolerating safe TLB inconsistency in our work (see Section 5.1.1).

Rosenburg proposed a simpler TLB synchronization algorithm for a Mach port onto the IBM RP3 [EGL85, PBG85, Ros89]. This algorithm is similar to the PTE-first algorithm described in Section 3.2.2. RP3 processors do not write dirty or referenced bits back to PTEs. (The operating system obtains such information by initially turning off the valid bit to cause a page fault on the first reference.) In other words, a newly updated PTE will never be overwritten as a result of setting status bits. Consequently, it is not necessary to stall processors as in the 2-phase algorithm, and the TLB synchronization algorithm becomes much simpler. Rosenburg reported performance improvements of his algorithm over the 2-phase algorithm.

6.2. Hardware Mechanisms

MIPS-X-MP avoids the TLB inconsistency problem by using a centralized TLB [ChH87, HeH86]. Without multiple TLBs, changing virtual memory mapping is as straightforward as in a uniprocessor. This solution, although simple, does not scale well because the centralized TLB would become a bottleneck when the number of processors is large. The Stellar graphics minisupercomputer, a commercial system with a small number of processors, also takes this approach.

SPUR eliminates separate TLBs by using an in-cache address translation mechanism [Rit85, TzS85, WEG86]. With virtually tagged caches, virtual-to-physical address translation is not needed on a cache hit. Therefore, separate TLBs are no longer critical to the system's performance. The system treats page tables as regular data and caches them in data caches. When a PTE is modified, the cache coherency mechanism ensures that all cached copies of that PTE are consistent, achieving "free" PTE coherency. However, as explained in Section 2.3, every cache line stores some information in the corresponding PTE, at least protection bits. Hence this approach does not solve the TLB inconsistency

[2] For example, in UNIX `fork()`, neither the parent nor the child process can continue before TLB synchronization is completed. Otherwise, the parent process might write to a page without causing a page fault.

problem; it eliminates separate TLBs, but turns every cache line into a distributed TLB entry. Nelson and Ousterhout also pointed out the problem of virtually tagged caches [NeO88].

The Motorola MC88200 cache and memory management unit (CMMU) can be directly controlled by multiple MC88100 CPUs, or even by other CMMUs [MMM88a, MMM88b]. The control registers of each CMMU occupy 4K of the control memory space, and can be accessed by any CPU via regular memory references. Thus, a single CPU can flush the TLB entries on all CMMUs without interrupting other CPUs. This approach potentially simplifies TLB synchronization, although concurrency control for the control memory space is still necessary.

In the IBM System/370 architecture, a processor broadcasts a hardware TLB invalidation signal to all processors on an instruction that updates a PTE [Liu89]. The instruction does not complete until all TLBs have been synchronized. This scheme greatly simplifies the software at the expense of hardware complexity. However, a broadcast-based mechanism often causes contention in the interconnection network when the number of processors is large [PfN85]. (System/370 currently includes configurations with no more than 6 processors.) To be scalable, this scheme could be extended to incorporate techniques used in scalable cache-coherent multiprocessors, such as directory-based invalidation [ASH88, WeG89]. However, the hardware expense would be high.

Teller *et al.* proposed three hardware mechanisms for TLB consistency in highly parallel machines [TKS88]. The first solution is to lock a PTE using a hardware reference count, which represents the number of processors having the PTE loaded in their TLBs. The hardware allows a PTE to be modified only when its reference count is zero. This is a passive solution—it does not remove stale TLB entries when the software needs to update a PTE, but restricts what PTEs the software can update. It also limits the size of TLBs; if the total number of TLB entries (size of a TLB times the number of processors) is larger than the total number of physical pages, the operating system might find every PTE in a TLB entry (*i.e.*, every physical page locked).

Teller's second solution is to use version numbers, similar to the idea used in [Smi86] and [Emb87] for multiprocessor cache coherency. Each physical page is associated with a version number stored in a special table in the memory subsystem. For a processor, the version number of a page is treated as part of the physical address of the page; it is loaded into the TLB with the physical address and it sent out on a memory reference. When the operating system updates a PTE, it also increases the corresponding version number stored in the memory subsystem. A subsequent memory reference based on a stale TLB entry can be detected by comparing the version number stored in the memory subsystem and that sent out by the processor. This solution increases processor to memory traffic for sending version numbers, uses extra memory for storing version numbers, and requires a more complicated and hence potentially slower memory subsystem. Moreover, this scheme makes caching difficult. If the cache does not store version numbers at all, a memory reference based on a stale TLB entry may get a cache hit. Similarly, if the cache does store version numbers but the operating does not invalidate the cache when it updates a PTE, a memory reference based on a stale TLB entry may still get a cache hit.

Teller's third solution is to translate addresses in the memory subsystem instead of in processors. This solution has the effect of a centralized TLB as in MIPS-X-MP, except that it distributes address translation information to multiple memory modules and potentially allows parallel translations. However, this solution prohibits data caches. Since address translation is done outside processors, a data cache local to a processor must be a virtually tagged cache. As stated before, a virtually tagged cache stores PTE information (such as protection bits) in every cache line. The TLB inconsistency problem remains unsolved if the system has virtually tagged caches.

6.3. Performance Evaluation

The idea behind the iterative method that we use for performance analysis is not new. Since the theme of this work in not performance analysis methodology, we review only one representative book in this area. Agrawal surveyed and classified a number of approximation methods, cataloged a number of useful model transformations, characterized iterative solution procedures, and gave theorems about their convergence [Agr85]. He also identified the underlying modeling process and provided tools and techniques for model development.

Synchronizing TLBs is similar to updating replicated data in a distributed environment. Lee and Garcia-Molina both used an M/G/1 queueing model for this problem [Gar81, Lee80]. They assumed that the arrival of update requests is independent of the state of the system, and is a Poisson process. They then calculated the synchronization time and service time for different algorithms and applied them to the M/G/1 model. Our performance model, on the other hand, is a closed system. It is more realistic, especially when the system is saturated, but is more difficult to analyze.

There are a large number of simulation languages and packages, such as GPSS [BKP76, Sch74], SIMSCRIPT II.5 [MKV87], SLAM [Pri86], INSIGHT [Rob83], and SIMAN [Peg82]. We do not use any of them. Instead, we combine the recently developed Berkeley Interactive Statistical Systems (BLSS) with our own simulation engines written in C [AbR88]. The C engines take raw input traces and produce raw output traces; the BLSS generates input traces and analyzes output traces. This approach exploits the power of BLSS to simplify programming. It also takes advantage of the efficiency and flexibility of C to simulate complex algorithms that involve deadlock avoidance and processor interaction. Moreover, our simulators can be used for trace-driven simulation without writing extra programs.

Chapter 7
Concluding Remarks

This chapter concludes the dissertation. Section 7.1 lists its main contributions. Section 7.2 summarizes the major results. Section 7.3 discusses directions in which this research might be extended.

7.1. Contributions

This dissertation makes the following contributions to the research on software mechanisms for multiprocessor TLB consistency.

- It shows that TLB inconsistency is a fundamental problem in shared-memory multiprocessors. The problem exists when page table information is replicated, either in separate TLBs or in virtually tagged caches.
- It develops optimistic TLB synchronization algorithms that are more CPU-efficient than the 2-phase algorithm used by other systems.
- It analyzes and simulates the performance of TLB synchronization algorithms, showing that none of the algorithms evaluated scale well.
- It proposes a combined virtual memory system and message-passing system design that efficiently exploits virtual memory remapping by tolerating TLB inconsistency.

The next section develops these points, summarizing the major results of the dissertation.

7.2. Summary of Results

TLB inconsistency is a fundamental problem in shared-memory multiprocessors. It is different from the well-known cache inconsistency problem. TLB inconsistency occurs when *meta-data* (*i.e.*, the mapping of data) is changed, while cache inconsistency occurs when *data* is changed. Although virtually tagged caches eliminate the need for separate TLBs, they do not eliminate the TLB inconsistency problem. Virtually tagged caches store page table information, such as protection bits, in every cache line. Such information, when updated, causes a meta-data inconsistency problem that is equivalent to the TLB inconsistency problem. Consequently, TLB inconsistency should be defined, more precisely, as inconsistency among different copies of page table information.

Algorithms for synchronizing TLBs resemble, but are more complex than, those used by database systems to update replicated data. The main difference is in handling concurrency. It is difficult to implement software locking at the level of memory references; a processor does not check a software lock when making a memory reference, which implicitly uses a TLB entry. The 2-phase algorithm implements locking by stalling processors. The optimistic algorithms do not stall processors, but may require more than one round, and provide weaker consistency semantics than the 2-phase algorithms. Hardware TLB characteristics also affect TLB synchronization algorithms. The algorithms are simpler if (1) the status bits (dirty and referenced) of page table entries are set atomically, or (2) TLB entries are loaded by software via traps.

It is difficult to analyze TLB synchronization algorithms as queueing networks because some algorithms involve interaction, such as blocking, among processors. Instead, we obtain approximations using a computationally efficient iterative analysis method, the accuracy of which is verified by simulation results. The performance results show that the evaluated algorithms are CPU-bound in extreme cases, and that the optimistic-synch and optimistic-asynch algorithms are much more CPU-efficient than the 2-phase algorithm. More important, TLB synchronization algorithms perform well only under light workloads (such as paging in a system with large physical memory and a small number of processors); they do not scale well with the rate of TLB synchronization operations, with the number of processors, or with the overhead of flushing a TLB entry. This suggests the need for avoiding TLB synchronization under high workloads.

We propose an integrated system design that tolerates TLB inconsistency. We first identify three types of tolerable TLB inconsistency (safe, transient and trusted), and then develop mechanisms that tolerate them. In short, the design reduces the needs for TLB synchronization by exploiting trust relationships and lazy remapping. When TLB synchronization is necessary, the design does it efficiently by exploiting asynchrony. The design tolerates TLB inconsistency in paging and in message-passing. There is no good mechanism for tolerating TLB inconsistency in the copy-on-write mechanism. We explain why this is difficult, and show alternatives to copy-on-write. The design is part of the DASH experimental operating system. It has been implemented and measured on Sun 3/50 workstations.

The design, particularly for message-passing, spans various components of the operating system—from the low-level virtual memory system up to the semantic rules of inter-address space trust. This approach illustrates an important principle of system design: an integrated solution to a problem is often more efficient than a local solution. This principle has been applied to other systems areas. For example, the MIPS processor [GHP88, HJP82, HJB83] does not solve the pipeline interlocking problem purely by hardware but integrates it with the compiler. Our design is more scalable than those used in other systems because it does not confine the TLB inconsistency problem to hardware or even to the machine-dependent part of the virtual memory system. Finally, our design fits well into the software structure of operating systems, demonstrating that an integrated solution does not necessarily violate software layering.

7.3. Future Work

This work can be extended in two directions: implementation and performance evaluation. Relative to implementation, the design for tolerating TLB inconsistency (Chapter 5) has not been implemented in a shared-memory multiprocessor. Many multiprocessor-specific details, such as inter-processor interrupting, are either not exercised or not implemented at all. We need to gain implementation experience to find out whether the design has flaws or missing components. We also need to implement different TLB synchronization algorithms and measure their performance to verify our performance predictions. Further, we need to build application programs to examine how easy it is to program on top of our system design, and to gain experience with the workload of virtual memory remapping.

The performance evaluation part of the dissertation can be extended by verifying our assumptions and models, and by evaluating more performance metrics. This research assumes that all interarrival times and service times are exponentially distributed. This assumption makes the analysis tractable but is not necessarily realistic. Future research can examine the validity and accuracy of this assumption by exercising the simulators with different distributions and comparing the results.

Future research could also be directed at extracting parameters for our performance model from real systems, and comparing the analytic and simulation results with the measured results. Moreover, future research could drive the simulators with real traces, and compare the results with those derived from stochastic workloads. Such comparisons will show how closely our performance model represents real systems.

Although the iterative analysis method converges for all reasonable values of the parameters we have chosen, future research should formally study its convergence, *e.g.*, show under what conditions it converges.

Future research should also evaluate more performance metrics, such as the duration of interrupt-disabled time due to TLB synchronization. Often, the maximum time is more important than the average time because it determines whether critical external events might get lost. It is important to obtain not only mean values but also higher-order moments, if not distributions, of performance metrics.

Finally, future research could investigate the effect of TLB synchronization on cache and bus performance. As stated in Section 2.3, machines with virtually tagged caches have an equivalent TLB synchronization problem. When a page is remapped, all cache lines on all processors containing stale PTE information, such as protection bits, must be flushed. Therefore, in addition to the CPU overhead and latency we have taken into account, virtual memory remapping may affect future cache miss rates, and may even cause bus contention if all processors flush their caches concurrently [MBC86]. The effect of context switches on cache footprints and startup overhead has been studied [ThS87]. Since virtual memory remapping has a similar effect, it deserves similar attention.

Bibilography

[AbR88] D. M. Abrahams and F. Rizzardi, *BLSS: The Berkeley Interactive Statistical System*, W. W. Norton, Inc. , New York, 1988.

[ABB86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the 1986 Summer USENIX Conference*, Atlanta, Georgia, June 9-13, 1986, 81-92.

[ASH88] A. Agarwal, R. Simon, J. Hennessy and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherency", *Proceedings of The 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawali, May 1988, 280-289.

[Agr85] S. C. Agrawal, *Metamodeling: A Study of Approximations in Queueing Models*, The MIT Press, Cambridge, Massachusetts, 1985.

[AnF86] D. P. Anderson and D. Ferrari, "The DASH Project", *ACM SIGOPS Workshop on Distributed Systems*, Amsterdam, Sep. 1986.

[AFR87] D. P. Anderson, D. Ferrari, P. V. Rangan and S. Tzou, "The DASH Project: Issues in the Design of Very Large Distributed Systems", Technical Report No. UCB/CSD 87/338, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Jan. 1987.

[And88] D. P. Anderson, "A Software Architecture for Network Communication", *Proc. of the 8th International Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[ATG88] D. P. Anderson, S. Tzou and G. S. Graham, "The DASH Virtual Memory System", Technical Report No. UCB/CSD 88/461, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Nov. 1988.

[AnT88] D. P. Anderson and S. Tzou, "The DASH Local Kernel Structure", Technical Report No. UCB/CSD 88/463, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Nov. 1988.

[AnF88] D. P. Anderson and D. Ferrari, "The DASH Project: An Overview", Technical Report No. UCB/CSD 88/405, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Feb. 1988.

[ATW89] D. P. Anderson, S. Tzou, R. Wahbe, R. Govindan and M. Andrews, "Support for Continuous Media in the DASH System", Technical Report No. UCB/CSD 89/537, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Oct. 1989.

[BaJ81] O. Babaoglu and W. Joy, "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits", *Proc. of the 8th ACM Symp. on Operating System Prin.*, Pacific Grove, California, Dec. 14-16, 1981, 78-86.

[Bas85] E. Basart, "The Ridge Operating System: High Performance through Message-Passing and Virtual Memory", *Proc. of the IEEE 1st International Conf. on Computer Workstations*, San Jose, California, Nov. 11-14, 1985, 134-143.

[BKT87] B. Beck, B. Kasten and S. Thakkar, "VLSI Assist For A Multiprocessor", *Proc. Second International Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, Californic, Oct. 5-8, 1987, 10-20.

[BeG81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys 13*, 2 (June 1981), 185-221.

[BHG87] P. A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, Reading, MA, 1987.

[BRG89] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill and R. V. Baron, "Translation Lookaside Buffer Consistency: A Software Approach", *Proc. Third International Conf. on Architectural Support for Programming Languages and Operating Systems* , Boston, Massachusetts, Apr. 3-6, 1989, 113-122.

[BKP76] P. A. Bobillier, B. C. Kahan and A. R. Probst, *Simulation with GPSS and GPSS/V*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[BBM72] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10", *Comm. of the ACM 15*, 3 (Mar. 1972).

[BKN89] W. R. Byrne, T. A. Kilm, B. L. Nelson and M. D. Soneru, "Broadband ISDN Technology and Architecture", *IEEE Network*, Jan. 1989, 23-28.

[CHK88] L. F. Cabrera, E. Hunter, M. Karels and D. Mosher, "User-Process Communication Performance in Networks of Computers", *IEEE Trans. on Software Eng. 14*, 1 (Jan 1988), 38-53.

[ChM88] A. Chang and M. F. Mergen, "801 Storage: Architecture and Programming", *Trans. Computer Systems 6*, 1 (Feb. 1988).

[Che87] R. Cheng, "Virtual Address Cache in UNIX", *Proceedings of the 1987 Summer USENIX Conference*, Phoenix, Arizona, June 8-12, 1987, 217-224.

[ChZ83] D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations", *Proc. of the 9th ACM Symp. on Operating System Prin.*, Bretton Woods, New Hampshire, Oct. 10-13, 1983, 128-140.

[Che84] D. R. Cheriton, "The V Kernel: a Software Base for Distributed Systems", *IEEE Software 1*, 2 (Apr. 1984), 19-43.

[CSB86] D. R. Cheriton, G. A. Slavenburg and P. D. Boyle, "Software-Controlled Caches in the VMP Multiprocessor", *Proc. 13th Int.*

Symp. of Computer Architecture, June 1986, 366-374.

[CGB89] D. R. Cheriton, H. A. Goosen and P. D. Boyle, "Multi-level Shared Caching Techniques for Scalability in VMP-MC", *Proc. of The 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, May-June 1989, 16-24.

[ChH87] P. Chow and M. Horowitz, "Architecture Tradeoffs in the Design of MIPS-X", *Proc. of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, June 2-5, 1987.

[EGL85] J. Edler, A. Gottlieb and J. Lipkis, "Considerations for Massively Parallel UNIX systems on the NYU Ultracomputer and the IBM RP3", *Proceedings of the 1985 Winter USENIX Conference*, Dallas, Texas, January 23-25, 1985, 193-210.

[Emb87] D. R. Emberson, "A Cache Coherence Management Technique for Hypercube Multiprocessors", *Proc. of the 1987 International Conf. on Paralleling Processing*, Aug. 1987, 262-265.

[Ens74] P. H. Enslow, ed., *Multiprocessors and Parallel Processing*, John Wiley & Sons, New York, NY, 1974.

[FHM87] S. J. Farnbam, M. S. Harvey and K. D. Morse, "VMS Multiprocessing on the VAX 8800 System", *Digital Technical Journal*, Feb. 1987, 111-119.

[Fit86] R. P. Fitzgerald, *A Performance Evaluation of the Integration of Virtual Memory Management and Inter-Process Communication in Accent*, Ph.D. Dissertation, Carnegie-Mellon University, Oct. 1986.

[FiR86] R. Fitzgerald and R. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent", *Trans. Computer Systems 4*, 2 (May 1986), 147-177.

[GaP85] D. D. Gajski and J. Peir, "Essential Issues in Multiprocessor Systems", *IEEE Computer*, June, 1985, 9-28.

[Gar81] H. Garcia-Molina, *Performance of Update Algorithms for Replicated Data*, UMI Research Press, Ann Arbor, Michigan, 1981.

[GMS87] R. A. Gingell, J. P. Moran and W. A. Shannon, "Virtual Memory Architecture in SunOS", *Proceedings of the 1987 Summer USENIX Conference*, Phoenix, Arizona, June 8-12, 1987, 81-94.

[Goo87] J. R. Goodman, "Coherency for Multiprocessor Virtual Address Caches", *Proc. Second International Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, Californic, Oct. 5-8, 1987, 72-81.

[GoW88] J. R. Goodman and P. J. Woest, "The Wisconsin Multicube: A New Large-scale Cache-coherent Multiprocessors", *Proceedings of The 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawali, May 1988, 422-431.

[GVW89] J. R. Goodman, M. K. Vernon and P. J. Woest, "Efficient Synchronization Primitives for Large-scale Cache-coherent Multiprocessors", *Proc. Third International Conf. on Architectural*

Support for Programming Languages and Operating Systems , Boston, Massachusetts, Apr. 3-6, 1989, 64-73.

[Gov89] R. Govindan, "Read-only Sharing in Operating Systems", *U.C. Berkeley, Masters's Report*, Nov 1989.

[GHP88] T. Gross, J. Hennessy, S. Przybylski and C. Rowen, "Measurement and Evaluation of the MIPS Architecture and Processor", *Trans. Computer Systems 6*, 3 (Aug. 1988), 229-257 .

[HMS88] R. Haskin, Y. Malachi, W. Sawdon and G. Chan, "Recovery Management in QuickSilver", *Trans. Computer Systems 6*, 1 (Feb. 1988), 82-108.

[HJP82] J. Hennessy, N. Jouppi, S. Przbyski, C. Rowen, T. Gross, F. Baskett and J. Gill, "MIPS: A Microprocessor Architecture", *15 Annual Workshop on Microprogramming*, Oct. 1982, 17-22.

[HJB83] J. Hennessy, N. Jouppi, F. Baskett, T. Gross and J. Gill, "Hardware/Software Tradeoffs for Increased Performance", *Proc. first International Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 1983, 33-54.

[HeH86] J. L. Hennessy and M. A. Horowitz, "An Overview of the MIPS-X-MP Project", Technical Report STANCSL 86-300, Computer Systems Laboratory, Stanford Univ., Apr. 1986.

[Hil86] M. Hill, "Design Decisions in SPUR", *IEEE Computer*, Nov. 1986, 8-22.

[Hil87] M. D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, Ph.D. Dissertation, Technical Report No. UCB/CSD 87/381, Nov. 1987.

[III88] *N10 Programmer's Reference Manual, Preliminary Draft 2.4*, Intel, Apr. 1988.

[KaC88] H. Kanakia and D. R. Cheriton, "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors", *Proc. of ACM SIGCOMM 88*, Palo Alto, Calif., Aug. 1988, 175-187.

[Kle75] L. Kleinrock, *Queueing Systems, Volume I: Theory*, Johe Wiley & Sons, Inc., 1975.

[Kle76] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*, John Wiley & Sons, 1976.

[Lee60] F. F. Lee, "Study of 'Look Aside' Memory", *IEEE Trans. on Computers 18*, 11 (Nov. 1960), 1062-1064.

[Lee80] C. Lee, "Queueing Analysis of Global Locking Synchronization Schemes for Multicopy Databases", *IEEE Trans. on Computers c 29*, 4 (May 1980), 371-384.

[Lei88] B. Leiner, editor. "Critical Issues in High Bandwidth Networks", DARPA Internet RFC 1077, Nov. 1988.

[LeL82] H. M. Levy and P. H. Lipman, "Virtual Memory Management in the VAX/VMS Operating System", *IEEE Computer*, Mar. 1982, 35-41.

[Liu89] L. Liu, *Private communication*, IBM Yorktown Research Center, Sep. 1989.

[MKV87] H. M. Markowitz, P. J. Kiviat and R. Villanueva, *SIMSCRIPT II.5 Programming Language*, CACI, Los Angelos, CA, 1987.

[MBC86] M. A. Marsan, G. Balbo and G. Conte, *Performance Models of Multiprocessor Systems*, The MIT Press, Cambridge, Massachusetts, 1986.

[Mey85] E. L. Meyer, "Survey of multiprocessors", *VLSI Systems Design 6*, 11 (November 1985), 30-39.

[MMM86] *MIPS System Programmer Guide*, Mips Computer Systems, Inc., Mountain View, CA, 1986.

[MMM88a] *MC88100 User's Manual, Revision 0.6*, Motorola Microprocessor Group, Apr. 1988.

[MMM88b] *MC88200 User's Manual, Revision 0.4 Preliminary Copy*, Motorola Microprocessor Group, Apr. 1988.

[NeO88] M. Nelson and J. Ousterhout, "Copy-on-Write for Sprite", *Proceedings of the 1988 Summer USENIX Conference*, San Franscisco, CA, June 20-24, 1988, 187-202.

[OCD88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, "The Sprite Network Operating System", *IEEE Computer 21*, 2 (Feb. 1988), 23-36.

[Peg82] C. D. Pegden, *Introduction to SIMAN*, System Publishing Corporation, 1982.

[PfN85] G. A. Pfister and V. A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks", *Proc. of the 1985 Intl. Conf. on Parallel Processing*, 1985, 790-797.

[PBG85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proc. of the 1985 Intl. Conf. on Parallel Processing*, 1985, 764-771.

[Pri86] A. A. B. Pritsker, *Introduction to Simulation and SLAM II, 3rd Ed.*, System Publishing Corporation, 1986.

[PHH88] S. Przybylski, M. Horowitz and J. Hennessy, "Performance Tradeoffs in Cache Design", *Proceedings of The 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawali, May 1988, 290-298.

[R.S83] R.Suri, "Robustness of Queueing Networks", *J. ACM*, July 1983.

[RTY88] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky and J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *IEEE Trans. on Computers*, Aug. 1988, 896-908.

[Ras89] R. Rashid *et al.*, "MACH: A foundation for Open Systems", *The Second Workshop on Workstation Operating Systems*, Pacific Grive, CA, Sep. 27-29, 1989.

[Rit85] S. A. Ritchie, "TLB for Free: In-Cache Address Translation For a Multiprocessor Workstation", Technical Report No. UCB/CSD 85/233, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, May 1985.

[Rob83] S. D. Roberts, *Simulation Modeling and Analysis with INSIGHT*, Regenstrief Institute. Distributed by SysTech, Inc., Indianapolis, Indiana.

[Ros89] B. S. Rosenburg, "Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors", *Proc. of the 12th ACM Symp. on Operating System Prin.*, Litchfield Park, Arizona, 1989, 137-146.

[Ros86] F. E. Ross, "FDDI - A Tutorial", *IEEE Communications Magazine*, May 1986, 10-15.

[Sat80] M. Satyanarayanan, *Multiprocessors: A Comparative Study*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[Sch74] T. J. Schriber, *Simulation Using GPSS*, John Wiley & Sons, NY, 1974.

[SSS87] *Symmetry Technical Summary*, Sequent Computer Systems, Inc., 1987.

[Smi82] A. J. Smith, "Cache Memories", *Computing Surveys 14*, 3 (Sep. 1982), 473-530.

[Smi86] A. J. Smith, "Software Cache Consistency Control Using "One Time Identifier"", Technical Report No. UCB/CSD 86/290, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Apr. 1986.

[Sto85] M. Stonebraker, "Virtual Memory Transaction Management", *Operating Systems Review 19*, 2 (Apr. 1985), 8-16.

[SKP88] M. Stonebraker, R. Katz, D. Patterson and J. Ousterhout, "The Design of XPRS", *Proc. 14th Intl. Conf on Very Large Data Bases*, Aug. 1988, 318-330.

[Str86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[Sul90] M. Sullivan, "Software Fault Tolerance in Highly Available Database Systems", Technical Report, University of California/Electronics Research Lab, To appear, Jan. 1990.

[SSS85] *Sun-3 Architecture Manual, Version 2.0*, Sun Microsystems Inc., July 1985.

[TKS88] P. J. Teller, R. Kenner and M. Snir, "TLB Consistency on Highly-Parallel Shared-Memory Multiprocessors", *Proc. 21st Annual Hawaii Intl. Conf. on System Sciences*, 1988, 184-193.

[ThS87] D. Thiebaut and H. S. Stone, "Footprints in the Cache", *Trans. Computer Systems 5*, 4 (Nov. 1987), 305-329.

[TBJ88] M. Y. Thompson, J. M. Barton, T. A. Jermoluk and J. C. Wagner, "Translation Lookaside Buffer Synchronization in a Multiprocessor

System", *Proceedings of the 1988 Winder USENIX Conference*, Dallas, Texas, February 9-12, 1988, 297-302.

[TzS85] S. Tzou and Y. Shim, "A Study on the In-Cache Address Translation Mechanism of SPUR", CS252 Term Project, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Fall 1985.

[TAG87] S. Tzou, D. P. Anderson and G. S. Graham, "Efficient Local Data Movement in Shared-Memory Multiprocessor Systems", Technical Report No. UCB/CSD 87/385, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Dec. 1987.

[TzA88] S. Tzou and D. P. Anderson, "A Performance Evaluation of the DASH Message-Passing System", Technical Report No. UCB/CSD 88/452, Computer Science Div., EECS Dpt., Univ. of Calif. at Berkeley, Nov. 1988.

[WaM87] R. W. Watson and S. A. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices", *Trans. Computer Systems 5*, 2 (May 1987), 97-120.

[WeG89] W. Weber and A. Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors", *Proc. Third International Conf. on Architectural Support for Programming Languages and Operating Systems* , Boston, Massachusetts, Apr. 3-6, 1989, 243-256.

[Wil87] R. Wilson, "Designers Rescue Superminicomputers From I/O Bottleneck", *Computer Design*, Oct. 1987, 61-71.

[WEG86] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz and D. A. Patterson, "An In-Cache Address Translation Mechanism", *Proc. 13th Intl. Symp. of Computer Architecture*, June 1986, 358-365.

[WoK89] D. A. Wood and R. H. Katz, "Supporting Referenced and Dirty Bits in SPUR's Virtual Address Cache", *Proc. of The 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, May-June 1989, 122-130.