Translation-Lookaside Buffer Consistency in Highly-Parallel Shared-Memory Multiprocessors

Patricia Jane Teller

This research report is identical to
the author's doctoral dissertation
from New York University.

Translation-Lookaside Buffer Consistency
in Highly-Parallel Shared-Memory Multiprocessors

Patricia Jane Teller

Abstract

To implement virtual memory efficiently, virtual-to-physical address translation information is stored in *page tables* and cached in *translation-lookaside buffers (TLBs)*. In multiprocessors with multiple TLBs, page-table modifications can result in outdated TLB entries, the use of which can cause erroneous memory accesses.

We propose three new solutions to this *TLB consistency problem*, which unlike other solutions for highly-parallel shared-memory multiprocessors do not require interprocessor synchronization and communication, and neither interrupt processor execution nor introduce unnecessary serialization. The cost of these solutions is embodied in the cost of *TLB reloads*, which load into TLBs translation information for referenced pages. Two assume TLBs at processors and one assumes TLBs at memory.

We study their performance in scalable multiprocessor architectures with multi-stage interconnection networks via a trace-driven simulation system capable of simulating a range of architectures using just one address trace.

Our results show that system performance improves if TLBs are located at memory, rather than processors, provided that memory is organized as multiple *paging arenas*, where the mapping of pages to arenas is fixed.

A class of parallel workloads can produce a number of TLB reloads, R, that grows linearly with N. A set of our simulations for processor-based TLBs validate this model.

A processor-based TLB reload costs $O(\log N)$ because of network transit. Thus, the overhead of managing processor-based TLBs, be it consistency ensuring or not, grows as $R \log N$.

The cost of a memory-based TLB reload within a paging arena can be made smaller than that of a processor-based TLB reload, since additional network transits are not required. Simulation results show that memory-based TLBs with one paging arena exhibit generally larger miss rates than processor-based TLBs of equal size, and the related overhead is generally larger. Memory-based TLBs with two paging arenas produce smaller miss rates than processor-based TLBs of equal size, and the related overhead is generally smaller. For memory-based TLBs to maintain low overhead for large machines, it is likely that the number of paging arenas must grow as $O(N)$.

Dedication

This thesis is dedicated to my dear friend and loving companion, Joseph Teller, for his support, encouragement, and fortitude during many years of change. Joe's ability to make me smile under the dourest of circumstances has made all the difference.

This thesis is also dedicated to my mother, Adeline Cacciaguida Brady, my brother-in-law, Peter Van De Brook, and two special animal friends, Kim-Chu and Rouge, who passed away during my years as a graduate student.

Finally, this thesis is dedicated to the innocent animals of this world.

Acknowledgement of Financial Support

Part of the research reported in this thesis was conducted while at New York University where the author was supported by the National Science Foundation under grant number MCS-8301768, monitored by John Lehman, and the U.S. Department of Energy under grant number DE-FG02-88ER25052, administered by Donald Austin. The other part was conducted while at the IBM Corporation.

Acknowledgements (in somewhat chronological order)

Many thanks to Allan Gottlieb and Ralph Grishman for helpful discussions, constructive criticism, good advice, and moral support; Marc Snir for working with me at the beginning of this effort to formulate the three TLB consistency solutions presented in this thesis and for his encouragement to publish; Gregory Pfister for arranging my initial visit to IBM; Frederica Darema for her advice regarding porting parallel programs to IBM's VM/EPEX system; Kimming So for his help in modifying IBM's PSIMUL to generate augmented traces; Evelyn Melton and Jan Stone for working with me during the beta-testing of the VM/EPEX system; Richard Kenner for many helpful technical discussions and "arguments", and for his encouragement; Harold Stone for inviting me to visit IBM and for his guidance with respect to technical writing, analyzing and presenting data, and making presentations; Jan Edler and Bryan Rosenburg for helpful discussions regarding operating system issues; Kevin McAuliffe for sparking my interest in issues related to demand paging in multiprocessors; Marsha Berger for her technical appraisal of my thesis and for being on my doctoral commission; Elaine Weyuker for encouraging me to go to graduate school and for being on my doctoral commission; Malcolm Harrison for being on my doctoral commission and for his interest in my research; Susan Flynn Hummel for helpful technical discussions and for her encouragement; and, last but not least, Edith Schonberg for her encouragement, her perspective, and for "always being there".

Also, many thanks to my family for their love and my sister, Lorraine Van De Brook, for being my kindred spirit; Myron Norris for encouraging me to go to college; Mark Perlin for his never-failing, long-time friendship; the many graduate students, staff members, and faculty at NYU that added something special to my years as a graduate student; my colleagues at NYU's Ultracomputer Research Laboratory for making it a rich and enjoyable environment in which to work; and, finally, to my friends, old and new, who have helped me understand who I am and what is important in life.

Table of Contents

List of Illustrations

List of Tables

List of Graphs

Part I: The Problem

This part of the thesis includes:

- an introduction to and an overview of the thesis,
- definitions of terms and concepts used throughout the thesis,
- a description and illustration of the problem, and
- a discussion of the generality with which the problem is defined by the architecture and operating system.

Chapter 1. Introduction

This thesis proposes three new solutions to the translation-lookaside buffer (TLB) consistency problem targeted for highly-parallel shared-memory multiprocessors, and evaluates their performance in scalable multiprocessor architectures. Performance is measured via a trace-driven simulation study.

A *translation-lookaside buffer (TLB)* [Lee, 1969] is a special-purpose virtual-address cache that is used to implement virtual memory efficiently. TLBs are used to store and provide processors with fast access to recently-referenced virtual-to-physical address-translation information [Satyanarayanan and Bhandarkar, 1981]. This translation information is cached in TLBs and stored in data structures, called *page tables*.

In a multiprocessor architecture, such as the one pictured in Figure 1, it is generally the case that multiple TLBs are employed, conventionally one in each processor. In such an architecture, because the translation information for shared pages can be read and written by multiple concurrently-executing processes, a page-table modification can result in TLBs storing stale (*i.e.*, no longer valid) translation information, the use of which can lead to erroneous memory accesses.

The *TLB consistency problem* refers to how the use of stale translation information can be prevented. To efficiently implement virtual memory in a multiprocessor, TLBs must be managed efficiently. Thus, an efficient TLB consistency solution is required.

A question not answered by prior related research, but answered by our research, is: In a scalable multiprocessor architecture with a multi-stage network interconnecting processors and memory, how does the computational cost of a solution to the TLB consistency problem grow with N, the number of processors in the system? In such a multiprocessor the cost of a solution depends on N. For example, in a system with processor-based TLBs, the cost of a solution depends on the cost of network transit, which is $O(\log N)$.

After briefly discussing related research below in Section 1.1, the main contributions of this thesis are summarized in numbered paragraphs in Section 1.2. Section 1.3 ends this chapter with a description of the organizational structure of this thesis.

1.1. Related Research

Since network transit can severely degrade multiprocessor performance, good cache and TLB performance are important. Accordingly, the general problem of cache consistency (or cache coherence) in multiprocessor systems has been a very active research topic during the past few years (for a general survey of the area see Stenstroem [1990]; Cheong and Veidenbaum [1990]; Chaiken, *et al.* [1990]; Thakkar, *et al.* [1990]; and references therein). In comparison, the TLB consistency problem has received little attention. The next two sections briefly discuss the research done with respect to both the cache and TLB consistency problems.

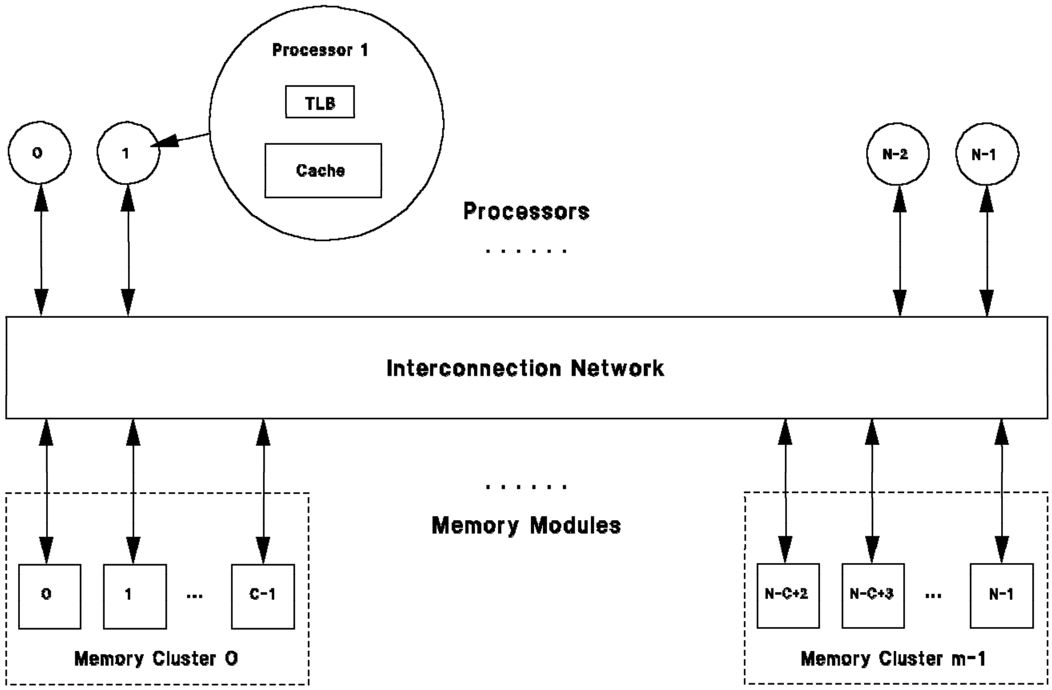


Figure 1. Multiprocessor Architecture

1.1.1. Cache Consistency

If shared data are stored in processor-specific caches, then in order to ensure cache consistency [Tang, 1976; Censier and Feautrier, 1978], the modification of shared data must trigger the invalidation or modification of its cache images. In addition, it must be ensured that a reference to the data generated by a processor without a valid cache copy is satisfied with an image that reflects the data's current state. Some existing solutions to the general cache consistency problem employ special-purpose hardware that can be implemented only in multiprocessors with efficient broadcasting capability. For example, some multiprocessors with a shared bus interconnecting processors and memory utilize hardware associated with each processor's cache that monitors bus transactions and maintains cache consistency essentially independent of the processor (*cf.*, [Archibald and Baer, 1986; Smith, October 1985; Yen, Yen, and Fu, 1985]). As mentioned by Vernon, Lazowska, and Zahorjan [1988], the simplest protocol employed by such solutions is a write-through protocol in which all writes to a cache are also written to main memory and cause other caches to update or invalidate their copies of the data [Smith, 1982]. Since Goodman's [1983] write-once protocol, which is a copy-back multicache consistency protocol, a number of more sophisticated protocols have been proposed, for example:

- the Synapse protocol [Frank, 1984],
- the Illinois protocol [Papamarcos and Patel, 1984],
- the RWB protocol [Rudolph and Segall, 1984],
- the Dragon protocol [McCreight, 1984], and
- the Berkeley protocol [Katz, *et al.*, 1985].

The efficiency of several of these solutions is examined by Archibald and Baer [1986], where the authors indicate to what extent each solution may limit the number of processors in the system.

Solutions of this type are not suitable for multiprocessors where the processors and memories are interconnected by more general networks, such as a multi-stage network, because the average latency associated with interprocessor or intercache communication increases with system size. Initially, some of the earlier scalable multiprocessor architectures of this type proposed not caching shared read/write data (*cf.*, [Gottlieb, 1987; Kuck, Davidson, Lawrie, and Sameh, 1986; Pfister, *et al.*, August 1985]). Since that time compiler-directed cache management has been explored for these types of systems (*cf.*, [Cytron, Karlovsky, and McAuliffe, 1988; Cheong and Veidenbaum, 1988]). In addition, the performance of distributed directory schemes in scalable multiprocessor architectures is currently being evaluated [Agarwal, *et al.*, 1988; James, *et al.*, 1990; Lenoski, *et al.*, 1990].

1.1.2. TLB Consistency

Although a TLB is, in fact, a cache, approaches that can be taken to solve the TLB consistency problem are not necessarily appropriate for solving the general cache consistency problem. This is primarily because the ratio of the number of modifications to the number of references for address-translation information is very small in comparison to the same ratio for other kinds of data.

TLB consistency solutions proposed by other researchers fall into two classes:

1. Solutions in the first class assume that the targeted multiprocessor has efficient broadcast capability. Accordingly, at the present time, these solutions are limited to multiprocessors that interconnect processors and memory by a shared bus. Solutions in this class are those of Cheriton, *et al.* [1986], Wood, *et al.* [1986], and Goodman [1987]. All these solutions assume virtual-address general-purpose caches that store would-be TLB entries along with other data and are kept consistent via a particular protocol.
2. Solutions in the second class require explicit interprocessor synchronization and communication, forcibly interrupt processor execution, and may introduce serialization. Solutions in this class are those of Thompson, *et al.* [1988], Black, *et al.* [1989], and Rosenburg [1989]. Black, *et al.* introduced the TLB Shootdown algorithm. Rosenburg improved upon this algorithm and Thompson, *et al.* employ a TLB Shootdown-like algorithm to implement their solution.

Only solutions in the second class can be implemented in multiprocessor systems with more general interconnection networks. Therefore, these are alternate solutions to ours.

Black, *et al.* [1989] and Rosenburg [1989] have studied the performance of their TLB consistency solutions in real multiprocessor systems, where the average cost of network transit remains relatively constant. Black, *et al.* study the performance of TLB Shootdown in a 16-processor NS32332 Encore Multimax that interconnects processors and memory via a shared bus. Rosenburg compares the performance of his modified version of TLB Shootdown with that of the original algorithm in IBM's 64-processor RP3 that interconnects processors and memory via a multi-stage network.

To our knowledge, trace-driven simulations have not been used previously to study the performance of TLBs in multiprocessor systems. Trace-driven simulations that have been used to study the performance of general-purpose caches in multiprocessors have conventionally used either synthetic reference streams [Archibald and Baer, 1986] or a set of address traces, one trace for each of P processors, generated by real multiprocessor systems [Eggers and Katz, 1987].

1.2. Main Contributions

The main contributions of this thesis are summarized below:

1. ***Three New TLB Consistency Solutions***: We propose three new solutions to the TLB consistency problem [Teller, Kenner, and Snir, 1988]:

- Read-locked TLBs,
- Validation, and
- Memory-based TLBs.

The first two solutions assume that TLBs are located at processors, while the third solution assumes that TLBs are located at memory. These solutions are targeted for multiprocessors with general interconnection networks, in particular scalable multiprocessors with multi-stage networks, such as the NYU Ultracomputer [Gottlieb, 1987], CEDAR [Gajski, Kuck, Lawrie, and Sameh, 1983], the IBM RP3 [Pfister, *et al.*, August 1985], the BBN Butterfly [Rettberg and Thomas, 1986], and the Tera Computer [Smith, 1989].

Unlike previous solutions that can be employed in such systems, our solutions allow processors to act independently of other processors while performing actions that ensure TLB consistency. In addition, processor execution is not forcibly interrupted and serialization is not introduced. The performance cost of these solutions is reflected primarily in the cost of TLB reloads. (A *TLB reload*, which is triggered by a TLB miss, loads into a TLB the translation information for a referenced page.)

2. ***Simulation Method***: Our simulation study of the performance of TLB consistency solutions in scalable multiprocessor architectures includes the effects of varying the:

- number of processors (N),

- network transit time,
- TLB size,
- page size,
- workload, and
- frequency with which the TLB consistency problem would arise if a TLB consistency solution was not in place.

In contrast, prior related performance studies were conducted in real systems with either 16 or 64 processors, where the average cost of network transit remains relatively constant and the size of TLBs, pages, and frames are fixed.

Our simulation methodology allows us to simulate a range of multiprocessor systems of different sizes and architectures with just one uniprocessor address trace of a parallel program. This contrasts with the conventional practices of either generating a synthetic reference stream per processor or employing a set of multiprocessor address traces for each system size.

In addition, our methodology employs trace compression techniques that significantly reduce simulation time and the disk space required to store the traces of a class of parallel programs. For the two programs used in this study, the size of the trace of one iteration of each of the initialization, computational, and termination sections of the program was reduced by a factor of 8 and 26, respectively. Since these programs are iterative problem solvers, we are able to further reduce the size of the traces by only tracing one iteration of each different flow of control of the computational section of the program. Thus, if i iterations of the computational section of the program are simulated, in effect the size of the trace is reduced by an additional factor of i.

3. ***Model of TLB-reload Overhead***: Our model of TLB-reload overhead estimates the overhead as $R \times C_{reload}$, where C_{reload} is the contention-free cost of a TLB reload and R is the number of reloads. The results of our simulations of systems with processor-based TLBs validate this model, but the results of our simulations of systems with memory-based TLBs do not. Since the cost of our solutions is embodied primarily in the cost of TLB reloads, their computational overhead is measured by $R \times T_{reload}$, where T_{reload} is the average cost of a reload.

4. ***Model of the Growth of R with N***: Our model of the growth of R with N indicates that certain parallel programming constructs, contained in the programs used in our study, can cause R to grow linearly with N. Our simulations of systems with processor-based TLBs that were driven by the trace of one of the programs used in our study validate this model by showing that R grows linearly with N. Our other simulations do not indicate that R grows linearly with N. Evidence of a linear growth of R with N may depend on the the number of processors, TLB size, page size, the scheduling of tasks to processors, and, in the case of memory-based TLBs, the number of paging arenas. (A *paging arena* is a cluster of memory modules to which the mapping of a page is fixed.)

5. ***Simulation Results***: Our simulation results indicate that:

- The overhead of our solutions is primarily determined by R, the number of TLB reloads.
- The reload overhead of processor-based TLBs grows as $O(R \log N)$.
- For memory-based TLBs, R and, thus, the TLB-reload overhead decrease with the number of paging arenas.
- For larger N, processor-based TLBs outperform memory-based TLBs with one paging arena, but memory-based TLBs with multiple paging arenas outperform processor-based TLBs.
- For the workloads that illustrate R growing linearly with N, we give evidence that for large N, TLB Shootdown would have a significantly higher computational overhead than our solutions.

1.3. Thesis Organization

This thesis is organized into four parts:

1. Part I, entitled *The Problem*, includes this chapter and Chapter 2, which discusses the milieu in which the TLB consistency problem arises, and the problem itself.
2. Part II, entitled *Solutions*, contains Chapter 3, 4, and 5, which present solutions to the TLB consistency problem. Chapter 3 begins with a discussion of the strategies that have been employed to solve the TLB consistency problem and the factors that contribute to whether a solution is suitable for a given multiprocessor system. The remainder of the chapter is primarily devoted to solutions of other researchers that assume efficient broadcast capability. Chapter 4 begins with a discussion of issues related to the performance of a TLB consistency solution in large-scale multiprocessor systems. Then, it presents solutions of other researchers that can be employed in systems with more general interconnection networks. Our solutions to the TLB consistency problem are presented in Chapter 5.
3. Part III, entitled *Performance Evaluation via Simulations*, contains Chapters 6, 7, and 8, which are devoted to our simulation study. Chapter 6 describes our simulation methodology. The input parameters and traces used in our study are discussed in Chapter 7. And, the results of these simulations are presented in Chapter 8.
4. Part IV, entitled *Conclusions and Future Research*, is comprised of only Chapter 9, which outlines our conclusions and discusses possible directions for future research that are suggested by this thesis.

Chapter 2. Background

In order to introduce the reader to the TLB consistency problem, this chapter describes the milieu in which this data consistency problem arises and discusses the problem itself. The TLB consistency problem arises as a result of using special-purpose virtual-address caches, which are called translation-lookaside buffers (TLBs), to implement paged virtual memory efficiently. Therefore, we first present, in Sections 2.1, 2.2, and 2.3, tutorial material that describes the functions of a memory hierarchy, a paged virtual memory, and a TLB. Although these sections may be skipped by some readers, they serve to define many of the terms that are used in this thesis and point out how data inconsistencies arise in general.

In both uniprocessor and multiprocessor architectures, data consistency problems can arise among different levels of the memory hierarchy, but in multiprocessor architectures data consistency problems can arise among multiple components on the same level of the hierarchy as well. For example, in a uniprocessor the image of data cached in a TLB must be kept consistent with the image stored in a data structure, called a page table, which is at a different level of the memory hierarchy. In a multiprocessor with multiple TLBs, images of the same data can be cached in more than one TLB. Therefore, in addition to a data modification causing inconsistency between a TLB entry and a page-table entry, it can cause inconsistencies among these multiple TLB images as well.

The TLB consistency problem is the subject of Section 2.4, where we illustrate how TLB inconsistencies arise in the context of both uniprocessor and multiprocessor architectures. The problem arises in a multiprocessor because processes that can be executed in parallel have access to images of the same data that are stored in TLBs and page tables. As a result, a process that modifies a page-table entry may cause TLB entries to become stale. Since data stored in a TLB indicate a page's location in main memory, the use of a stale TLB entry may cause erroneous memory accesses. Hence, it must be ensured that stale TLB entries are not used to access memory.

In Section 2.4 page-table modifications are classified as either safe or unsafe changes. Safe changes can be detected by the operating system during program execution but unsafe changes cannot. Thus, we consider unsafe changes to be the only source of potential TLB inconsistencies. Unsafe changes are shown to be instances of the classical problem of obtaining a consistent view of a shared, complex data structure. In this case the data structure is comprised of a page's virtual-to-physical address translation information, which is stored in a page table and may be cached in one or more TLBs. Accordingly, we show that readers-writers semantics are necessary, either implicitly or explicitly, in order to solve the TLB consistency problem.

The chapter ends with Section 2.5, which discusses the generality with which the TLB consistency problem is posed. We show that the architecture and operating system, in defining how memory is allowed to be shared among processes, define the context in which the TLB consistency problem is to be solved and, thus, the way in which the problem is cast. The more general the definition of shared memory, the more general the TLB consistency problem. The most general TLB consistency problem is defined when memory sharing is unrestricted and unsafe changes can be made to data that is cached in multiple TLBs and is being used by concurrently executing processes.

2.1. Hierarchical Memory

Many computer systems employ a hierarchical memory in order to provide a large-capacity memory system that is cost effective, without compromising performance. The goal of the memory hierarchy is to match the relatively high speed of the processor with the relatively low speed of main memory. A memory hierarchy, such as the one depicted in Figure 2, provides a computer system with various levels of storage. By exploiting the principle of locality the average delay for memory access in such a system is little more than that of the fastest level of the hierarchy, but the cost per bit is close to that of the lowest level of the hierarchy.

In the memory hierarchy pictured in Figure 2 the registers are at the highest level of the hierarchy and the auxiliary memory is at the lowest level. The time required to access data stored in a memory device on a particular level of the hierarchy increases with the level's distance from the processor. In general, as the access speed of a device decreases, its capacity increases and its cost decreases. Thus, a memory device on the highest level of the hierarchy, *i.e.*, closest to the processor, provides the fastest access to data. The caveat is that the amount of storage available at the faster levels is most likely smaller and more expensive than that at the slower levels. Just as the storage capacity grows from one level of the hierarchy to the next lower level, so does the unit of transfer. For example, in Figure 2 the size of the data that moves from auxiliary memory to main memory is 256 times as large as the size of the data that moves from main memory to the cache; and the size of the data that moves from main memory to the cache is 4 times as large as the size of the data that moves from the cache to a register. Before the execution of a program begins, the lowest level in the memory hierarchy stores the data and instructions associated with the program.

The goal of a memory hierarchy can only be met if the movement of data in the hierarchy is managed so that as the time interval before data are referenced decreases, a copy of the data migrates from lower to higher levels of the hierarchy. Since the storage capacity diminishes as data travels upward in the hierarchy, as data are referenced other data that were previously referenced are replaced. A replacement policy is employed at each level of the hierarchy that attempts to place data in the hierarchy according to when it will be referenced in the future. The more imminent a datum's reference, the closer its desired proximity to the processor.

This memory management strategy is effective because it has been shown that programs generally exhibit *locality of reference* [Denning, 1972]. In other words, during a time interval programs tend to reference memory locations in close proximity of one another, *e.g.*, neighboring instructions of a program or elements of a single array. In addition, programs tend to reference repeatedly a set of memory locations, *e.g.*, a computational loop of

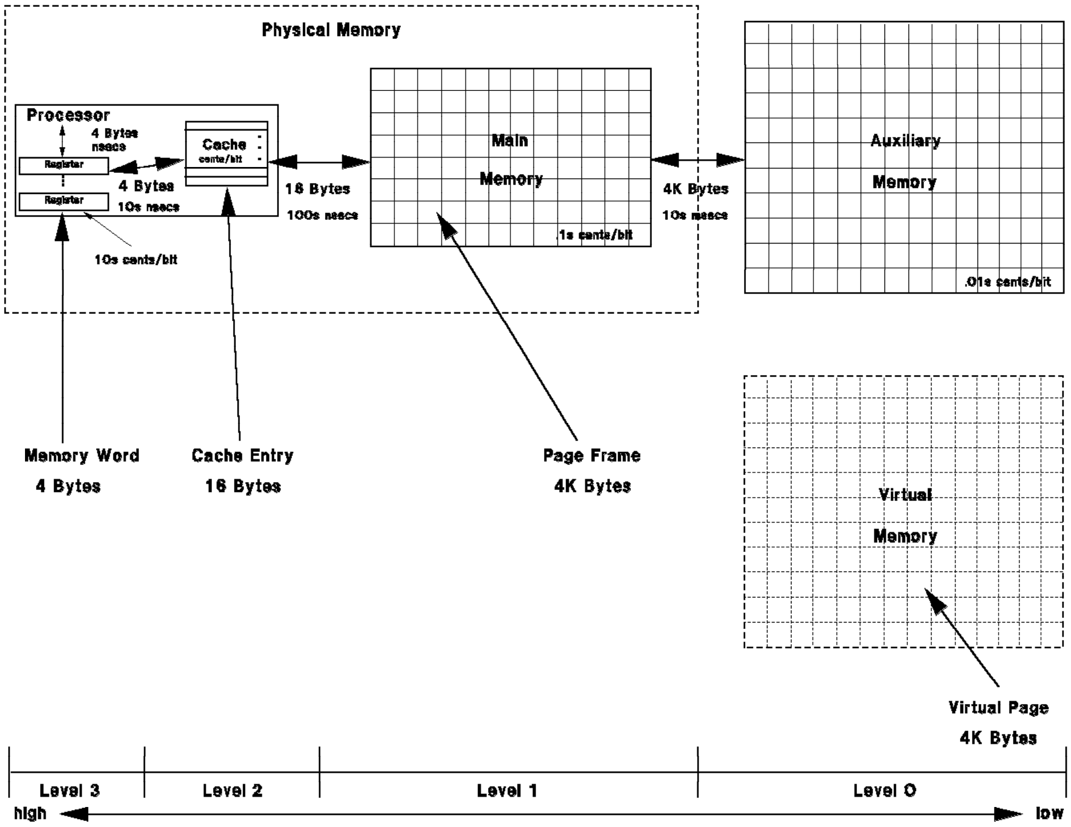


Figure 2. A Memory Hierarchy in a Uniprocessor

a program. These two reference patterns are called *spatial* and *temporal* locality, respectively.

A hierarchical memory organization may be attractive for the shared memory of multiprocessors because most parallel programs exhibit locality of reference to shared code and many programs exhibit significant locality of reference to shared data. Some programs, however, may require reorganization in order to improve the locality of reference to shared data among processes.

2.2. Paged Virtual Memory

A hierarchical memory system that supports paged virtual memory includes both main and auxiliary memories. *Virtual memory* [Denning, 1970] is an abstraction that permits the memory of a computer system to be managed in such a way that a program can be executed when only a portion of its instructions and data are memory resident. As a result, an application program that requires a memory space much larger than the size of main memory can be executed. In addition, a virtual memory system allows the computer to share its computational resources among several programs or processes, the instructions and data of which are only partially resident in main memory. Other advantages that can be realized by a computer system with virtual memory include multiprogramming without requiring contiguous memory partitions for each program, better main memory utilization, and easier implementation of protection facilities.

Many application programs, which are intended to run on multiprocessors as well as uniprocessors, require a virtual address space much larger than physical main memory. To support these applications, an efficient implementation of a paged virtual memory system is required. As explained below, paging is a way of implementing virtual memory where fixed size units of virtual memory images resident in auxiliary memory migrate to and from physical main memory. Paged virtual memory is already supported in large supercomputers such as the CDC Cyber 205 [Control Data Corporation, 1980, 1981], the Fujitsu VP-200 and the Hitachi S810/20 [Lubeck, Moore, and Mendez, 1985], in commercially available, bus-connected multiprocessor systems such as the Elxsi System 6400 [Sanguinetti, 1986] and Sequent's Balance 8000 and 21000 [Sequent Computer Systems, Inc., 1986], and in BBN's Butterfly family of multiprocessors (*cf.*, [BBN, 1985, 1988, 1990]). In addition, experimental results of Li [1988] suggest that virtual memory may be employed effectively in in loosely-coupled multiprocessors.

In a paged virtual memory system, physical main memory consists of a set of page frames, or simply frames, and virtual memory consists of a set of virtual pages, or simply pages. Each *frame* contains an equal number of memory locations and, since pages are mapped to frames, each *page* consists of a number of contiguous virtual memory locations equal to a multiple of the number of memory locations in a frame.[1] An image of a virtual page can reside in both main and auxiliary memories, but only the image in main memory can be read or written. Therefore, these two images are often inconsistent.

A program is comprised of one or more computational sections, which we call *tasks*, and each task is associated with one or more computational entities, which we call *processes*. The relationship between tasks and processes depends on how virtual memory is implemented.

A process' *address space* is the set of virtual addresses that can be generated by a processor while executing the process. In a paged virtual memory system a process' address space consists of the set of virtual pages that the process is permitted to access. While executing a process, a processor generates virtual addresses in order to access data referenced by the process. The high-order bits of a virtual address identify the page in which the data is stored, and the low-order bits indicate the displacement from the beginning of the page to the location of the data. For example, if the page size is 4 Kbytes then the lower 12 bits of the address indicate the location of the referenced word within the page, and the high-order bits identify the virtual page number. A processor can only access a datum when it is stored in physical memory, *i.e.*, main memory and levels of the memory hierarchy closer

[1] Some systems have pages of different sizes. The size of a page is equal to either that of a frame or a multiple of the frame size.

to the processor. In some systems, before accessing data the virtual address of the page first must be mapped to the physical address of its frame, where the high-order bits identify the frame and the low-order bits, a displacement. Such a mapping only exists when the page is resident in main memory. Address translation is not necessary if a virtual-address general-purpose cache is associated with each processor and the referenced data are cacheable and cache resident.

If a page is referenced and is not resident in main memory, a frame or frames must be allocated to store the page. If all of main memory is allocated, the referencing of non-resident data results in the eviction of another page. Before a page is evicted, however, it must be ensured that an exact image of the page resides in auxiliary memory; this may require that the page being evicted be written to auxiliary memory before the newly referenced page is stored in the frame that is being vacated. Then, if the newly referenced page has an image in auxiliary memory, it must be read from the device on which it is stored to the allocated frame.

In order to manage virtual memory, information must be stored that defines the status and location of each page. This information is stored in a data structure, which is called a *page table*, that has an entry for each virtual page. A page-table entry contains a page's virtual-to-physical address translation information, its location in auxiliary memory, and information used by the operating system, *e.g.*, to implement the replacement policy for the main memory level of the memory hierarchy. A page's translation information includes:

- the location of the page in physical memory (a frame number) if it is resident;
- a protection field, which defines the ways in which the page can be accessed, *e.g.*, read/write or read-only;
- a valid bit, sometimes called a resident bit, which indicates whether or not the page is resident; and
- a modify bit, sometimes called a dirty bit, which indicates whether or not the page has been modified since it became main-memory resident, *i.e.*, if an exact image of the page does not reside in auxiliary memory.

There are various ways in which virtual memory can be implemented. The address space of a process can be defined so that two processes have unique virtual memory spaces, however, each memory space is addressed by the same set of virtual addresses. As depicted in Figure 3, in this case a page table is associated with each process and a page accessed by multiple processes may or may not be addressed differently by each process.

Another way of implementing virtual memory, as illustrated in Figure 4, is to define one large global address space. In this case, processes sharing a page may use the same virtual address to reference it. The address spaces unique to processes are distinguishable from one another and conceptually one page table is shared by all processes. If different virtual addresses are not used to reference a shared page then the *address synonym problem* [Smith, 1982] is avoided.

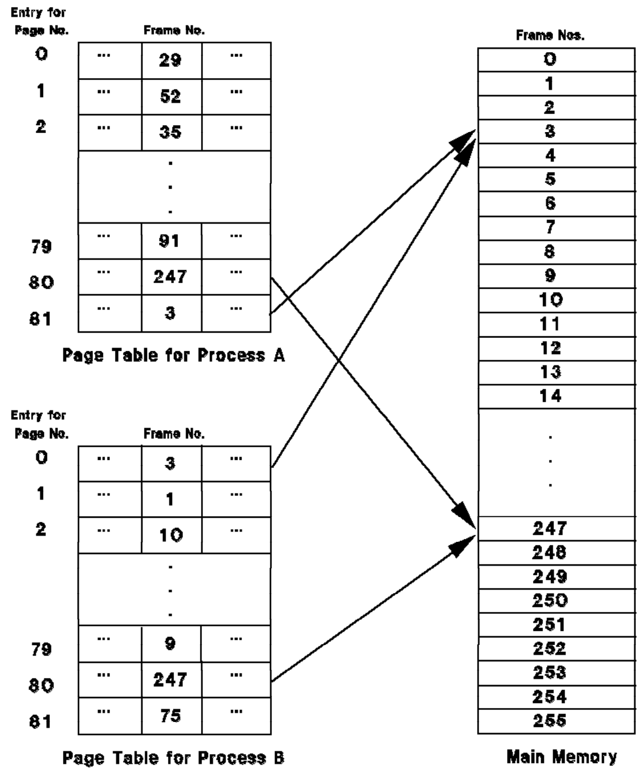


Figure 3. Nonunique Virtual Memory Addressing and Page Sharing

Virtual memory can also be implemented in such a way that a memory space is associated with each task, rather than with each process. Then, as shown in Figure 5, the address spaces of two tasks are not unique and processes associated with a task share the task's address space and its page table. Processes associated with the same task reference the same page with the same virtual address. Processes in different tasks may or may not reference the same page with the same virtual address.

As mentioned above, while executing a process, a processor generates virtual memory addresses. When virtual memory is implemented so that process address spaces are indistinguishable, a process identifier can be affixed to each address in order to make it recognizable as belonging to a particular process. Whether or not this is done, in order to access referenced data these addresses must be translated into physical memory addresses by using translation information stored in page-table entries. Programs generally exhibit locality of reference: during a short time interval a process usually makes multiple references to the same page. Thus, a cache that stores translation information allows the processor to avoid accessing main memory to obtain a virtual-to-physical mapping on each memory reference.

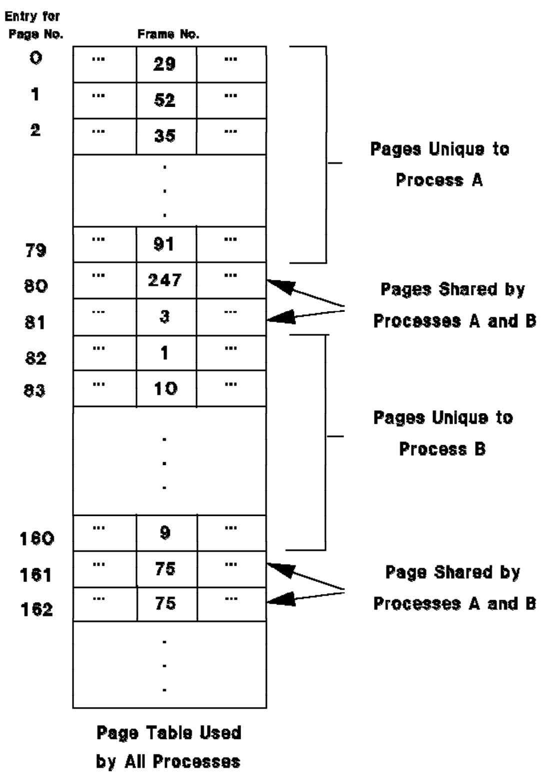


Figure 4. Unique Virtual Memory Addressing and Page Sharing

The function of this special-purpose cache, which is called a translation-lookaside buffer, is described in the next section.

2.3. Translation-Lookaside Buffer

A translation-lookaside buffer (TLB), also known as a translation buffer (TB) or directory-lookaside table (DLAT), is a special-purpose virtual-address cache that stores recently used address translation information. A TLB entry, a sample of which is depicted in Figure 6, caches an image of the translation information stored in a page-table entry.

As described above, rather than access a page table whenever address translation is required, TLBs are used to provide processors with fast access to the translation information for recently referenced pages. Since programs generally exhibit locality of reference, a process usually makes multiple references to the same page during a certain time interval. Thus, a TLB often eliminates the need for a processor to access translation information from

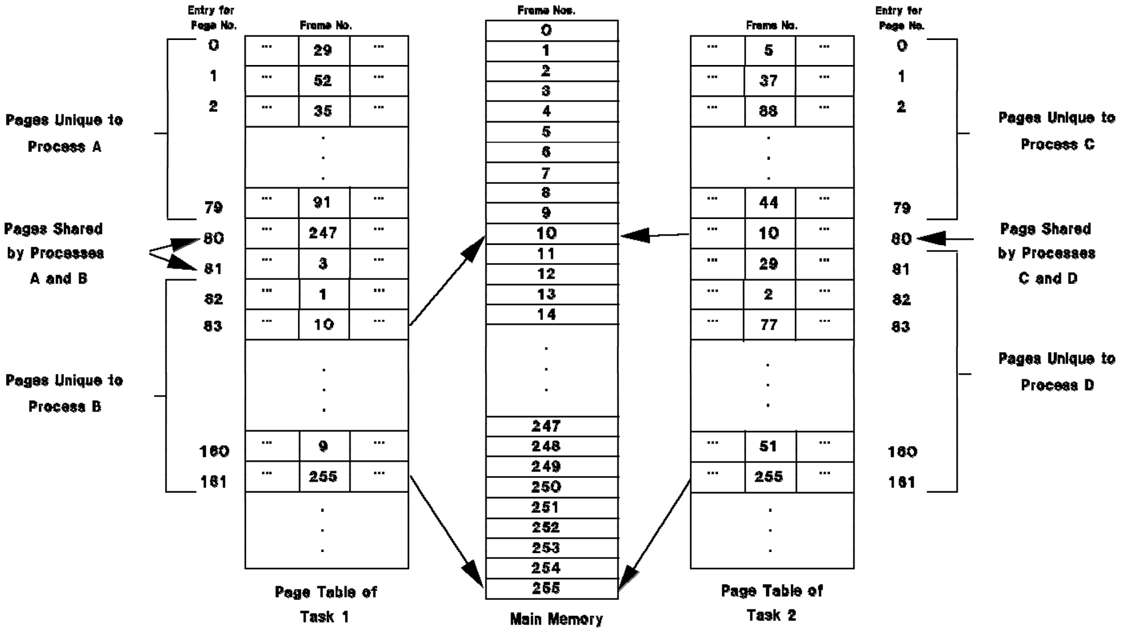


Figure 5. Process Sharing of Task Address Space and Page Sharing

a page table in main memory. Measurements of the VAX-11/780 reveal that translation information can be accessed from a TLB rather than from a page table more than 97 times out of 100, *i.e.*, the TLB miss rate is under 3% [Clark and Emer, 1985]. Without a TLB, one or two additional memory accesses may be required to satisfy a data request. When virtual memory is organized in segments, where each segment contains a set of pages, a data structure, which is called a segment table, may have to be accessed in order to obtain the location of the page table containing the translation information for the referenced page. Similarly, in the VAX-11/780 [Digital Equipment Corp., 1977], the mapping for process space addresses involves two memory references: One is for translating the process space address into a physical memory address, and the other is for translating the system virtual address of the table containing the first translation.

In a multiprocessor where the processors and memory are interconnected by a multi-stage network, the importance of caching translation information grows with the size of the system because the cost of accessing memory grows with the dimensions of the network. Moreover, if the translation information for a shared page is not replicated, it may become

Page Number	Frame Number	Protection Information	Valid Bit	Modify Bit	Algorithm Specific

Figure 6. A TLB Entry

a "hot spot" of memory activity [Pfister and Norton, 1985] unless a combining network [Dickey, *et al.*, 1985] is used.

The data paths of a typical physical-address cache and TLB design, where the page and frame sizes are assumed to be equal, are illustrated in Figure 7. In this case, cache access and address translation can be overlapped to some extent [Smith, 1982]. A processor uses the high-order bits of the virtual address (the page number) to access the translation information for the referenced page from the TLB. A *TLB hit* occurs if a valid TLB entry exists for the page. On a TLB hit the number of the frame in which the page is stored is concatenated with the low-order bits of the virtual address. This results in the physical address of the referenced data, which can be used to access both the cache and main memory. Otherwise, a *TLB miss* occurs which initiates the execution of a TLB-reload routine. A *TLB reload* accesses the referenced page's translation information from the page table (using the virtual page number to form the address of the page-table entry) and loads it in the TLB. In general, one entry must be removed from the TLB in order to make room for another.

If the referenced page is not resident in main memory, a *page fault* occurs and, if the page is not already in the process of becoming resident, page-in processing is initiated. A *page-in* allocates a frame to the page and, if necessary, copies the image of the page in

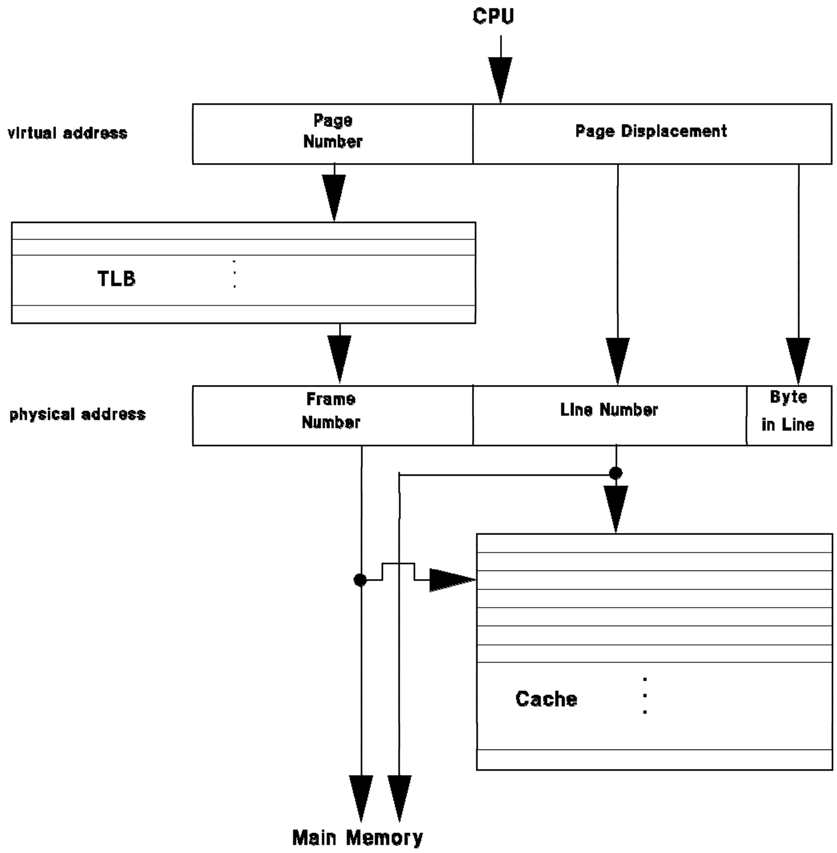


Figure 7. Data Paths of a Typical Physical-Address Cache and TLB Design

auxiliary memory to the allocated frame. If all of physical memory is allocated, a page-in results in the eviction of another page, *i.e.*, the removal of a page from main memory. As mentioned in Section 2.2, before a page is evicted it must be ensured that an exact image of the page resides in auxiliary memory; this may require that the page being evicted be written to auxiliary memory before it is overwritten by the newly referenced page.

Processes may share a page, *i.e.*, the page may be accessed by multiple processes. As previously described, this sharing can be accomplished by allowing either a one-to-one or a many-to-one mapping of pages to frames. If pages with indistinguishable virtual page numbers can map to two different frames, then all the entries in a TLB must be invalidated on a *context switch*, the switching of processor execution from one process to another. This invalidation process is called a *TLB flush*. If pages are mapped to frames on a one-to-one basis or process identifiers are affixed to virtual page numbers, a frame is always referenced using a unique page number. Then a context switch does not require a TLB flush in a uniprocessor (or a multiprocessor that forbids process migration). This is also the case in a multiprocessor in which processes are allowed to migrate among processors if it is guaranteed that the use of stale TLB entries to access memory is prevented.

As described in the next section, the use of TLBs gives rise to the data consistency problem known as the TLB consistency problem.

2.4. TLB Inconsistency

As described above, two images of the translation information for a page can exist: one stored in a page table and the other stored in a TLB. Both of these images may be accessible to a process; the TLB image is used to perform virtual-to-physical address translation and the page-table image is used to perform TLB reloads or to modify translation information. Since a page's translation information is modified whenever its status changes, the translation information for a shared page is always writable, even if the mapped page is read-only. Therefore, page-table modifications can cause TLB entries to become inconsistent with the page-table entries they are supposed to mirror. Since the use of inconsistent TLB entries may generate erroneous memory accesses, this TLB consistency problem [Teller, June 1990] needs to be solved by either ensuring consistency among TLBs and page tables or preventing the use of inconsistent TLB entries to access memory.

A page-table entry is modified to reflect a change in a page's status with respect to its:

- residency in physical memory,
- virtual-to-physical mapping,
- access permissions, or
- reference history.

TLB inconsistencies resulting from updates of page-reference history information, *e.g.*, bits that record whether or not a page has been modified or referenced during some time interval, are harmless if certain precautions are taken when updating page tables. We separate the other page-table modifications into safe and unsafe changes, which we describe below. Safe changes can be detected easily during program execution, while unsafe changes cannot.

It is not important that page-reference history information be consistent among TLBs and page tables. However, it is crucial that the reference history information stored in page tables reflect states that will not cause erroneous program behavior. A modify bit, which indicates whether or not a page has been written while resident in main memory, must be accurate. If it is not, important information can be lost. For example, suppose that a page is dirty, *i.e.*, it has been written while memory resident, but its modify bit is not set. Then it is possible for the only accurate representation of the page, stored in main memory, to be overwritten without a copy of it being resident in auxiliary memory. On the other hand, a bit that indicates whether or not a page has been referenced need not be accurate since it is generally used as a heuristic to select a page to replace in main memory. Replacing one page instead of another does not cause errors, although it may have performance consequences.

If processors in a multiprocessor system can concurrently modify a page-table entry, *e.g.*, one can be in the process of changing a page's mapping, while another is in the process of setting the page's modify bit, the updating of a page's reference history must be handled in such a way that it does not corrupt current page-table information. For example, architectures that update page-usage information by rewriting a page-table entry must face the problem that such a rewrite can obliterate recently modified translation information. In addition, architectures that use selective atomic operations to effect such modifications, such as a fetch-and-or operation (described later), must ensure that the modification of translation information does not eradicate recently updated page-usage information.

A *safe change* results from:

- a reduction in page protection, *e.g.*, the modification of access rights from read-only to read-write, or
- a page becoming main-memory resident.

The use of a TLB entry that is inconsistent as a result of a safe change can be permitted by designing the hardware so that when the entry is used an exception is generated, which invokes an operating system trap routine. The trap causes the TLB entry to be invalidated and/or corrected. For example, suppose that a valid entry of the TLB accessed by processor X stores the translation information for page a and defines the page's protection to be read-only. Further suppose that page *a*'s page-table entry is subsequently modified by processor *Y* to reflect a change in protection from read-only to read/write. Since the TLB entry for page *a* accessed by processor *X* defines page *a*'s protection to be read-only, the operating system intervenes when processor *X* attempts to write page *a*. Upon checking page *a*'s page-table entry, the operating system finds that page *a*'s protection has been decreased and, thus, invalidates the stale TLB entry. Assuming that processor *A* does not switch to the execution of another process, when it resumes execution, after reloading a consistent TLB entry for page *a*, the modification of page *a* is successfully and correctly executed.

In contrast, *unsafe changes* cause TLB inconsistencies that cannot be detected during program execution. It is this class of page-table modifications that brings about the need for consistency-ensuring TLB management. An unsafe change results from:

- the invalidation of a virtual-to-physical memory mapping,
- an increase in page protection, or
- the remapping of a virtual or physical page, *i.e.*, a change in the mapping of a page to a frame.

The virtual-to-physical mapping of a page is invalidated when either virtual memory is deallocated or a page is evicted from main memory. Virtual memory is said to be deallocated when virtual pages are removed from the address space of a process. The copy-on-write optimization and interprocess communication can be implemented by increasing the protection of a page that is shared among processes. As explained in a bit more detail in Section 2.5, *copy-on-write* saves copying overhead by allowing a writable page to be shared read-only by multiple processes, until one of the sharing processes actually attempts to modify it.

Unsafe changes can cause TLB inconsistency in both uniprocessor and multiprocessor systems. Initially consider a multiprocessor system and, referring to Figure 8, suppose that valid entries exist that map page *a* to frame *A* in both the TLB accessed by processor *X* and the one accessed by processor *Y*. Further suppose that a process executing on processor *X* subsequently evicts page *a*, resident in frame *A*, and then maps page *b* to frame *A*, updating the TLB of processor *X* accordingly. Unless processor *Y* is prevented from using the inconsistent TLB entry for page *a*, it may erroneously access page *b* when attempting to access page *a*. Now consider processor *X* by itself. Whether in a uniprocessor or multiprocessor system, processor *X* faces the same problem as processor *Y* if it does not update its own TLB after modifying page *a*'s mapping and before issuing any other memory requests to page *a*.

Since the maintenance of page-reference history information is a separate issue and since TLB entries made inconsistent by safe changes can be detected and corrected by the operating system, in the remainder of this thesis the TLB consistency problem refers only to TLB inconsistencies caused by unsafe changes.

The general problem of maintaining the consistency of translation information is not unique to systems with multiple processors, TLBs, or demand paging because a copy of a page-table entry always exists, either explicitly or implicitly. For example, in a multiprocessor system without TLBs there is an implicit copy of a page-table entry stored in a register of a processor between the time the processor reads a page's address translation in-

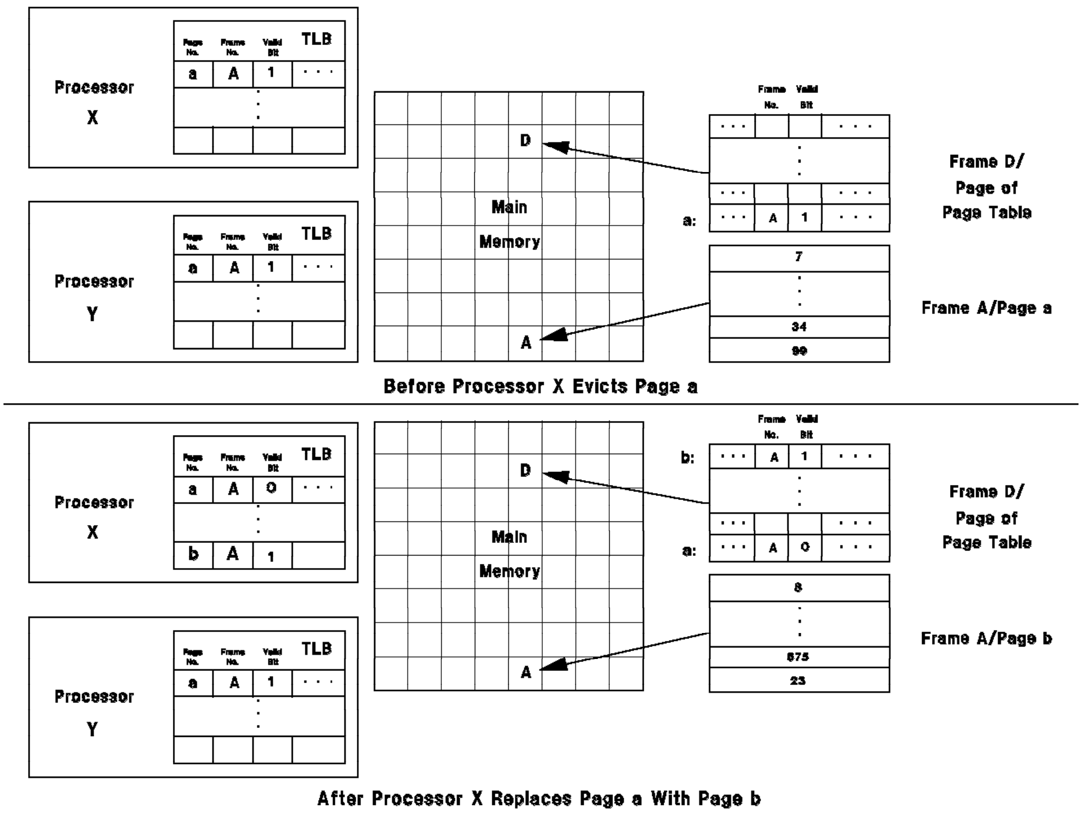


Figure 8. TLB Inconsistency in a Multiprocessor

formation and the time it accesses data in the referenced page. Due to the presence of the processor-to-memory interconnection network we must make the pessimistic assumption that an arbitrary number of memory accesses by other processors may occur between these two accesses. For example, in a large NYU Ultracomputer or IBM RP3 typical access time to shared memory would be on the order of one microsecond, but worst-case access time can be in the millisecond range. Thus, in the case of page eviction, for example, after one processor reads a page's translation information but before it accesses data stored in the page, another processor may evict the page and map another page to the related frame. Although this situation is highly improbable, the same problem with respect to other unsafe changes is not and situations like this must be considered to ensure the correctness of algorithms that propose to solve the TLB consistency problem.

2.4.1. Locking Translation Information

The TLB consistency problem represents an instance of the classical problem of obtaining a consistent view of a shared, complex data structure. References to the data structure, in this case the contents of a page and its translation information, must be done in an atomic fashion. For example, if a page is in the process of being remapped, no reader is allowed to observe an intermediate state where the page has been remapped but its translation information has not been changed to reflect the remapping (or vice versa). This situation may be violated when a copy of the page's translation information is cached in a TLB.

A classical solution to these kinds of problems adopts *readers-writers semantics*. If a process needs to obtain a consistent view of a data structure but does not intend to modify it, the process requests a *read lock*. An arbitrary number of read locks may be granted and, thus, an arbitrary number of readers may access the data structure, as long as no process is concurrently modifying it. If a process desires to modify the data structure, it requests a *write lock*. Only one write lock is granted at a time and a write lock is granted only when no read locks are active. Accordingly, only one process may modify the data structure at a time and during this time the data structure is not accessible to any other process.

When readers-writers semantics are employed, a reference to a resident page formally consists of four steps:

- obtain a read lock on the page-table entry;
- read the address translation information from the page-table entry;
- access the referenced memory location; and
- release the read lock on the page-table entry.

(A processor must hold a write lock on a page-table entry in order to modify it.)

In a uniprocessor these readers-writers semantics are implemented by not allowing any actions to occur between the access of a page's translation information and the subsequent access of the data stored in the page. As we demonstrate in this thesis, in a shared-memory multiprocessor these semantics are enforced by the algorithm that solves the TLB consistency problem or by its implementation, or they are encapsulated in atomic operations used by such an algorithm.

2.5. Generality of the TLB Consistency Problem

The more general the definition of shared memory, the more general the problem of TLB consistency. The flexibility with which memory may be shared among processes can be constrained by the way the operating system defines and/or manages virtual memory and by restrictions placed on the virtual-to-physical mappings of pages by the architecture. A restriction in the sharing patterns allowed in a multiprocessor system translates into a restriction in the number of ways TLB inconsistencies can arise. In general, the fewer the ways that TLB inconsistencies arise, the easier it is to solve the TLB consistency problem.

Parallel processing has motivated operating systems and application programs to use virtual memory in more sophisticated ways and, thus, define shared memory in more general terms. For example, virtual memory operations are being used by operating systems to support interprocess communication between related and unrelated processes, read/write and copy-on-write sharing, and memory-mapped files. Interprocess communication is being implemented using copy-on-write sharing [Rashid, *et al.*, 1987] and by remapping virtual memory regions [Anderson, *et al.*, 1988]. *Copy-on-write* sharing, sometimes referred to as virtual copying, allows a page to be shared, as a read-only page, by multiple processes until one of the processes attempts to modify it. Thus, the same page, while in the read-only state, can be associated with different virtual addresses in different address spaces. When a process attempts to modify the page, the contents of the page are copied to a newly-allocated page in the address space of the modifying process. In addition, page-protection techniques are being used to implement concurrent algorithms such as concurrent garbage collection and concurrent checkpointing [Appel and Li, 1991].

General memory sharing patterns are produced when:

- the mapping of pages to frames is unrestricted and can be many-to-one;[2]
- parallel execution within the same address space is supported, *i.e.*, multiple processes can generate references to the same set of virtual pages; and
- operations on remote address spaces are permitted, *i.e.*, the modification of a page-table entry by an executing process may alter the status of a virtual page that can be referenced by other executing processes.

When the use of shared memory is defined with this flexibility, the TLB consistency problem is defined in its fullest generality if, in addition,:

- processes are allowed to migrate among processors,
- multiprogramming is implemented, and
- TLBs are not flushed on context switches.

A solution to the problem when posed in this way is a fully general solution. It prevents the use of a TLB entry made inconsistent by any unsafe change to a page-table entry. In particular, it guarantees consistency when one processor changes the status of a page that is accessible to another processor.

Similar TLB inconsistencies can arise whether processes execute in private address spaces or shared address spaces. In either case translation information for the same page can be present in more than one TLB. For example, in either case if processes migrate among processors without flushing TLB entries, translation information can remain in the TLB of the processor from which a process migrates, while also being resident in the TLB of the processor to which it migrates. Another process that becomes resident on the

[2] Note that when a many-to-one mapping of pages to frames exists then it must be ensured that when an unsafe change is made to a shared page, all the related address translation data are modified accordingly. This may translate into a sequence of unsafe changes.

processor from which the process migrated may not replace the translation information, which was left behind, during its normal course of execution.

As shown above, the architecture and the operating system define the generality with which memory is shared and, thus, the context in which the TLB consistency problem is to be solved. In addition, the architecture may not be amenable to an implementation of a particular solution. For example, the solutions that we describe in the next chapter are restricted to multiprocessors with efficient broadcast capability.

Part II: Solutions

This part of the thesis includes:

- an outline of the strategies that can be used to solve the TLB consistency problem,
- a discussion about the factors that determine the suitability of a solution for a given multiprocessor system,
- a description of TLB consistency solutions that are suitable for multiprocessors with efficient broadcast capability,
- a discussion of related performance issues, and
- a description of TLB consistency solutions that are suitable for multiprocessors with more general interconnection networks and, thus, inefficient broadcast capability.

Chapter 3. Solutions for Bus-Based Multiprocessors

In this chapter we begin to explore various methods that can be used to solve the TLB consistency problem in multiprocessors. In particular, we

- classify the strategies that have been employed to solve the problem,
- discuss the points to consider when matching a solution to a multiprocessor system, and
- show how the most general TLB consistency problem can be solved in a multiprocessor with efficient broadcast capability.

Section 3.1 begins this chapter by defining three strategies that have been employed to solve the problem. An avoidance strategy avoids the problem. An invalidation strategy invalidates TLB entries when they become inconsistent. And, a detection strategy detects stale translation information before it is used to access memory. The latter two strategies have been used to develop fully general solutions, *i.e.*, solutions to the most general TLB consistency problem. Two of the solutions presented in Section 3.3, two presented in Chapter 4, and one presented in Chapter 5 employ an invalidation strategy and solve the most general TLB consistency problem. One of the solutions of Chapter 5 employ a detection strategy and is a fully general solution.

Section 3.2 discusses issues to consider when evaluating the suitability of a solution for a multiprocessor system. In particular, we explore the influence that the architecture, the operating system, and the targeted application programs have on the way in which the TLB consistency problem is cast. The way in which processors and memory are interconnected can limit the types of solutions that can be employed. For example, solutions that require the broadcasting of data modifications are not suitable for a system with a multi-stage interconnection network. Parallelism inherent to a solution is attractive for multiprocessors since unsafe changes can be generated and TLB entries can be updated in parallel. For a system with a restricted definition of memory sharing a less general solution is favored over a fully general solution if its overhead is less. Finally, the parallelism and memory sharing inherent in the software targeted to a system may stress some solutions in the sense that their overheads may affect the speedup in execution time that would be achievable otherwise.

Section 3.3 presents three solutions that require data modifications to be broadcast to all processors. Consequently, at the present time they are suitable only for bus-based multiprocessor systems, for which they have been proposed. All three of these solutions [Wood, *et al.*, 1986; Goodman, 1987; and Cheriton, *et al.*, 1986] assume virtual-address general-purpose caches kept consistent by hardware and employ an invalidation strategy. Two of them solve the most general TLB consistency problem.

In a bus-based multiprocessor, these solutions have a small impact on performance because the execution of a processor is rarely interrupted by consistency-ensuring actions, additional communication is not required to maintain consistency, and processor synchronization is not necessary.

3.1. Strategies

In a computer system with one TLB, the TLB consistency problem can be solved easily by blocking access to the TLB until a page-table modification is complete and the TLB is updated accordingly. In a uniprocessor this is accomplished by the operating system inhibiting *context switching*, *i.e.*, switching execution from one process to another, from the time the processor begins to modify a page-table entry until the modification is complete and any necessary updates to the TLB are made.

This method of ensuring TLB consistency does not work in a multiprocessor with multiple TLBs unless an image of the translation information for a page exists in only the TLB of the processor modifying the page-table entry. Inhibiting the context switching of the processor that is modifying the page table ensures only the consistency of the TLB of that processor and, thus, ensures the consistency of only one image of the translation information stored in TLBs. Thus, other methods of ensuring TLB consistency must be employed in multiprocessor architectures with multiple TLBs. Since we have dismissed the TLB consistency problem in uniprocessors and multiprocessors with a central TLB [Stanford, 1985; McCreight, 1984], in the rest of this thesis whenever we refer to the TLB consistency problem, we mean the TLB consistency problem in shared-memory multiprocessors with multiple TLBs.

Before we examine various solutions to the TLB consistency problem that have been proposed in the literature, we describe the three main strategies they employ:

- avoidance,
- invalidation,
- and detection.

Techniques that employ an avoidance strategy avoid the TLB consistency problem by restricting unsafe changes to translation information that is not resident in multiple TLBs. Although this strategy may be effective in some systems, in others it may result in a situation where an unsafe change may be postponed indefinitely. For example, suppose multiple TLBs contain an entry for page *a* and during the normal course of execution these TLB entries are not replaced. If this situation continues to exist, no unsafe change to page *a*'s translation information can be effected. Thus, a solution to the TLB consistency that employs an avoidance strategy may be only a partial solution in some systems. Partial solutions, however, may improve the effectiveness of other fully general solutions and, thus, should be considered as adjuncts. Read-locked TLBs, a solution described in Chapter 5, employs an avoidance strategy and is an example of a solution that should be considered as an adjunct to a fully general solution to the TLB consistency problem.

Algorithms based upon an invalidation strategy solve the problem by preventing TLB reloads of translation information that is in the process of being modified and by invalidating TLB entries when they become inconsistent. For example, the IBM System/370 includes an instruction, *ipte*, that atomically invalidates a page-table entry and flushes it from all the TLBs in the system. Since in a multiprocessor it is not usually possible for one processor to directly flush entries in the TLBs of other processors, other methods of accomplishing this task have been proposed. Akin to the S/370 solution, some of the proposed solutions use an invalidation strategy and, in doing so, solve the most general problem. Two of these, Multi-tagged Caches and VMP Caches, are described in Section 3.3 and two others, TLB Shootdown and Modified TLB Shootdown, are described in Chapter 4. The first two are restricted to multiprocessors with efficient broadcast capability, but the latter two can be employed in a larger class of multiprocessors, including those with more general interconnection networks. Memory-based TLBs, described in Chapter 5, also uses an invalidation strategy and is a fully general solution to the TLB consistency problem if only one paging arena is employed. (The use of multiple paging arenas restricts the virtual-to-physical mapping of pages to frames.) In other cases, such as in the case of Lazy Devaluation, which is described in Chapter 4, a solution is achieved that, although adequate for the targeted system, is not a fully general solution.

Solutions that use a detection strategy are able to identify whether or not a TLB entry is inconsistent. This can be done by tagging TLB entries with unique identifiers, similar to time stamps, that indicate whether or not an entry is stale. TLB entries can be allowed to become inconsistent since stale entries can be identified when used to access memory and, when identified, can be invalidated and updated. A fully general solution that employs a detection strategy, Validation, is described in Chapter 5.

3.2. Solution/Multiprocessor Match

The suitability of a particular solution for a shared-memory multiprocessor system depends on:

- the multiprocessor architecture,
- the operating system, and
- the classes of application programs targeted to the system.

The architecture defines the size of the multiprocessor, *i.e.*, the number of processors and memory modules in the system, as well as the interconnection among them. Some solutions, such as those described in this chapter, rely upon the efficient broadcast of data to processors. Thus, at the present time, these solutions are restricted to bus-connected multiprocessor architectures, where the number of processors is limited by the available bus bandwidth. These solutions and algorithms are not suited for multiprocessors with more general interconnection networks because the broadcasting of data modifications cannot be efficiently implemented in such architectures. Other solutions, however, can be employed in architectures that employ more general interconnection networks, such as multi-stage networks. These are described in Chapters 4 and 5.

In a bus-based architecture only inconsistencies caused by one page-table modification are corrected at any one time. In contrast, an architecture with a more general interconnection network allows inconsistencies to be generated in parallel and inconsistencies caused by a number of page-table modifications to be corrected in parallel. The amount of parallelism that can be exploited in consistency-ensuring management depends upon the adopted solution.

As discussed in Section 2.5, the architecture and the operating system define the flexibility with which shared memory is used and, thus, the generality with which the TLB consistency problem is posed. Unrestricted use of shared memory implies that the mapping of virtual to physical memory is not limited in any way and that the translation information associated with any virtual memory region can be modified by any process. The TLB consistency problem is posed in its most general form when the flexibility of shared memory is unrestricted and a fully general solution solves the problem when it is defined in this way. Although a fully general solution is attractive in the sense that it can be employed in a range of multiprocessor architectures, a fully general solution is not necessary if the architecture and the operating system for which the problem is being solved restrict the use of shared memory. In this case, a less general solution with a smaller overhead is favored over a fully general solution with a larger overhead.

The other factor that influences the suitability of a solution for a particular multiprocessor architecture is the application programs that are intended to be run on the system. The efficiency of a solution can be affected by:

- the parallelism exhibited by an application,
- the extent to which pages are shared among processes,
- whether an application makes intensive use of private memory or shared memory, and
- the frequency of unsafe changes.

For example, in a large-scale multiprocessor, where there are thousands of processors, extensive sharing of pages among processes coupled with a high degree of parallelism demands a solution that does not have an adverse affect on the speedup that can be achieved for a specific program.

With these points in mind, we explore three hardware solutions to the TLB consistency problem which are suitable only for multiprocessors with efficient broadcast capability, in particular bus-based multiprocessors with virtual-address caches. Each of these solutions uses an invalidation strategy, which is based on the following precepts:

- protect the integrity of page-table entries by implicit or explicit read/write locks,
- prevent the reload of inconsistent entries, and
- invalidate entries in remote TLBs when they become inconsistent.

Two out of the three solutions solve the most general problem.

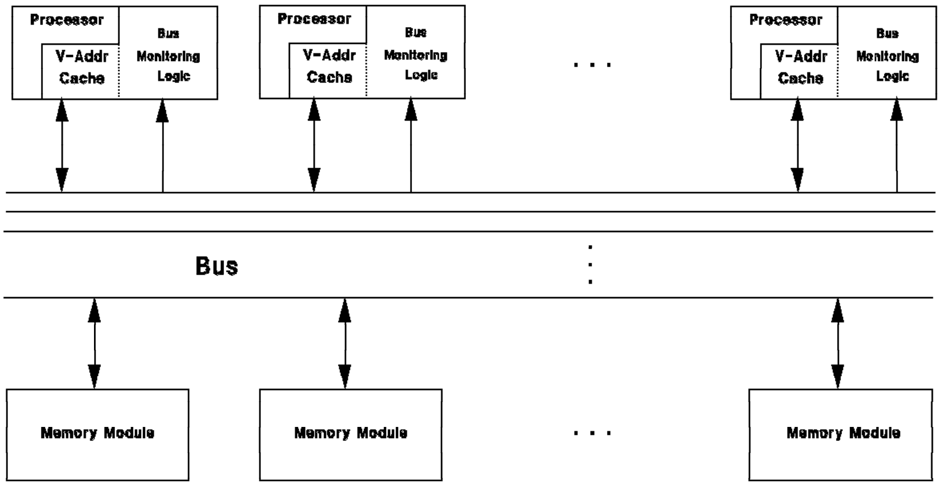


Figure 9. Bus-Based Multiprocessor with Virtual-Address Caches

3.3. In-Cache Translation

The solutions described in this section solve the TLB consistency problem while solving the virtual-cache consistency problem [Knapp, 1985].

The TLB consistency problem is incorporated into the general cache consistency problem by using “in-cache translation” [Ritchie, 1985]. Virtual-to-physical address translation is accomplished by accessing page-table entries from the virtual-address general-purpose cache, rather than from a TLB. Would-be TLB entries stored in the cache, along with other cache-resident data, are kept consistent via a snoopy-cache consistency protocol (*cf.*, [Sweazey and Smith, 1986]). The use of a virtual-address cache gives rise to the *address synonym problem* [Smith, 1982] when the mapping from virtual to physical addresses is not one-to-one, *i.e.*, two distinct virtual addresses map to the same physical address.

In systems with virtual-address caches, part of a page's translation information is replicated in associated cache blocks. When a page-table entry is modified, in addition to causing cached copies of the entry to be out-of-date, the associated translation information stored in cache blocks is caused to be out-of-date. This problem is called the *virtual-cache consistency problem*. To solve this problem, blocks from the subject page must be invalidated or updated. Although the coordination of cache and TLB management are beyond the scope of this thesis, we briefly discuss this in Section 5.1, which describes our multiprocessor model.

The implementation of the solutions described in this section employ snoopy-cache techniques and protocols and, thus, require that the processors and memory be interconnected by a bus. Accordingly, the targeted bus-based multiprocessor architectures are similar to the one depicted in Figure 9.

3.3.1. SPUR Caches

The solution of Wood, *et al.*, [1986], which we call *SPUR Caches*, is being implemented as part of the SPUR workstation project [Hill, *et al.*, 1986] at U.C. Berkeley. SPUR is a shared-bus multiprocessor architecture that has a virtual-address cache per processor. The snoopy-cache consistency protocol described by Katz, *et al.* [1985] is implemented via snooping caches and a dual-address bus (both the virtual and physical addresses are transmitted with a bus transaction). The snoop compares the virtual addresses of all bus transactions with the cache address directory. The synonym problem is avoided by providing a single global virtual address space that is shared by all processes and by restricting page sharing among multiple processes in the sense that processes that share a page access it with the same virtual address. The adopted snoopy-cache consistency protocol guarantees the consistency of cached page-table entries since page-table modifications are visible to all caches. If virtual-to-physical translation is performed whenever memory is accessed, then no erroneous references to memory are made.

A problem arises, however, when virtual memory is deallocated. Before the virtual page numbers associated with the deallocated memory are reassigned, it must be ensured that no cache contains entries associated with the deallocated memory. The proposed solution to this problem is to postpone the unsafe change until all caches are flushed. The following algorithm is suggested:

1. instruct each processor to flush its cache by sending it a message or interrupt;
2. each processor atomically increments a counter in shared memory when it completes its flush;
3. it is safe to make the change when the value of the counter equals the number of processors.

As mentioned above, SPUR Caches eliminates the synonym problem by restricting sharing, *i.e.*, a page shared by multiple processes has the same virtual address in each process' address space. With full generality of sharing, two distinct virtual pages can map to the same physical page, even with a single process running on a single processor. Thus, SPUR Caches is not a fully general solution to the TLB consistency problem. This approach, however, does lead to a fully general solution if the following idea of Goodman [1987] is adopted.

3.3.2. *Multi-tagged Caches*

Goodman proposes a snoopy cache that has two cache directories, thus, we call this solution, *Multi-tagged Caches*. The directory accessed by the processor is accessed by virtual address and can be accessed in parallel with the memory that stores the cached data. The directory accessed by the bus-monitoring logic, which stores a tag for each line in the data memory, is accessed by physical address. Each tag in this directory includes information that, together with the page offset to the cache block within the page, uniquely defines a line in the data memory. Thus, to access data via the physically-addressed cache directory requires an additional cycle, in comparison to accessing data via the virtually-addressed cached directory. Goodman assumes a cache consistency scheme appropriate for a Futurebus implementation [Sweazey & Smith, 1986] and additional hardware to translate the virtual address to a physical address when translation is necessary. In this case, we assume that the translation information can be accessed from the virtual-address cache or from main memory. The system binds and unbinds virtual addresses to physical addresses in response to bus transactions, and in doing so solves the synonym problem.

3.3.3. *VMP Caches*

The solution of Cheriton, *et al.*, [1986] has been incorporated in the design of the VMP multiprocessor, an experimental shared-memory bus-based multiprocessor being built at Stanford University. This solution, which we call *VMP Caches*, assumes that an address space identifier is included as part of the address presented to the virtual-address cache. The caching of a page is initiated by a cache miss. A bus monitor (a simple but specialized state machine) is connected to each processor by a nonmaskable interrupt and a first-in, first-out (FIFO) queue of interrupt requests. The primary use of the bus monitor is to maintain cache consistency. The bus monitor's action table contains information for each cached physical page frame. Each bus transaction triggers the bus monitor to perform one of four actions which depend on the type of bus transaction, the physical address of the transaction, and the contents of the action table. The consistency of the virtual-address caches is maintained by a variant of the distributed ownership protocol described by Frank [1984] and Goodman [1983]. The action of the bus monitor is keyed to the physical page number rather than the virtual page number, therefore, the synonym problem is solved since each processor observes the consistency protocol with respect to all the entries in its cache that map to the physical page. The TLB consistency problem is solved by each processor following the procedure described below when it wishes to make an unsafe modification to a page-table entry.

1. The processor first gains an exclusive copy of the cached page containing the page-table entry.
2. Then the processor attains ownership of the physical page to which the virtual page is mapped.

3. When the processor gains ownership of the page, each copy of the physical page cached in another processor is flushed or written back and no other processor can access the page until the ownership is relinquished.
4. The processor updates the page-table entry.
5. Finally, the processor relinquishes ownership of the two pages.

Since all unsafe changes can be handled in this fashion, this solution is fully general.

3.3.4. Summary

As we have shown, by encompassing the problem of TLB consistency into the general cache consistency problem, bus-watching techniques can be used to achieve fully general solutions to the TLB consistency problem. Even though the solutions described in this chapter require virtual-address caches, it is not hard to devise a similar solution for bus-based multiprocessors with physical-address caches and separate TLBs. In this case, page-table modifications must be recognizable so that they trigger coordinated consistency-ensuring responses from both the TLB and cache of each processor.

These solutions have a very small affect on performance. The overhead associated with such solutions is very low because of the following reasons:

- A processor is not interrupted to perform actions required to maintain consistency. Each processor's bus-watcher acts independently of the processor, updating would-be TLB entries when necessary. (Notice, however, that some penalty may be realized as a result of competition between the processor and the bus-monitoring logic.)
- Additional communication is not required in order to maintain consistency. The bus allows all page-table modifications to be visible so that when a transaction occurs that endangers consistency, specific measures dictated by the consistency protocol can be applied by the bus-monitoring hardware to maintain cache consistency.
- Processor synchronization is not necessary since the bus serializes transactions and the consistency protocols used by these solutions do not allow concurrent page-table modifications and use of potentially stale translation information.

Readers-writers semantics are encapsulated in the protocol used to maintain cache consistency and in the serialization enforced by the bus.

These solutions have low computational overhead when implemented in systems with bus-watching hardware. Since their efficient implementation requires that page-table modifications be broadcast to processors, they cannot be employed efficiently in scalable multiprocessor systems, such as those with multi-stage interconnection networks. TLB consistency solutions that can be employed efficiently in shared-memory multiprocessors with more general interconnection networks, such as multi-stage networks, are examined in Chapter 4 and 5. Although the small performance overhead of the solutions presented in this chapter is not achieved by the solutions presented in the next two chapters, it gives us a base for comparison.

Chapter 4. Solutions for More General Multiprocessors

The TLB consistency solutions presented in Chapter 3 require only the processor making an unsafe change to participate in the algorithm, and its participation per unsafe change is small and constant; the bus monitors maintain consistency essentially independent of the processors. The small performance overhead associated with these solutions makes them very attractive. However, such solutions are suitable only for bus-based multiprocessors with bus-monitoring logic.

Alternate solutions, such as those presented in this and the next chapters, are needed for bus-based multiprocessors without special hardware that ensures cache and TLB consistency and multiprocessors with more general interconnection networks, such as multi-stage networks. In a bus-connected multiprocessor memory requests are broadcast, one at a time, on a single shared bus. In contrast, in a multiprocessor with a multi-stage network interconnecting processors and memory, memory requests are concurrently generated and transmitted along different paths of the network. Thus, it is impossible for a hardware device at a processor to monitor all memory requests. In addition, the algorithms on which the solutions of Chapter 3 are based require the broadcasting of data modifications, which cannot be supported efficiently in a multiprocessor with a multi-stage interconnection network. As a result, extra communication and/or synchronization are required to ensure TLB consistency in such an architecture, and solutions to the TLB consistency problem have more of an impact on performance than solutions like those presented in Chapter 3.

The solutions of this and the next chapters can be classified into two groups based on the nature of their performance overheads, which we define to be the processor time, both busy and idle, that is expended in order to ensure TLB consistency. The first group consists of the solutions proposed by other researchers, which are presented in this chapter. Their overheads grow with the number of unsafe changes and the number of processors participating in the algorithm. The second group consists of our solutions, which are presented in Chapter 5, have overheads that grow with the number of TLB reloads. These solutions require that a processor or processors invoke either a software or hardware mechanism in conjunction with each unsafe change and/or each TLB reload. As described in Section 4.1.1, when employed in a multiprocessor with a multi-stage interconnection network, the overheads of these solutions depend on the dimensions of the network, *i.e.*, log N, where N is the number of processors in the system. This is because the cost of the communication and/or synchronization necessary to implement one of these solutions depends on the cost of network transit.

The overhead associated with a solution is important since it can degrade system performance if the overhead causes the attainable speedup in program execution to decrease. Thus, especially in the case of scalable architectures, it is important to examine the overheads of solutions with respect to how they increase with N. In Section 4.1.2 we illustrate the potential affect of a solution's overhead on program speedup. We contrast how program speedup is affected when the problem size remains fixed as N increases versus when the problem size increases with N.

Solutions for multiprocessors with more general interconnection networks that have been proposed by other researchers are presented in Section 4.2 of this chapter [Black, *et al.*, 1989; Rosenburg, 1989; and Thompson, *et al.*, 1988]. These solutions require interprocessor synchronization and communication, interrupt processor execution, and introduce serialization. They have the flavor of directory-based methods of ensuring cache consistency in the sense that they require a record of the TLBs that may have entries for each memory-resident virtual page. (Although it is not clear that directory-based methods will be efficient for ensuring cache consistency in large-scale systems, their use may provide efficient solutions to the TLB consistency problem. The direct use of these methods to solve the problem has not been explored as of yet.)

The amount of interprocessor communication and synchronization required by these solutions at each unsafe change is of $\Theta(P)$, where P is the number of processors participating in the algorithm. For a system with a multi-stage network this overhead is $\Theta(P \log N)$ since the cost of the network transit required to perform the necessary communication and synchronization is $\Theta(\log N)$. If P grows linearly with N then this overhead is $\Theta(N \log N)$. If the number of unsafe changes is small then a $P \log N$ overhead per unsafe change may not be detrimental to performance. However, if the number of unsafe changes is large then a solution with such an overhead may adversely affect performance.

Since we are especially interested in scalable architectures we evaluate these solutions, as well as our own, with respect to their potential impact on the performance of large-scale systems. Thus, before we present the solutions, we digress in Section 4.1 with a discussion of the performance overhead of solutions to the TLB consistency problem and their impact on program speedup.

4.1. Performance Considerations

A solution's performance overhead over a range of multiprocessor architectures is difficult to determine because it is program and operating system dependent. For example, the performance overheads of the solutions that are described in this chapter depend on the number of unsafe changes that occur during a program's execution and the number of processors that participate in the algorithm. The performance overheads of our solutions, described in Chapter 5, depend on the number of TLB reloads that occur during a program's execution. Without a good estimate of the number of unsafe changes and the amount of page sharing that occur during a program's execution, it is difficult to evaluate the solutions that have been proposed by other researchers and are presented in this chapter. However, as illustrated in Section 4.1.1, we can characterize the overheads these solutions, as well as our own, and by doing so we are able to make good estimates of the overheads that would be incurred in systems of different sizes under a range of assumptions. In addition, since the performance overheads of all three of our solutions are defined in a similar way, we are able to compare them by using the results of a trace-driven simulation study. (Our simulation system and the results of our study are discussed in Chapters 6, 7, and 8.)

System performance is degraded by the overhead associated with a solution to the TLB consistency problem if the overhead causes the attainable speedup in program execution time to decrease. As shown in Section 4.1.2, the affect of a solution's overhead on a program's speedup may differ depending on how speedup is measured, *i.e.*, whether speedup is measured using a fixed size problem or a problem with a size that grows with *N*, the number of processors in the system.

4.1.1. Solution Overhead

Many of the components that comprise a solution's overhead are interrelated. Some of these represent computational costs, while others represent side effects of adopting the solution. Among the factors that contribute to a solution's overhead are:

- processor time expended to execute the algorithm on which it is based,
- synchronization inherent in the algorithm,
- related communication among system components,
- changes in page-in, page fault, and TLB miss rates,
- foregone uses of time-saving optimizations, and
- lack of parallelism inherent in the adopted algorithm, *e.g.*, the inability to perform up to *N* different unsafe changes in parallel.

In this thesis we evaluate and compare the computational overheads of solutions. We refer to a solution's *computational overhead* as *OH* and define it to be the processor time, both busy and idle, that is consumed by participating in the algorithm that ensures that memory is accessed using only consistent TLB entries. The other components of a solution's overhead constitute the performance cost of side effects that occur as a result of adopting the solution. For example, if the overhead causes an increase in the execution time of the individual tasks comprising a program, then the scheduling of these tasks may be altered. As a result, TLB miss rates may change.

In this and the next chapters, we describe the potential impact of each solution's overhead in different computing environments. We do this by studying the algorithm on which each solution is based and the way in which the solution is implemented, and generating an expression for the solution's overhead that is used to evaluate the solution's performance in systems of different sizes under a range of assumptions. As discussed in Chapters 6, 7, and 8, we use a simulation study to evaluate the computational overheads of our solutions and to determine the changes in page-in, page-fault, and TLB-miss rates as *N* (the number of processors) and the network dimensions of a multiprocessor architecture increase. Our simulator does not determine quantitatively other side effects such as scheduling changes, network contention, the forfeiture of time-saving optimizations, and serialization that result from adopting a particular solution to the TLB consistency problem. However, in this and the next chapters we do identify all the side effects of each solution.

The overheads of the solutions presented in this chapter can be expressed, in general, as

$$OH = \beta \times (T_{change} + P \times T_{participate}),$$

where β is the number of unsafe changes that occur, T_{change} represents the average time, both busy and idle, expended by a processor making an unsafe change, $T_{participate}$ represents the average time expended by each processor that also participates in the algorithm, and P is the average number of participating processors. The computational overhead depends on N since in a multiprocessor with a multi-stage network interconnecting processors and memory, the processor time expended in ensuring TLB consistency grows with N because the network transit time grows with the dimensions of the network. The values of β and P are operating system and program dependent and may also depend on N. For example, the values of β and P depend on:

- the parallelism inherent in the operating system and application programs,
- the generality with which memory sharing is defined, and
- the number of processors that cooperate in the execution of the program(s).

Thus, the computational overhead of such a solution cannot be determined fully without a representative average workload with both user and operating system references. Such a workload for multiprocessor systems is not available yet.

The performance overheads of our solutions to the TLB consistency problem can be expressed generally as

$$OH = \beta \times T_{change} + R \times T_{reload,}$$

where β is the number of unsafe changes that occur, R is the number of TLB reloads that occur, and T_{reload} represents the average time it takes for a processor to execute that part of the TLB-reload routine that is unique to the solution. As in the case described above, OH depends on N. Although T_{change} may have a different value than T_{change} in the case described above, in both cases it is the other component of OH that is significant, *i.e.*, $\beta \times (P \times T_{participate})$ and $R \times T_{reload}$. In contrast to the former, $R \times T_{reload}$ does not depend on the number of unsafe changes that occur during program execution, but rather on the number of TLB reloads that occur. Without a representative average workload, we do not have a good estimate of R and do not know how it grows with N on average. However, as described in Chapter 8, our simulation study has allowed us to determine this information for at least two different programs running in various simulated computing environments.

A comparison of the computational overheads of solutions does not lead us to a conclusion as to which solutions will perform best in large-scale systems. This is because the components of these overheads that depend on β and P or R are program dependent. Note, however, that unlike the value of β, the value of R can be influenced by changing the TLB size. The values of β and R can be influenced by changing the page size. The value of β may also be influenced by batching unsafe changes [Appel and Li, 1991].

4.1.2. Performance Impact

As explained in [Teller, 1990], it is important to examine the overhead of a TLB consistency solution because it can have an impact on system performance. Accordingly, for a scalable multiprocessor system it is important to examine how a solution's overhead increases with N. In doing this, questions arise with respect to the adaptability of a solution: Can the solution solve the problem for an architecture with tens, hundreds, or thousands of processors without impeding performance? Does a solution's overhead degrade the speedup attainable by application programs? These questions are clearly program dependent and, thus, can be answered only on a program-by-program basis. In addition, as we illustrate below, the answers to these questions may depend on how a program's speedup is measured, *i.e.*, in terms of a fixed problem size or in terms of a problem with a size that increases with N.

For a particular multiprocessor architecture running only one program let $T_1(p)$ represent the time it takes to execute a program that solves a problem of size p on one processor. And, let $T_N(p)$ be the time it takes on N processors. Ideally, the speedup achieved by running the program on N processors is N. This implies that the speedup grows linearly with N or the speedup scales with N. Symbolically the speedup achieved on N processors is expressed as

$$S_N(p) = \frac{T_1(p)}{T_N(p)} = \alpha N, \qquad (4-1)$$

where α is called the efficiency of the parallel execution. The impact of the overhead due to running on N processors causes α to fall away from unity. In the ideal case, α remains equal to 1 as N grows, implying that the overhead is essentially 0.

In practice, when a program runs in parallel on a multiprocessor architecture, such an ideal speedup is rarely achieved. That is, usually $\alpha<1$. This is true for several reasons. For example,

- the computational load is rarely balanced among the processors cooperating in the execution of the program, thus, processors may be idle while waiting for other processors to complete their allocated tasks;
- the contention among processors within the interconnection network and at memory modules may idle processors that are accessing memory;
- processors encountering page faults at the same time may be idled if there are no runnable tasks to execute; and
- communication and synchronization among processors may be required during the execution of the program.

If program performance on a multiprocessor is measured using the definition of speedup given by Equation (4-1), then the value of α is affected by the overhead of a solution to the TLB consistency problem. Thus, speedup may be affected also by the way in which the TLB consistency problem is solved.

There is a continuing controversy with respect to the way program speedup should be measured (*cf.*, [Gustafson, 1988; Heath and Worley, 1989; and Zhou, 1989]). For example, the effective performance of a multiprocessor can be viewed in terms of either a fixed problem size or a problem size that grows with N. Given a problem of sufficient size, Gottlieb [1987] views the effective performance of a highly-parallel computer in terms of this fixed-size problem. He explains that the hardware and software design of a highly-parallel computer should scale upward to a very high level, *i.e.*, given a problem of sufficient size, a k-fold increase in the number of processors should yield a speedup factor of almost k. Adopting the definition of speedup given by Equation (4-1), this means that for a problem of sufficient size, a multiprocessor architecture with N processors should be able to execute the program that solves the problem N times as fast as the program can be executed by one processor. Thus, as N grows, the total time that processors are idle, busy-waiting, or doing nonjob related work should remain relatively constant. Forgetting the side effects resulting from adopting a solution, this means that for a given problem of fixed size as N increases ideally OH should remain constant. If OH remains constant, OH per processor grows as $\frac{1}{N}$. This situation is illustrated in the top part of Figure 10, where a program's gross execution time is equal to its net execution time plus the overhead incurred by a TLB consistency solution. Since the solutions presented in this chapter incur an overhead only for $N > 1$, in Figure 10 we do not associate a TLB-consistency-related overhead with a one-processor system and assume that OH does not significantly impact the speedup achieved by two processors. As the top part of this figure illustrates, if OH remains constant then the program's speedup remains relatively unchanged. Referring to Equation (4-1) this means that OH has a negligible impact on α and speedup scales with N.

In contrast, if we assume that OH grows linearly with N, *i.e.*, OH per processor remains constant as N increases, then, as shown in the bottom part of Figure 10, a program's speedup and, thus, system performance are adversely affected. In this case, as N increases, speedup is reduced and no longer grows linearly with N.

With respect to solutions for which $OH = \beta \times (T_{change} + P \times T_{participate})$, given that T_{change} and $T_{participate}$ grow as $\log N$, if β and P remain constant or β grows as $\frac{1}{N}$ and P grows linearly with N, then OH remains relatively constant as N increases and speedup scales with N. In contrast, if β remains constant and P grows linearly with N, then OH grows linearly with N and speedup does not grow linearly with N. It is not clear how β and P grow as N increases. And, it is not clear what the magnitude of β is. Given a specific multiprocessor architecture and operating system, β and P may increase or decrease as N increases depending on the program(s) being executed. For highly-parallel programs with a large amount of page sharing it is likely that P will grow linearly with N. If $P = N$ and β remains constant then we have the situation illustrated in the bottom part of Figure 10, where a program's speedup is reduced so that it no longer grows linearly with N. In this case OH if $\Theta(N \log N)$.

With respect to our solutions for which $OH = \beta \times T_{change} + R \times T_{reload}$, given that $\beta \times T_{change}$ is inconsequential in comparison to $R \times T_{reload}$, for OH to remain relatively constant as N in-

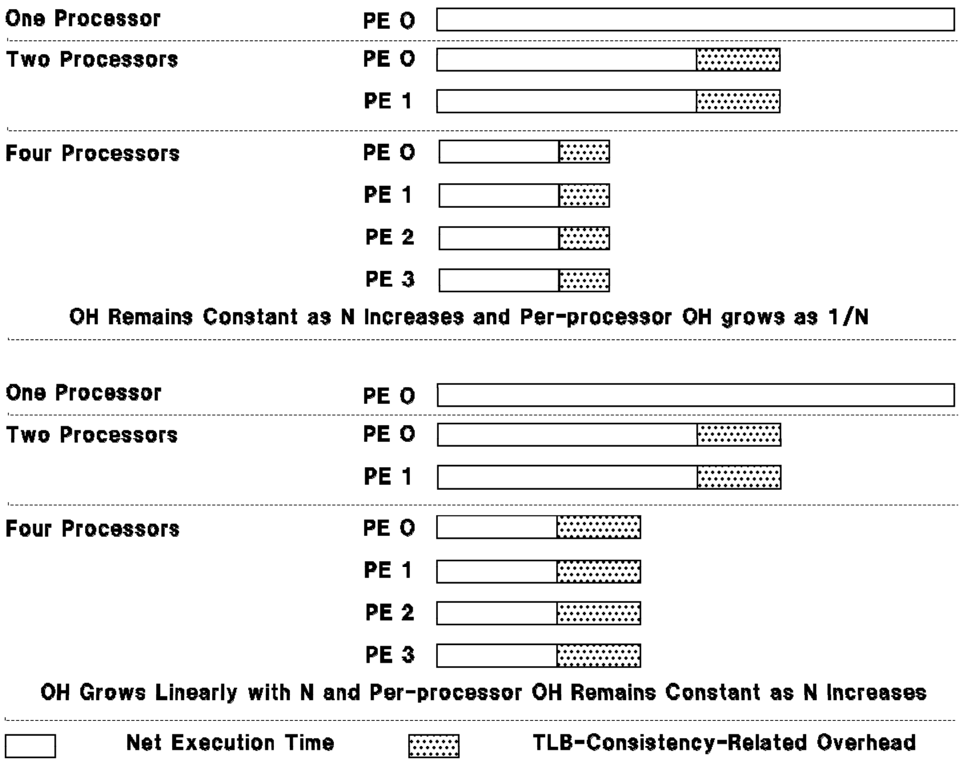


Figure 10. Execution Time vs. N with Fixed Size Problem

creases, it is generally true that *R* must remain constant or decrease as *N* increases. In this case, since T_{reload} grows as log *N* in a system with a multi-stage interconnection network, the overhead of our solutions for processor-based TLBs is $\Theta(\log N)$ and a program's speedup is not significantly affected. However, for highly-parallel programs with a large amount of page sharing it is unlikely that *R* remains constant or decreases as *N* increases because as *N* increases more TLB contexts exist and, thus, more TLB reloads are likely. In this case, *OH* grows as $R \log N$. Thus, if *R* grows linearly with *N*, which is the case for a class of parallel programs (see Section 8.2.1), then *OH* grows as $N \log N$. As shown above an *OH* of $\Theta(N \log N)$ can adversely affect a program's speedup. As shown in Chapter 8, the computational overhead of our third solution, which assumes memory-based TLBs, grows with *R*, and the growth of *R* depends on *N* and the employed memory organization. In contrast to our processor-based TLB consistency solutions, it is not so easy to characterize the growth of *OH* for our third solution, Memory-based TLBs. However, it looks promising that certain implementations of Memory-based TLBs may result in an *OH* that will not significantly reduce a program's potential speedup.

As mentioned above and as described by Hoshino [1989], performance can be measured in terms of the behavior exhibited by the system as *p*, the size of the problem being solved, grows linearly with *N*. The average amount of net execution time associated with each of the N processors is fixed regardless of the value of p. Thus, if p_1 is the size of the problem executed on a one-processor system and p_N is the size of the problem run on an N-processor system, $p_N = N \times p_1$. In this case, as depicted in the top part of Figure 11, the computational overhead may grow linearly with the problem size and not affect a program's speedup.

As shown in the top part of Figure 11, in this case, again ignoring the overhead due to the side effects that result from adopting a solution, if OH grows linearly with N, *i.e.*, if OH per processor is constant, then a program's speedup is unaffected. With respect to the solutions with $OH = \beta \times (T_{change} + P \times T_{participate})$ this situation occurs if β remains constant and P increases linearly with N. Using the model of program speedup where the program size is fixed these same assumptions resulted in an adverse affect on speedup. However, as the problem size, p, increases with N, it seems likely that β and P will too. Accordingly, instead of a program's speedup being unaffected, the program's speedup is adversely affected, as depicted in the bottom portion of Figure 11.

With respect to our solutions, for which $OH = \beta \times T_{change} + R \times T_{reload}$, if as p grows linearly with N, R grows linearly with N, the per-processor overhead may remain constant and speedup may be unaffected.

The impact that a solution to the TLB consistency problem has on performance changes when a program's speedup is viewed in this way. A solution with an OH that degrades performance when the problem size is fixed and N increases may not have a detrimental effect when p grows with N. This may be the case for our solutions. If the problem size is fixed, a $\Theta(N \log N)$ overhead reduces a program's speedup, as depicted in the bottom part of Figure 10. However, if the problem size grows with N and R grows linearly with N at the same rate as it does for a fixed size problem, then this overhead does not reduce the program's speedup and we have the situation illustrated in the top part of Figure 11. This is also the case for the solutions presented in this chapter. If the problem size is fixed and β remains constant as N increases, a program's speedup is adversely affected. If the problem size increases with N and β remains constant then the program's speedup is unaffected. Note, however, that it is unlikely that β will remain constant as the problem size increases.

In Figure 10 and Figure 11 situations are illustrated where both the net execution time and OH are both evenly distributed among the processors. In fact, neither may be evenly distributed.

Our main concern is that a solution performs well for large-scale systems and for a values of β and P or a value of R that is representative of such systems. Thus, in this and the next chapters we examine each solution's overhead and delineate the circumstances under which OH has only a negligible affect upon the performance of large-scale multiprocessors and those circumstances under which performance will suffer. This, at the very least, gives an idea as to whether or not a solution may perform well in a particular multiprocessor computing environment.

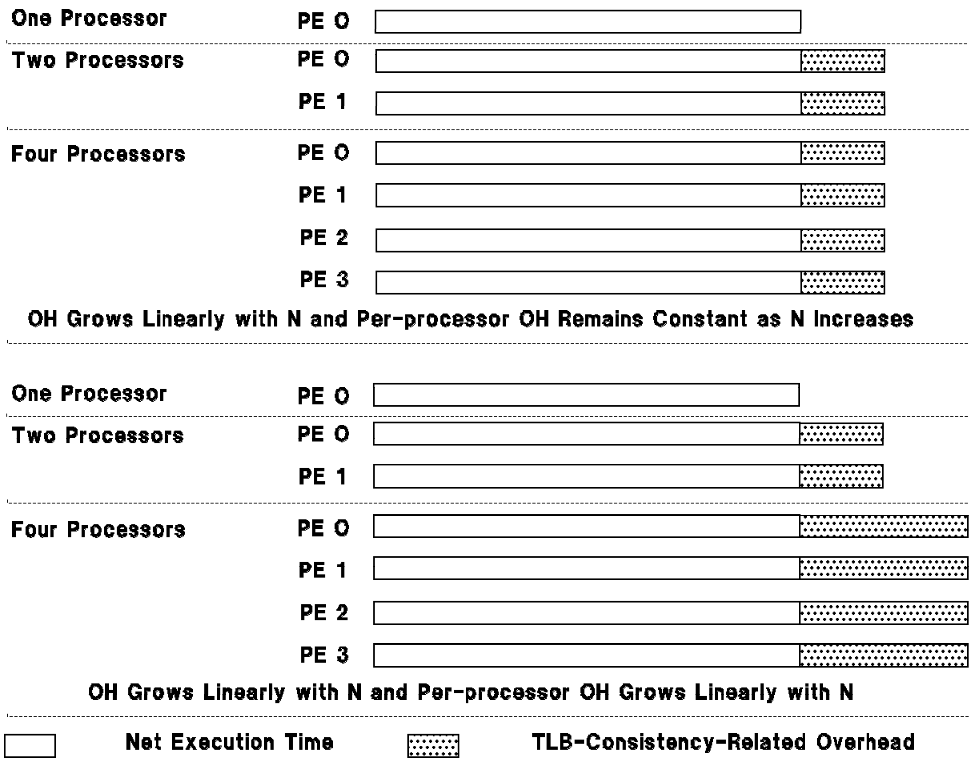


Figure 11. Execution Time vs. N with Problem Size Increasing with N

4.2. Solutions

For the three solutions that are described in this section in general

$$OH = \beta \times (T_{change} + P \times T_{participate}),$$

where β is the number of unsafe changes that occur, T_{change} represents the average time, both busy and idle, expended by a processor making an unsafe change, $T_{participate}$ represents the average time expended by each processor that also participates in the algorithm, and P is the average number of participating processors. Assuming a multiprocessor with a multi-stage interconnection network, T_{change} and $T_{participate}$ grow as log N. If we further assume that P grows linearly with N and $P \times T_{participate}$ dominates T_{change} for large N, OH is $\Theta(N \log N)$. In highly-parallel programs with large amounts of page sharing, it is likely that if P is not equal to N, it will at least grow linearly with N. As described in Section 4.1.2, the impact of these solutions depends on β and P and their growth with N.

These solutions, which all use an invalidation strategy, can be employed in either bus-based multiprocessors or multiprocessors with more general interconnection networks. TLB Shootdown, in its original form [Black, *et al.*, 1989] and as revised by Rosenburg

[1989], is a software solution to the TLB consistency problem that is described in both forms in Section 4.2.1. In its original form it requires essentially no hardware support. Although Rosenburg's revised TLB Shootdown offers better performance, it is at the cost of hardware support. Lazy Devaluation, the last solution described in this chapter, in Section 4.2.1, employs a detection strategy in addition to an invalidation strategy. Unlike the other two solutions, it does not solve the most general problem. In particular, it does not support multiple processes executing in the same address space or allow a process to modify the address space of another concurrently executing process.

After outlining the algorithm followed by each solution, we discuss the generality with which it solves the TLB consistency problem and the effect its computational overhead could have on system performance, especially for systems with a large number of processors. In addition, we comment upon the overhead contributed by the side effects of adopting each solution. The main attraction of the two versions of the TLB Shootdown algorithm is that they solve the most general TLB consistency problem, the original algorithm requiring essentially no hardware support. The main drawback of all three solutions presented in this chapter is the required synchronization and communication among processors that share a page table. For highly-parallel programs with large amounts of page sharing these solutions can incur high overheads that will impede system performance because all processors sharing a page table are interrupted to participate in the algorithms whether or note an unsafe change will affect them.

4.2.1. TLB Shootdown–Two Variations

Two software TLB consistency solutions that employ an invalidation strategy are described in this section. They are closely related since Rosenburg's algorithm [1989] is an improved version (performance-wise) of the solution proposed by Black, *et al.* [1989]. The algorithm of Black, *et al.* is included in Carnegie Mellon University's MACH operating system, which has been ported to multiprocessor architectures including the BBN Butterfly [Rettberg and Thomas, 1986], Encore's Multimax [Encore, 1987], IBM's RP3 [Pfister, *et al.*, August 1985], and Sequent's Balance and Symmetry systems [Sequent, 1986 and Lovett and Thakkar, 1988]. Rosenburg's algorithm is incorporated in the RP3 version of MACH. In both algorithms, a lock associated with each page table must be secured in order to modify a page-table entry or to modify the list of processors that may be using the page table.

4.2.1.1. TLB Shootdown

The algorithm of Black, *et al.* basically can be described by the following sequence of actions. (Some implementation details have been omitted below.)

1. A processor that wants to modify a page table disables interprocessor interrupts and then clears its active flag (such a flag is associated with each processor and indicates if the processor is actively using any page table). The processor then locks the page table, flushes entries in its TLB related to pages for which translation information is to be modified, enqueues a message, which describes the TLB actions to be performed, for

each processor that currently might be using the page table, and sends an interprocessor interrupt to each of these processors.

2. When a processor receives an interprocessor interrupt, it clears its active flag.
3. The modifying processor busy-waits until the active flags of all interrupted processors are clear and then modifies the page table. Finally, after releasing the page-table lock and setting its active flag, the modifying processor resumes execution.
4. After clearing its active flag, each interrupted processor busy-waits until none of the page tables it is using remain locked. (The processor does not know which page table is being modified.) Then, after executing the required TLB actions and setting its active flag, it resumes execution.

This algorithm idles all processors that may be using a page table while it is being modified. In addition, the modifying processor cannot make any modifications until all interrupted processors have cleared their active flags. This synchronization can be very costly for applications where many processes share pages. As Rosenburg [1989] explains, the level of synchronization exhibited by the original TLB Shootdown algorithm is necessitated by deadlock avoidance and architectural features of multiprocessors to which MACH is targeted. With respect to this, Rosenburg raises three particularly interesting points:

1. The fact that the interprocessor interrupts provided by DEC, Sequent, and Encore multiprocessors are assigned a low priority permits subtle deadlocks involving unrelated data structure locks possible. In addition, as a result of the low priority of interprocessor interrupts, when a processor masks out device or clock interrupts it also masks out interprocessor interrupts. As explained above, since the modifying processor and, thus, all other participating processors cannot resume execution until all participating processors acknowledge receipt of the generated interrupts, this can cause serious performance degradation.
2. In these multiprocessor architectures, page-table entries are rewritten in order to record page-usage information temporarily maintained in TLB entries. Thus, the level of synchronization exhibited by TLB Shootdown prevents a processor from overwriting a page-table entry that was just modified by another processor.
3. Since some commercial multiprocessors have a relatively small physical page size (*e.g.*, 512 bytes on a VAX), the MACH operating system groups pages into larger system pages in order to reduce the amount of space required for virtual memory data structures. Since the interface to the MACH virtual memory system deals only with system pages, the modification of the translation information for a system page requires the modification of the translation information for several adjacent page-table entries. Since such changes cannot be made atomically, the extra synchronization is required.

4.2.1.2. Modified TLB Shootdown

Rosenburg's version of this algorithm [1989], which we call *Modified TLB Shootdown*, uses features of IBM's RP3 and requires less synchronization. Instead of an active flag associated with each processor, the RP3 algorithm assumes that a pair of counters (requests and updates) stored in shared memory are associated with each processor. And, instead of a queue of TLB-related messages being associated with each processor, a single message indicating the range of virtual addresses for which translation information was modified is associated with the modified page tables and each specific invocation of the consistency algorithm. The following steps outline the basic algorithm. (Some implementation details have been omitted below.)

1. A processor that wants to modify a page table locks the page table. The processor then modifies the page table and attaches to it a one-word message that describes the TLB actions to be performed. At this point, the processor atomically increments the requests counter associated with every other processor using the page table via a fetch-and-add operation (saving the old values) and then sends an interrupt to each of these processors. The increment of a processor's requests counter serves to notify the processor of pending TLB actions to be performed.
2. The modifying processor invalidates entries in its own TLB that are associated with pages for which translation information was modified and then waits for the interrupted processors to invalidate the corresponding entries in their TLBs. This condition is recognized when the updates counter of each of the interrupted processors is greater than the saved value of its requests counter.
3. Upon being interrupted, a processor records the current value of its requests counter. After performing the pending TLB actions associated with the page tables it is using, the processor sets its updates counter to the recorded value of its requests counter and then resumes execution. When the value of a processor's requests counter exceeds the value that was saved by the modifying processor, then the modifying processor knows that this processor has completed the required TLB actions.
4. When all the interrupted processors have completed their TLB actions, the modifying processor removes the corresponding message from the page table, unlocks the page table, and then resumes execution.

The smaller synchronization overhead of Modified TLB Shootdown, in comparison to the original algorithm, is due to several features of the RP3 architecture, listed below, which are in contrast to the features of the DEC, Sequent, and Encore multiprocessors mentioned above.

1. The RP3 provides an interprocessor interrupt with a priority greater than that of any device, including the clock. (In fact, the interrupt priority of the clock is software settable.)
2. In the RP3 instead of recording page-usage information when a TLB entry is invalidated or is being replaced, a standard technique that manipulates access permission bits, explained later in Chapter 7, is used to capture this information. This, however, does not entirely solve the problem because the protection fault handler that modifies the page table in order to record page-usage information still runs the risk of overwriting an entry that was just modified. In addition, a modifying processor can overwrite page-usage information when it modifies translation information. Thus, neither the protection handler nor a modifying processor is allowed to rewrite a page-table entry. Instead,

selective atomic operations, such as fetch-and-ϕ operations [Gottlieb, Lubachevsky, and Rudolph, 1983; Gottlieb and Kruskal, 1981], must be used to modify page-table entries. The operation *fetch-and-*ϕ(M,v) atomically stores in the memory location with address M the result of performing the operation ϕ on the contents of M and the value of v, and returns the prior value stored at M. (Fetch-and-ϕ operations are explained in more detail in Section 5.1.) A permission bit can be cleared by using a fetch-and-or operation and a modifying processor can use either a fetch-and-and or a fetch-and-store operation to modify a page-table entry.

3. There is no grouping of hardware pages into larger system pages because the RP3 hardware page size is 16 Kbytes.

A rich set of atomic fetch-and-ϕ operations are provided by the RP3. As mentioned by Rosenburg, although fetch-and-ϕ operations are well suited to the tasks at hand, they are not strictly necessary. For example, other atomic operations, such as the System/370 *compare-and-swap* instruction, can be used. The System/370 compare-and-swap instruction takes three arguments: an address and two register operands. If the contents of the memory location specified by the address matches the value of the first operand, then the second operand is stored in the addressed memory location. Whether or not the swap was made, the current contents of the memory location are returned and condition codes indicating whether the store was accomplished are set. Typically, the following sequence is repeated until the store is accomplished: the contents of a memory location are read into a register, modified as desired, and then written to the memory location if and only if the contents of the memory location is still as was read.

Experiments on the RP3 show that the parallel processing time spent by the P interrupted processors can remain constant even as N, the number of processors in the system, grows. This is compared to an increasing parallel processing time associated with the P interrupted processors exhibited by the original TLB Shootdown algorithm. Note that what is being measured is the “parallel” processing time and that it is the number of processors that are involved in the algorithm that is growing, not the number of processors in the system. As will be shown, for either version of the algorithm, the computational overhead per unsafe change is $\Theta(P)$ and, thus, for constant β, OH grows with P. In a system with a multi-stage network interconnecting processors and memory, the computational overhead per unsafe change is $\Theta(P \log N)$. If P grows linearly with N, then this overhead is $\Theta(N \log N)$.

4.2.1.3. Comparison

Unlike the original TLB Shootdown algorithm, Modified TLB Shootdown allows a processor to make an unsafe change to a page table without waiting for processors to synchronize. In addition, even though page-table modifications are serialized in both versions of the algorithm, in the modified algorithm processors can use a shared page table while unsafe changes are being made. This is made possible by the fact that the page-table modifications are accomplished by atomic operations. Each page is mapped by a single page-table entry, which is stored in a single 32-bit word that can be modified atomically with a fetch-and-ϕ operation. In allowing active sharing of a page table during its modification, Modified TLB Shootdown increases the allowable parallelism and adopts different semantics than those of TLB Shootdown.

Modified TLB Shootdown guarantees that no processor will use stale translation information after the modifying processor completes its part of the algorithm but it does not guarantee that the processors using the table will see the changes made as an atomic transaction. In contrast, the original algorithm does not make the modification until all processors that may be using the table are idle and the algorithm ensures that only the updated translation information will be loaded into TLBs since these processors must invalidate stale TLB entries before resuming execution. In other words, the semantics of Rosenburg's algorithm are slightly different from the original TLB Shootdown algorithm, since at any given moment one processor can use a page's old translation while another processor uses a new one.

When an unsafe change is made, both algorithms interrupt all processors that may have inconsistent TLB entries in order to invalidate stale translation information. Although this policy leads to a fully general solution, it forces all processors that might have stale translation information to participate in the algorithm whether or not the unsafe change will affect them. In a large-scale system the performance of a highly-parallel application program with a large amount of page sharing among many concurrently executing processes may be handicapped by this policy. If β, the number of unsafe changes that occur during the execution of such a program, is not small enough, the interruption of the execution of a large number of processors may result in a severe impact on performance. In addition, in both algorithms only one process at a time can modify a specific page table. While the table is being modified no processors can be added to the list of processors using the table. (This is a restriction which Rosenburg [1990] believes can be removed from his algorithm.)

The computational overhead per unsafe change, of these algorithms is linear in P, the number of processors that participate in the algorithm. In the original TLB Shootdown the modifying processor enqueues P TLB-action messages, sends P interprocessor interrupts, and monitors the active flags of P processors. Modified TLB Shootdown requires the modifying processor to update P counters, send P interrupts, and monitor P updates counters. In highly-parallel programs with a large amount of page sharing it is likely that P grows linearly with N. If a system supports the broadcasting of messages then interrupting P processors takes $Q \times f(N)$ time, where $Q \times f(N)$ is $\Theta(1)$ for a bus-connected multiprocessor, $Q \times f(N)$ is $\Theta(Q \log N)$ for a multiprocessor with a multi-stage interconnection network, where $Q = 1$ for a system with broadcast capability and $Q = P$ otherwise.

In the original algorithm the modifying processor waits until each of the P processors has acknowledged its interrupt by clearing its active flag. Thus,

$$T_{change} = g(P,N) + Q \times f(N) + t_{slowest} \ ,$$

where $g(P,N)$ represents the modifying processor's time while actively engaged in the execution of the algorithm (*i.e.*, while locking the page table, flushing TLB entries, enqueueing messages, etc.), $Q \times f(N)$ is the modifying processor's time spent sending interprocessor interrupts, and $t_{slowest}$ represents the modifying processor's time inactively engaged in the algorithm, *i.e.*, while waiting for the slowest of the P processors to acknowledge its interrupt.

For Modified TLB Shootdown the equation for T_{change} generally looks the same, however, in comparing the experiments conducted by Rosenburg and Black, *et al.*, in general the time required for the slowest processor to complete its part of the algorithm is greater in the case of the original algorithm. Although the values of T_{change} differ for the two different versions of the algorithm, in either case for a system with a multi-stage network T_{change} grows with P and and N.

Modified TLB Shootdown does better than the original algorithm with respect to $T_{participate}$. It allows participating processors to act independently of one another and, thus, as shown by experiments on the RP3, the parallel processing time can remain constant even as P, the number of processors participating in the algorithm, increases. Rosenburg's experiments compare $T_{participate}$ for the original and modified algorithms and show that Modified TLB Shootdown is clearly superior to the original algorithm in an absolute sense. More important is the fact that the behavior of the algorithms diverges as P increases. In contrast to Modified TLB Shootdown, for the original TLB Shootdown $T_{participate}$ climbs fairly steadily after $P \approx 15$.

Even though the parallel processing time may remain constant as P grows towards N, as P grows more time is expended towards ensuring TLB consistency. For Modified TLB Shootdown $T_{participate}$ represents the average time expended by each processor in a system with N processors. For the original TLB Shootdown algorithm $T_{participate}$ depends on N in a different way. Since participating processors cannot resume execution until the page table is modified, $T_{participate}$ depends upon the time it takes for all participating processors to acknowledge their interrupts and clear their active flags. Thus, the slowest reacting of the P processors establishes the value of $T_{participate}$.

Even though Rosenburg's algorithm has better performance than the original TLB Shootdown algorithm, the main point is that in either case OH grows with P and β.

As demonstrated above, both algorithms have a complexity of $\Theta(P \log N)$. In addition, one of the side effects of adopting either of these algorithms is that a synchronization lock is placed on an entire page table rather than on a single page-table entry. This limits parallelism in maintaining TLB consistency in two ways:

1. only one process can modify a given page table at one time and
2. the scheduling of a process is delayed if a page table it may use is being modified.

Black, *et al.*, state that the overhead of the original TLB Shootdown algorithm could pose problems for large-scale systems because it scales linearly with the number of processors. Extrapolated measurements from an instrumented MACH kernel running on a 16-processor Encore Multimax indicate that machines with a few hundred processors might not experience a performance problem, except perhaps with regard to kernel space. To surmount this potential problem, Black, *et al.* suggest restructuring the operating system's use of memory so that TLB shootdowns are limited to groups of processors rather than all processors. Note, however, that parallel programs with the same degree of page sharing as exhibited by the operating system will encounter the same performance problems as the operating system. This was exhibited by Rosenburg's experiments which were discussed earlier.

In summary, both versions of TLB Shootdown have the following properties:

- *OH* per unsafe change grows with the number of processors that participate in in the algorithm and hence *OH* per unsafe change potentially grows with *N*.
- The execution of all processors that might have a TLB entry that will become inconsistent (or that is inconsistent, in the case of modified TLB Shootdown) as a result of an unsafe change are interrupted, whether or not they have used or will use the translation information in question. Thus, $T_{participate}$ includes processor time that potentially could have been used in the execution of the program.
- Parallelism is limited, since only one process at a time can modify a given page table at one time and processors must synchronize and communicate with one another.
- The scheduling of a process is delayed if a page table it may use is being modified.

TLB Shootdown ensures TLB consistency no matter how memory is shared, since processors cannot use translation information associated with a page table while the table is being modified. As mentioned above, Rosenburg's version has slightly different semantics and requires extra hardware support, but it is as effective and potentially has better performance. TLB Shootdown is a good solution to the TLB consistency problem for multiprocessors with no hardware support for this purpose. In addition, if unsafe changes rarely occur, *i.e.*, β is very small, and memory is not widely shared among processes, then either version of the algorithm should perform well. Otherwise, the overhead of these solutions is likely to adversely affect multiprocessor performance.

Notice that readers-writers semantics are implicit in both algorithms. Obviously, in the original TLB Shootdown algorithm, a processor cannot reload a TLB from a page table that is being modified and a modifying processor cannot effect an unsafe change until the execution of all processors using the targeted page table has been interrupted. Thus, read locks are implicit in this algorithm. In Modified TLB Shootdown processors can reload TLBs from a page table that is being modified. This is because Rosenburg assumes that a page-table entry can be modified with an atomic operation. The lock is implicitly encapsulated in the atomic operation. Rosenburg's [1990] states that if the RP3 employed hardware rather than software TLB reload, no modifications to his TLB shootdown algorithm would be necessary. If page-table entries could not be modified atomically (*e.g.*, if they spanned word boundaries), then some sort of locking would be required. Rosenburg [1990] believes that in this case individual page-table entries, rather than page tables, would be locked. Of course, a hardware TLB reload has to respect the lock, otherwise the algorithm for modifying page tables has to be altered.

4.2.2. *Lazy Devaluation*

Why not postpone invalidating TLB entries until absolutely necessary? Thompson, *et al.*, [1988] proposed this strategy, called *Lazy Devaluation*, to ensure TLB consistency when virtual memory is deallocated and when page protection is increased. Lazy devaluation has been implemented in a multiprocessor architecture based on the MIPS R2000 processor running the System V.3 UNIX operating system. Although this solution is effective for the targeted multiprocessor system, as observed by Black, *et al.* [1989], neither this UNIX implementation nor this TLB consistency solution supports multiple processes executing in the same address space or allows a process to modify the address space of another executing process.

The MIPS R2000 gives a TLB entry a six-bit field for an address-space identifier, so TLBs need not be flushed on context switches. Each process is assigned a TLBID, a TLB/processor identifier. To implement Lazy Devaluation, TLBIDs associated with active processes identify when TLB entries on a particular processor must be replaced or invalidated. A TLB reload also loads the TLBID of the executing process into the TLB entry's address-space identifier field. A TLB hit occurs when the referenced virtual address and the TLBID of the referencing process match the TLB entry's virtual address and TLBID.

When a process releases a region of virtual memory, a new TLBID is assigned to that process, making any TLB entries associated with the released space inaccessible to the process. When TLBIDs are recycled, all TLBs must be flushed. The eviction of a page is postponed until all related TLB entries are invalidated via a system-wide TLB flush. The remapping of a page is postponed until no TLB contains an entry for the page. If the remapping cannot be stalled, then all TLBs are flushed so that the change can occur. To handle increases in page protection, a data structure associated with each process or each TLBID tracks the processors that might have inconsistent TLB entries. When a process migrates to such a processor, those entries are flushed from its TLB.

In the case of virtual memory deallocation T_{change} is small and $T_{participate} = 0$. However, in general, for other unsafe changes Lazy Devaluation's overhead mirrors that of the TLB Shootdown algorithms presented in the Section 4.2.1, *i.e.*, OH grows as $P \log N$ and potentially as $N \log N$. The exceptions to this are when a remapping can be stalled until no TLB contains an entry for the page and when page protection is increased. In the former case if the stalling of the change does not affect performance then T_{change} is small and $T_{participate} = 0$. In the latter case $T_{participate}$ depends on the number of TLB entries that must be flushed.

When virtual memory is deallocated, modifying a process' TLBID makes associated TLB entries inaccessible to that process but not to other processes with the same address space. The procedure suggested for maintaining TLB consistency in the face of increases in protection is only effective for a modifying processor's TLB. It does not work if processes concurrently executing on separate processors share the same address space, although it can be used to implement the copy-on-write optimization. However, since this implementation of the 5.3 UNIX operating system supports neither parallel execution within the same address space nor operation on remote address spaces, this solution solves the problem of TLB consistency for this particular system.

If processes do not migrate frequently and unsafe changes rarely occur, then Lazy Devaluation could be efficient for systems similar to its target system. In fact, preliminary performance figures indicate that this approach succeeds for the target system without excessive TLB flushing. However, Lazy Devaluation could adversely affect multiprocessor performance if unsafe changes are not rare events. To improve performance, Thompson, *et al.* suggest maintaining extra information to allow selective invalidation of TLB entries rather than system-wide TLB flushes. In either case, since a TLB-Shootdown-like algorithm must be employed under certain circumstances, interprocessor communication and synchronization are required. If processes migrate frequently or pages are shared among many processes, as N grows more processors may have to participate in the algorithm, *i.e.*, the execution of more processors may have to be interrupted in order to maintain TLB consistency. In addition, the size of the TLBID field may affect the performance of this solution for large-scale multiprocessor systems since it restricts the number of processes that can be active at any one time. As mentioned by Thompson [1988], rather than managing the TLBID as a system-wide resource, the TLBIDs can be managed separately for each TLB. Finally, Lazy Devaluation does not support general memory-sharing patterns.

4.2.3. Summary

Of the three solutions presented in this chapter TLB Shootdown is the only one that can be implemented with essentially no hardware support. It requires that the architecture provide an interprocessor interrupt facility and architected TLB entry invalidation. Both versions of TLB Shootdown solve the most general TLB consistency problem. In contrast, Lazy Devaluation does not since it does not support parallel execution in the same address space nor remote address space modification.

Modified TLB Shootdown has better performance than the original TLB Shootdown algorithm but it does so at the cost of required hardware. This hardware includes a high-priority interprocessor interrupt facility, architected TLB entry invalidation, and hardware support for atomic operations. Page-usage information is updated via software. Even though Modified TLB Shootdown's performance is better, its computational overhead is of the same order as that of the original TLB Shootdown, *i.e.*, in general $OH = g(P,N) + Q \times f(N) + t_1 + P \times t_2(N)$, where $g(P,N) + Q \times f(N)$ represents the modifying processor's time while actively engaged in the algorithm's execution, t_1 is the modifying processor's time inactively engaged in the algorithm's execution, and $P \times t_2(N)$ is the participating processors' time engaged in the algorithm's execution. If β is not small then OH may adversely affect performance, the significance of this affect being dependent on the magnitude of β.

This holds true for Lazy Devaluation as well at least in the cases when it employs a TLB Shootdown-like algorithm to perform a system-wide TLB flush in the case of page eviction or unstallable remaps. Besides an interprocessor interrupt facility and architected TLB entry invalidation, Lazy Devaluation requires an extra TLB field to store a TLBID. The size of this field limits the number of processes that can be active at one time.

We believe that these characteristics of the three solutions presented in this chapter make one question the performance of these solutions in a large-scale multiprocessor system. If a general solution to the problem of TLB consistency is required, can a large-scale system gamble on the fact that β will be small? This is a good bet only if β is essentially independent of N. Assuming that the problem size increases with N, we believe that β grows with N, thus, we do not believe that these solutions will perform well in a large-scale multiprocessor system. Accordingly, in Chapter 5 we present other ways of solving the TLB consistency problem which have potentially better performance. When we compare our solutions to the solutions presented in this chapter, we assume that $g(P,N) + Q \times f(N)$ is comparable to the cost of making an unsafe change with respect to any one of our solutions and we assume $t_1(N) \equiv t_2(N)$. Thus, we estimate the overhead of solutions presented in this chapter to be $\beta \times ((P + 1) \times t_2(N))$, *i.e.*, the waiting time incurred by the modifying processor is considered to be equal to the average time expended by one of the participating processors.

Chapter 5. Solutions for Large-Scale Multiprocessors

If parallel programs targeted for large-scale systems exhibit large degrees of page sharing, the following properties of a TLB consistency solution can help to ensure that its computational overhead does not increase substantially with N:

- A processor's participation is independent of that of other processors.
- Necessary locks are placed on the smallest possible data entities.
- Serialization is not introduced.
- Extra communication is kept to a minimum.
- The amount of generated memory requests is such that it represents an insignificant impact on network traffic.

Unlike the solutions presented in Chapter 4, our solutions, presented in this chapter, have many of these characteristics. In particular, a processor's participation is independent of that of other processors and occurs only to make an unsafe change or to reload a TLB entry. The overhead of our solutions is comprised mainly of a small number of extra instructions and associated memory accesses that are executed on each TLB reload. Locks are associated with page-table entries rather than with page tables and parallelism with respect to both page-table modifications and consistency-ensuring TLB actions is provided.

Our three TLB consistency solutions are presented in Section 5.2. The data-locking mechanisms used by two of our solutions are first described in Section 5.2.1. Then Sections 5.2.2 and 5.2.3 present our two solutions for processor-based TLBs, Read-locked TLBs and Validation. Section 5.2.4 presents Memory-based TLBs, our solution that assumes TLBs located at memory, rather than at processors. Section 5.2.5 compares our solutions to those presented in Chapter 4 and Section 5.3 summarizes the main points discussed in Parts I and II of this thesis. Before we present our TLB consistency solutions, Section 5.1 defines the multiprocessor model that we assume.

5.1. Multiprocessor Model

Although the model presented in this section is not essential to our solutions, it facilitates their description by allowing us to discuss them in the same framework. In addition, it is the multiprocessor model that is used in our simulation study. We begin by presenting our architectural model in Section 5.1.1. Then in Sections 5.1.2 and 5.1.3 we describe the models that we assume for the physical and virtual memories, respectively. Finally, Section 5.1.4 contains a short discussion regarding page fault handling.

5.1.1. Architecture

Referring to Figure 1, a highly-parallel shared-memory multiprocessor is composed of many processing elements, typically hundreds, that access a global main memory via an interconnection network, which in this case is assumed to be a multi-stage network. Main memory is partitioned into memory modules, usually one per processor, that are grouped into one or more memory clusters. A cluster controller, which may be associated with each memory cluster, has access to all the memory modules in its cluster and may be used as an I/O controller. In some cases a cluster controller may be capable of executing code and communicating with all processors in the system.

Our model assumes that if a general-purpose cache is associated with each processor, then these caches are write-through caches that do not store translation information and their consistency is ensured by a separate mechanism from the one used to ensure TLB consistency. It is not clear what will be the most effective means of ensuring cache consistency in large-scale multiprocessors. We assume compiler-directed cache consistency and, since the compiler does not have any information regarding virtual-memory management, we assume that it is not safe to cache page-table entries. Thus, only page tables and TLBs store translation information. In addition, we assume that page tables are not paged. Two of our solutions, Read-locked TLBs and Validation, assume that a TLB is associated with each processor. Our other solution, Memory-based TLBs, assumes that a TLB is associated with each memory module.

Although the consistency of the general-purpose caches is beyond the scope of this thesis, we briefly address the cache management actions that may be required as a result of unsafe changes. These actions depend on whether the general-purpose caches are addressed by virtual or physical addresses. For example, if a virtual page is deallocated, the general-purpose caches must be flushed of entries associated with the page before the page is reallocated. Otherwise, when the page is reused, data from the deallocated page may be accessed erroneously. A page eviction causes the contents of a frame to change but not the contents of a page. Thus, if the caches are addressed by physical addresses, all the caches must be flushed of entries associated with a page before the page is evicted. This policy also may be advantageous in the case of virtual-address write-back caches since otherwise the page may be faulted back into memory just to be updated. The modification of the virtual-to-physical mapping of a page changes the contents of a frame, not the page. Thus, the entries of a virtual-address cache are not affected. On the other hand, since a frame is to be allocated to a different page, if the caches are addressed by physical addresses then entries for the page currently residing in the frame must be flushed before another page is mapped to the frame. Entries in virtual-address caches sometimes include protection bits. When this is the case, entries for a page must be updated or flushed when the page's protection is increased. Otherwise, access permission is verified using the protection bits stored in the TLB entry for the referenced page.

Our model provides fetch-and-ϕ operations [Rudolph, 1982; Gottlieb, Lubachevsky, and Rudolph, 1983; Gottlieb and Kruskal, 1981], atomic read-modify-write (RMW) operations, referred to individually as fetch-and-ϕ(M,v), where M is a memory location and v is some value. Such an operation returns the original contents of M to the executing processor and simultaneously stores at M the result of applying the function ϕ to the original contents of M and the value of v. For example, a *fetch-and-add*(M,v) operation increments the contents of M by v and returns the old contents of M; a load to memory location M can be performed as a fetch-and-add(M,0). If the previous value is returned when executing a store, a *fetch-and-store* operation is obtained, which swaps the old value in memory with the new value sent by a processor. The following fetch-and-ϕ operations are defined by various highly-parallel shared-memory multiprocessor architectures: fetch-and-add, fetch-and-and, fetch-and-or, fetch-and-min, fetch-and-max, fetch-and-store, fetch-and-store-if-nonnegative, and fetch-and-store-if-zero.

5.1.2. Physical Memory

The physical address space is divided into frames of equal size, each consisting of a range of contiguous memory addresses. A frame is fully contained in one memory cluster and is usually interleaved across all the memory modules in the cluster. (On the RP3, some frames, called local frames, are contained entirely within a single memory module.) Consecutive frames are interleaved across the memory clusters.

These interleavings map a physical address to a memory location: the frame number specifies a memory cluster and a unique frame within the cluster, and the displacement determines a particular location within that frame. Since all frames in a memory cluster are interleaved identically, the displacement (plus the number of the memory cluster) uniquely defines the memory module containing a specific location. This mapping is depicted in the lower portion of Figure 12; the upper portion of the figure illustrates the logical-to-physical address translation, which is described below.

5.1.3. Virtual Memory

In order to reduce the cost of context switching between processes, a global (i.e., system-wide) segmented virtual address space is desirable in a highly-parallel multiprocessor. With such an address space, a context switch requires neither a TLB flush nor the flushing of a large number of entries from the cache. The address space is composed of global segments. Each *global segment* refers to a unique virtual-memory object in the system and is the smallest portion of the address space that can be shared among processes. It is subdivided into pages, each the size of a frame, consisting of a range of contiguous virtual memory addresses. A *global virtual address* consists of a global segment number, page number, and displacement. Each process can access a subset of the global segments in the system. In this case two pages in different segments (and hence having different virtual addresses) may be mapped to the same frame in order to efficiently implement copy-on-write. When this aliasing is done, the pages are marked read-only and thus do not create cache consistency problems.

With a global virtual address space concurrent processes never use the same virtual address to refer to different objects. This implies that the global virtual address of a non-shared object, such as a private data area, cannot be known until run-time. To permit the

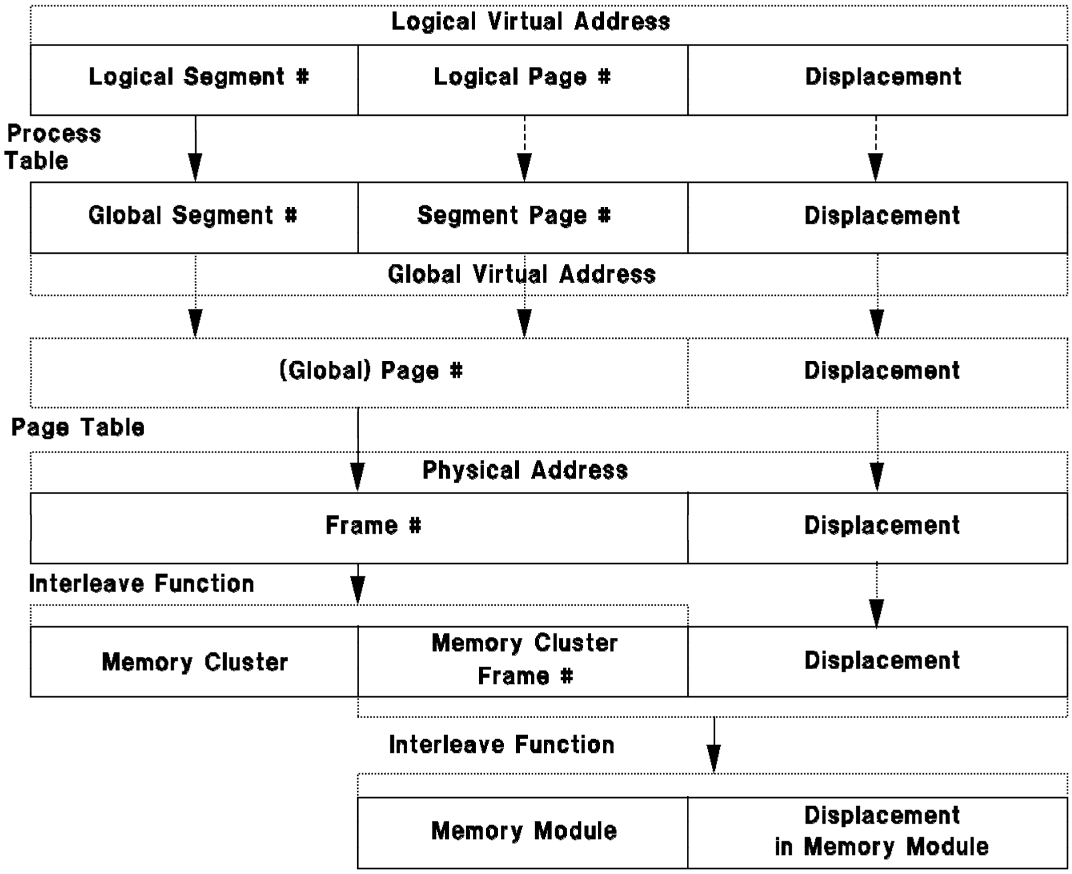


Figure 12. Virtual-to-Physical Address Translation

virtual addresses used by a process to be determined at link-time, we add a level of indirection that we call logical addresses. A *logical address* is composed of a logical segment number, page number, and displacement. The logical segment number specifies which of the global segments accessible by the process is to be used for a memory request. A processor maps a logical segment number into a global segment number using a per-process table that has an entry for each logical segment mapped by the process. That global segment number is used to construct the global virtual address of the memory location.

In the remainder of this thesis, we ignore the fact that virtual memory is divided into segments and consider a global page number to be the composite of a global segment number and a segment page number. We refer to the global page number simply as the page number, and treat a virtual address as being composed of a page number and a displacement within the page.

Pages can be relocated dynamically during program execution. A *page-frame table* is used by the operating system to keep track of available frames. It has the inverse function of the page table, identifying all the virtual pages mapped to each frame.

We define a *paging arena* as a region of memory to which the paging algorithm is applied independently. Each memory cluster can be made a separate paging arena by restricting which frames may be used to store a specific page so that the page number uniquely determines the memory cluster in which it can be stored. (With this restriction, each virtual address uniquely specifies the memory module in which it must reside.) Therefore, page replacement can be done locally within a memory cluster and the page table can be partitioned, with each cluster only storing entries for pages that can be contained in it. When main memory is comprised of more than one memory cluster, Memory-based TLBs requires that each cluster be a separate paging arena. Thus, a page, whenever resident, is always in the same memory cluster. Constraining the mapping of pages to memory modules in this way may not always result in more page-ins. It is similar to using a set-associative cache instead of a fully associative one. The size of the set is the number of frames per cluster, which will run into the thousands. If hashing is used to associate pages to memory clusters, the distribution of active pages over the clusters can be expected to be very even. However, in drawing this analogy we note the following differences:

- The time to satisfy a cache miss is much shorter than the time to satisfy a "page miss"; the system requires page fault rates much lower than cache miss rates.
- The shared memory acts as a "page cache" for all processors; it has to contain the resident sets of all running processes. Hence, the minimum acceptable associative set size is likely to grow with the number of processors.

5.1.4. Page Fault Handling

In this section we present a model of classical uniprocessor demand-paging implementations and contrast it to multiprocessor implementations. When a memory request refers to a nonresident page, a page fault is said to occur and the request cannot be satisfied until the page is made main-memory resident. If the page is being brought into memory, the request must wait until that page is loaded. If the page is still in memory, it can be reclaimed; otherwise, an I/O transfer from auxiliary memory may be required.

A frame must be allocated to store the faulted page. Often this requires a page to be evicted. If the page selected for eviction is dirty (has been written during its current memory residency), it must be written to disk before its frame can be made available. An optimization found in many operating systems uses free and dirty lists to reduce the amount of required I/O. In these systems, pages that are not in the resident set (the set of active pages) of any process are placed on either the free or dirty list depending on whether or not they have been modified while resident. A *page daemon* is used to monitor memory usage, remove pages from resident sets, and write dirty pages to disk. Then, a process encountering a page fault can simply obtain an available frame from the free list and need not be concerned with choosing a page to evict or handling dirty pages. (See Bach [1986] for a description of how this is done in the UNIX operating system.)

In this thesis and in our simulations we ignore this optimization and abstract page fault handling using a self-service rather than a server-based approach. When a process incurs a page fault and no other process is waiting for the page, it becomes the pager for that page and takes whatever actions are necessary to make the page memory resident. This may include evicting a resident page, writing a dirty page to disk, and reading the faulted page. If the faulted page is already assigned to a pager, the faulting process puts itself on a queue of processes awaiting the residency of that page. Before returning to user mode, a pager adds all processes that are waiting for its page to the ready queue. We are designing serialization-free algorithms for doing this using Wilson's serialization-free event processing techniques [Wilson, 1988].

In a uniprocessor locks required in page fault handling are implicit in the use of interrupt levels or the disabling of context switching. On a multiprocessor they must be made explicit and care must be taken to avoid improper operation of the paging system. For example, if two processes running simultaneously on different processors fault on the same page, only one may become the pager – the other must enqueue itself and wait. By using the fetch-and-ϕ operation and carefully ordering stores, it seems that much of the serialization otherwise required by these locks can be eliminated. The design of algorithms of this type is among our work-in-progress.

There is one additional aspect of multiprocessor page fault handling that merits discussion here. Consider a situation where multiple simultaneously active processes (multiple page daemons or pagers in a system without page daemons) are seeking a page to evict. Each must obtain a different page and, therefore, must skip pages that have been chosen for eviction by other processes. Suppose you have an optimal algorithm that chooses the best page to evict, analogous to Belady's [1966] optimal algorithm for uniprocessors. Using this algorithm, all these processes would simultaneously choose the same "best" page; one would get it and the others would retry. Then all but one would get the second best, and so on. Thus, the use of an optimal choice function for page eviction leads to serialization. This can be eliminated by using a suboptimal strategy that allocates a different page to each process. For example, the "Second Chance" algorithm of Belady [1966], an approximation to the least-recently-used (LRU) policy, can be modified using the fetch-and-add operation so that each concurrent evicter is given a unique page to evict. When there is more than one distinct paging arena, it may be best to have one or more page daemons per arena, each using an optimal eviction choice strategy.

Again, note that the model described above is presented so that the following discussion of our solutions to the TLB consistency problem can be presented in a common framework. Many details are not necessary. For example, with the exception of Memory-based TLBs, the equivalent of a global address space is not required.

5.2. Solutions

Unlike the general cache consistency problem, the TLB consistency problem is sufficiently specialized so that it admits both hardware and software solutions that may be efficient for large-scale systems. This is because of the following four reasons.

1. The primary factor is that the ratio of the number of modifications to the number of references for address translation information is very small in comparison to the same ratio for other kinds of data. A reference occurs for each virtual memory access, while a modification occurs only when a page's status or protection is changed.
2. Address translation information can be present anywhere on the path from a processor to memory.
3. When the memory mapping needs to be modified, one can often select which translation information to change, *e.g.*, there are few restrictions on which page to evict when a new page must be loaded into memory.
4. Optimistic recovery techniques [Strom and Yemini, 1985] permit address mapping information to be invalid if not used. Hence page-table modifications can be avoided some of the time.

Read-locked TLBs, discussed in Section 5.2.2, capitalizes on the third principle described above. It maintains a read-lock on a page-table entry during the entire time that an image of it resides in a TLBs, and restricts page-table modifications that can result in potentially harmful TLB inconsistencies. Validation, discussed in Section 5.2.3, exploits the first and fourth principles. It allows obsolete translation information to be cached but prevents its use to access memory. Finally, as is discussed in Section 5.2.4, Memory-based TLBs, capitalizing on the second principle, assumes that address translation is performed at the memory, where TLBs are interconnected by a bus that is used to enforce TLB consistency.

As was evidenced by the solutions presented in Chapters 3 and 4, either implicit or explicit readers-writers semantics are needed to implement a solution to the TLB consistency problem. Since two of our solutions make use of explicit readers-writers semantics, before presenting our solutions Section 5.2.1 discusses serialization-free methods of implementing readers-writers locks in a multiprocessor environment.

5.2.1. Data-locking Mechanisms

In this section we present two algorithms that employ the fetch-and-add operation and are used in the implementation of two of our TLB consistency techniques. These algorithms implement readers-writers semantics so that in the absence of writers no serialization of the readers occurs.

5.2.1.1. One-counter Algorithm

One way to implement the readers-writers algorithm is to maintain a counter that indicates the number of active readers and writers (*cf.*, [Gottlieb, Lubachevsky, and Rudolph, 1983]). This counter, C, is equal to $r - (nw)$, where n is no less than the maximum possible number of active readers in the system, and r and w are equal to the current number of active readers and writers, respectively. A prospective reader performs the following algorithm:

begin

```
    OK := false;
    repeat
       if C ≥ 0 then
          if fetch-and-add(C, 1) ≥ 0 then
             OK := true;
          else fetch-and-add(C, − 1);
    until OK;
    read the data;
    fetch-and-add(C, − 1);
end;
```

A prospective writer does the following:

```
begin
    OK := false;
    repeat
       if C = 0 then
          if fetch-and-add(C, − n) = 0
             OK := true;
          else fetch-and-add(C, + n);
    until OK;
    update the data;
    fetch-and-add(C, + n);
end;
```

One might think the first test in these algorithms is redundant, however, as explained by Gottlieb, Lubachevsky, and Rudolph [1983], it is necessary to prevent unacceptable race conditions. (Two readers-writers implementations are given in Gottlieb, Lubachevsky, and Rudolph [1983]. The first one treats readers and writers equally and, thus, easily can starve writers. The second one avoids starvation by giving writers priority over readers. For each, reader activity is fully parallel in the absence of writers. In contrast to these algorithms, Freudenthal and Gottlieb [1991] present, and prove the correctness of, an algorithm that requires only unit fetch-and-add addends, *i.e.*, it can be implemented with only fetch-and-increment and fetch-and-decrement operations.) Wilson [1988] proved a variant (*i.e.*, the algorithm can be implemented with only fetch-and-add operations, rather than fetch-and-increment and fetch-and-decrement operations).

5.2.1.2. Two-counter Algorithm

Instead of coordinating the actions of readers and writers via a mechanism based on mutual exclusion, the two-counter algorithm uses event counts [Reed and Kanodia, 1979], which indicate the sequencing of significant events that take place during an asynchronous computation. Using event counts, a reader is able to determine when it has received inconsistent data. Unlike the readers-writers algorithm, it does not prevent more than one active writer; we assume this protection is provided by another synchronization mechanism.

This technique maintains two counters, C_1 and C_2. A prospective reader executes the following algorithm:

begin
  repeat
    MyC1 := C_1;
    make a private copy of the data;
  until MyC1 == C sub 2;
  use the private copy of the data;
end;

A prospective writer does the following:

begin
  fetch-and-add(C_2, 1);
  update the data;
  fetch-and-add(C_1, 1);
end;

Because a writer first increments C_2 and finally increments C_1, and a reader first reads C_1 and finally reads C_2, a reader can always detect a writer that is in the process of modifying the data. (Here we assume that it is not possible for a counter to have been incremented past overflow and back to its original value between the time that a reader accesses the two counters.)

5.2.2. Read-locked TLBs

If a TLB entry never becomes invalid, then the problem of TLB consistency is avoided. This is the premise of an avoidance strategy. Such an idea has been incorporated in the Dynix operating system [Beck and Olien, 1987]. In this operating system TLB inconsistencies are prevented by [Beck, 1988 and 1990]:

- implicitly flushing TLBs on each context switch (the page-table base register is rewritten on a context switch and, thus, implicitly flushes the TLB),
- explicit TLB flushing at other times,
- system-wide TLB flushing at times, and
- not considering pages in the working set of any executing process as candidates for eviction.

In the same spirit, in order to avoid "shooting down" pages that are never used, the MACH operating system [Rashid, *et al.*, 1987] uses an avoidance strategy as an optimization. If a page is not mapped in by any process then TLBs cannot have inconsistent entries for the page and, therefore, when unsafe changes are made to the page-table entry for the page, it is not necessary to invoke the TLB Shootdown algorithm.

One of our solutions, *Read-locked TLBs* [Teller, Kenner, and Snir, 1988], also uses an avoidance strategy. Read-locked TLBs can be employed in a system with either processor- or memory-based TLBs. In this thesis, however, we describe it with respect to a system with processor-based TLBs. The problem is avoided by prohibiting an unsafe change while a valid copy of the translation information for the targeted page is resident in one or more TLBs, other than the one accessed by the processor desiring to make the unsafe change. In essence, a processor holds a read lock on a page-table entry from the time it loads the related translation information into one of its TLB entries until that TLB entry is invalidated or replaced. This makes the number of active read locks on a page-table entry equal to the number of TLBs that store a copy of its translation information. Since a write lock must be held by a processor while performing an unsafe change of a page-table entry, an unsafe change cannot be made unless no TLB, except that of the modifying processor, contains an entry for the related page. In this way the modification and subsequent use of translation information is serialized and translation information stored in TLBs and page tables is always consistent.

To implement this solution a counter, which is called the *replication counter* (RC), is associated with each resident page. A RC acts as a readers-writers lock on a page-table entry and indicates the number of TLBs that store the entry's translation information.[3] When a copy of a page's translation information is loaded from a page-table entry into a TLB, the page's RC is incremented. The RC is decremented when the TLB entry is invalidated or replaced. If a page's RC = 0, a processor that can set the RC to $-N$ obtains a write lock on the page's page-table entry and, thus, can make an unsafe change to the page's translation information. For pages that are to be permanently locked in memory (*e.g.*, system pages) a read lock that is never released can be obtained at boot time. To implement the required locks without unnecessary serialization we use the one-counter algorithm of Section 5.2.1.1.

Read-locked TLBs dictates that only those pages with RCs of zero are candidates for eviction. Virtual memory can be deallocated at any time. However, it must be ensured that a mapping for a deallocated page is not used after the virtual page has been reallocated. Similar to Lazy Devaluation, described in Section 4.2.2, this can be done by postponing the reuse of a virtual page until no TLB, except that of the processor reusing the page, contains an entry for it. In our model a segment, rather than a page, would be deallocated. Thus, the segment number could not be reused while any TLB contained an entry for a page in the segment. This can be ensured by associating a counter (distinct from the counter used by the operating system to denote the number or processes sharing a segment) with each global segment number; the counter is incremented on every associated TLB reload and decremented on each invalidation or replacement of an associated TLB entry. The other two

---

3 In addition to representing the read-write lock state of a page-table entry, the RC can be used to indicate whether or not a page is resident. A fetch-and-store-if-zero operation can then be used to atomically test whether a page may be evicted and note that a processor is evicting it, if permitted.

unsafe changes, remapping a page and increasing the protection of a page, can also be supported in this way. That is, the unsafe change is postponed until no other TLB, except that of the modifying processor, contains an entry for the page. This idea of postponing unsafe changes has also been adopted by Tzou, Anderson, and Graham [1987]. The goal of their research is to minimize the overhead of virtual memory remapping while addressing the TLB consistency problem. Their research proposes an efficient mechanism for moving data between virtual address spaces in shared-memory multiprocessors that integrates interprocess communication, virtual memory, and process scheduling mechanisms.

In some cases Read-locked TLBs is ineffective for all unsafe changes with the exception of page eviction. These unsafe changes may have to be postponed for long periods of time and, in some cases, indefinitely. For example, consider a set of processes that are cooperating in the execution of a program. Let the number of processes in this set be greater than one and less than or equal to N, the number of processors in the system. In addition, assume that all these processes are processor resident and are executing concurrently. What if one of these processes wants to remap a frame being shared among the other processes? If a mapping to the frame is resident in the TLBs of all the processors executing these processes and during the normal course of execution the mapping remains in the TLBs, the change cannot be made. Similar situations can occur if interprocessor communication is implemented using a copy-on-write mechanism, which is done in the MACH operating system [Rashid, *et al.*, 1987] or by remapping a page from the sender's space to the receiver's space, which is done in the DASH operating system [Anderson, Tzou, and Graham, 1988]. The receiver may have the translation for the targeted page stored in its TLB and during its normal course of execution it does not replace that entry. Or, as is explained a little later in this section, copy-on-write on process creation may not be feasible if the parent process migrated from another processor, leaving the translation information for the targeted page(s) in another TLB. Thus, in some cases postponement may not be feasible and in other cases postponement just may not work. If an unsafe change cannot be postponed or it seems likely that it will be postponed indefinitely, an alternate method, like TLB Shootdown, described in Section 4.2.1, must be employed.

Read-locked TLBs, although not a fully general solution by itself, may be a viable solution for some large-scale multiprocessor systems if it is coupled with another solution that can be employed as deemed necessary. It was originally designed for a system like the NYU Ultracomputer [Gottlieb, 1987] prototype, which is based on Motorola 68010 microprocessors [Motorola, 1985], running Symunix I [Edler, 1988], where process address spaces are considered to be logically private. The Ultracomputer prototype running a new version of the operating system, called Symunix II [Edler, Lipkis, and Schonberg, 1988a and 1988b], sometimes would have to forego the copy-on-write optimization unless it invoked an alternate TLB consistency solution. In the new version of the operating system, address spaces are still logically private to processes, but processes can share individual pages in fairly general ways.

Read-locked TLBs is implemented by enhancing the generic TLB-reload routine with instructions that implement the reader's portion of the one-counter algorithm. If an atomic update operation is supported by the targeted architecture, then the overhead associated with a TLB reload is minimized. Otherwise, explicit locking is required to update counters. Thus, Read-locked TLBs' overhead, which is referred to as OH_{Lock}, does not depend on β, the number of unsafe changes that occur during program execution. Instead, OH_{Lock}, primarily depends on R_{Lock}, the number of TLB reloads that occur during program execution, *i.e.*,

$$OH_{Lock} = R \times T_{reload},$$

where R equals R_{Lock}. and T_{reload} is either the average processor time expended to execute the TLB-reload routine associated with Read-locked TLBs (which is assumed in our simulations) or the average processor time expended to execute the instructions unique to Read-locked TLBs' reload code (which should be used when comparing Read-locked TLBs with other solutions, such as those presented in Chapter 4). Read-locked TLBs' reload routine includes the updating of the RCs associated with the page related to the TLB entry being replaced and the page whose translation information is being loaded. The TLB-reload routine that we use for Read-locked TLBs in our simulations is described in Section 7.5.3.5.

As we will show, for all our solutions OH is defined in terms of the number of TLB reloads, rather than unsafe changes, that occur during program execution. This cost includes the instructions that comprise the TLB-reload routine and the memory accesses generated by them. The processor time expended while waiting for the generated memory references to be satisfied depends on the depth of the network, which we assume is $\log N$. As explained in Chapter 7, our simulation model accounts for both network transit time and network waiting time. Thus, OH as it is defined can be used to compare our solutions to the TLB consistency problem.

If R grows with N then OH grows as $R \log N$. As N increases, if R remains constant, OH is $\Theta(\log N)$ and, thus, remains relatively constant. As we show in Chapter 8, for a class of parallel programs R_{Lock} grows linearly with N and, thus, OH_{Lock} has an asymptotic growth of $O(N \log N)$. It is not clear whether the number of unsafe changes has an affect on R. If this solution provides an effective page replacement policy then OH_{Lock} can be subsumed by the bookkeeping required to implement the page replacement policy. (To keep overhead low, the TLBs of idle processors must be flushed periodically.)

In evaluating the performance of Read-locked TLBs, two components of the side effects of adopting it must be addressed. First, Read-locked TLBs may cause an increase in the number of page-ins and/or page faults and, thus, increase execution time. This may occur as a result of pages being locked in memory until no TLB contains entries for them. Second, since Read-locked TLBs locks in memory all pages that have an entry in a TLB, a deadlock can occur if there is not a page available for eviction. It can be guaranteed that such a deadlock situation does not occur if the total number of page frames is greater than the total number of TLB entries plus the number of frames that can be allocated to nonresident pages at any one time. If each memory cluster defines a paging arena then for each cluster such a deadlock situation must be avoided in a similar manner. In this case the total number of frames per cluster should be greater than the total number of TLB entries plus the number of frames in the cluster that can be allocated to nonresident pages at any one time. Alternatively, we can drop the guarantee of there always being a page to evict and if deadlock occurs, then a recovery procedure can be employed. For example, if deadlocks are sufficiently rare, a mechanism like TLB Shootdown can be employed to evict a page that has its translation information stored in one or more TLB entries.

As demonstrated for Read-locked TLBs, as N increases, a balance must be maintained among the page size, the TLB size, and the size and configuration of physical memory. If such a balanced relationship exists then it is likely that the ratio of the total number of page frames to the total number of TLB entries is large in a large-scale multiprocessor system. Thus, it seems unlikely that the page-in or page fault rates is increased by this solution. On the other hand, if these rates decrease as a result of adopting this solution, the improvement in performance should be taken into account when comparing Read-locked TLBs with other candidate solutions for such an architecture. But recall that Read-locked TLBs must be considered only a partial solution to the TLB consistency problem. In addition, it should be considered as an adjunct solution to the TLB consistency problem for large-scale multiprocessors that require a fully general solution.

On process creation, an inherited page may be unnecessarily copied. The copy-on-write optimization can reduce the number of times pages must be copied on process creation: Before making the page accessible to the children, the parent makes an unsafe change to a page-table entry, changing the access permission of the page from read/write to copy-on-write. The copy-on-write page is shared by the parent and child processes as a read-only page until one of the processes attempts to modify it. At that time, the contents of the page are copied to a new page in the address space of the modifying process. Hence, copying is avoided unless it is absolutely necessary. With the adoption of Read-locked TLBs, the copy-on-write optimization can be utilized by a processor on process creation only when no other TLB, but its own, contains an entry for the page. Therefore, the overhead associated with Read-locked TLBs also includes the execution time associated with the unnecessary copying of pages resulting from having to forego the copy-on-write optimization.

Even if this time-saving optimization must frequently be foregone, the results of Nelson and Ousterhout [1988] suggest that in multiprocessors with virtual-address caches the use of copy-on-write for process creation may only yield a moderate performance gain over *copy-on-fork*, where all pages being inherited by a process from its parent are copied on process creation. This is because a large percentage of shared copy-on-write pages are modified. When a copy-on-write page is modified and, thus, copied, it incurs an extra page fault. These extra page faults are avoided by employing copy-on-fork, instead of copy-on-write. In addition, in multiprocessors, like SPUR [Hill, *et al.*, 1986], where general-purpose virtual-address caches store protection bits along with the data in individual cache lines, when a process *forks* (creates a new process), cache entries for all the pages to be inherited by its children must be flushed in order to mark them read-only. Therefore, in some multiprocessors the increase of Read-locked TLBs' overhead that corresponds to foregone uses of the copy-on-write optimization may be negligible. Note, however, that copy-on-write performs very well when a fork is followed by an *exec* (which overlays the image of a program onto the running process).

In summary, although Read-locked TLBs is only a partial solution to the TLB consistency problem, it may solve the problem in some multiprocessor systems. Read-locked TLBs, itself, requires neither special hardware support nor processor communication and/or synchronization, and processors participate in the algorithm in an independent manner. As a result, in a large-scale multiprocessor, the algorithm exhibits a large degree of parallelism. $OH = R \times T_{reload}$, where R is the number of TLB reloads that occur during program execution and T_{reload} is the processor time expended to execute Read-locked TLBs' reload routine. The side effects introduced by adopting Read-locked TLBs include delays introduced by postponing unsafe changes until it is safe to effect them, potential increases in the number of page-ins and page faults, and unnecessary copying of inherited pages on process creation. Since Read-locked TLBs is only a partial solution to the TLB consistency problem, in other systems it will have to be coupled with a solution like TLB Shootdown. In this case, the overhead contributed by the other solution must, of course, be factored in.

5.2.3. Validation

Detection strategies allow stale entries to reside in TLBs but detect an obsolete entry upon its use to access memory. *Validation*, a fully general solution to the TLB consistency problem, which was initially proposed by Teller, Kenner, and Snir [1988], uses this strategy. Similar to the techniques adopted by Emberson [1987] and Smith [1985] to ensure cache consistency, Validation associates a generation count (GC) with each frame. The value of a GC serves as a time stamp that is used to identify stale TLB entries. As shown in Figure 13, a validation table is associated with each memory cluster and a copy of a cluster's table is stored at each of the cluster's memory modules. A validation table has an entry for each frame located in the cluster (we assume that each frame is interleaved among the cluster's memory modules). The entry for a frame identifies the latest GC assigned to the frame. Memory requests from the processors are augmented with the GC stored in the TLB (accessed on a TLB reload) and the translation is atomically validated at the targeted memory module. The process of validation and the memory access, itself, can occur concurrently or the additional operations per memory request must be pipelined so that the memory service rate is not decreased. A memory request is satisfied only if the translation is valid, otherwise a message is sent to the processor to refresh the stale translation information stored in its TLB. When an unsafe change is imminent, the modifying processor multicasts a message to the memory modules in the memory cluster where the targeted page is resident. The receipt of the message initiates the updating of a generation count associated with the referenced frame and, thus, maintains consistency among the validation tables of the cluster.

Again we use readers-writers semantics. In contrast to the readers-writers locks employed in Read-locked TLBs, in Validation the locks are held only momentarily, being used to ensure that a page's translation information and its frame's GC are updated atomically. Assuming for the moment that a read-write lock is associated with each page-table entry, a TLB reload is accomplished by obtaining a read lock on a page-table entry, copying the translation information from it, reading the GC associated with the frame to which the page is mapped from the validation table of the memory cluster to which the page is mapped, and

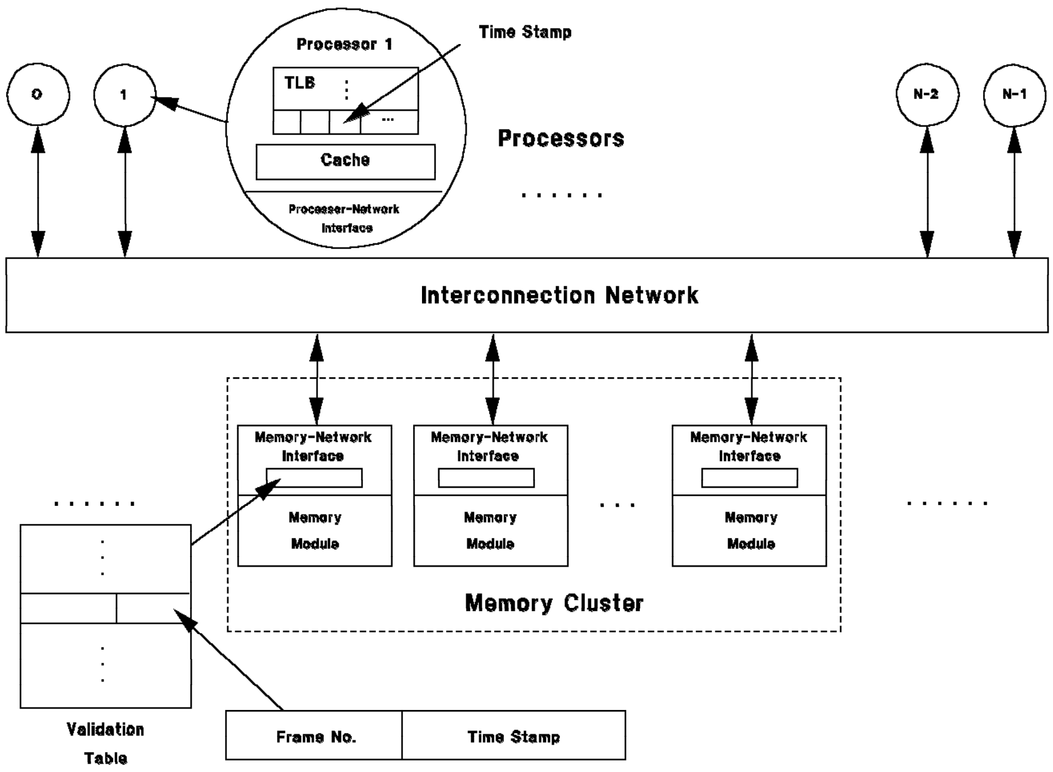


Figure 13. A Multiprocessor with Hardware Support for Validation

releasing the read lock. To make an unsafe change a processor must obtain a write lock on a page-table entry. Before releasing the lock, it updates the GC in all the validation tables of the cluster to which the related page is mapped. In this way, no reference to the page can be satisfied until the referencing processor has a current image of the translation information stored in its TLB. This makes Validation effective no matter how memory is shared. Thus, it is a fully general solution to the TLB consistency problem.

Although a read-write lock could be associated with each page, it is more efficient to associate it with each frame. Less storage space is required because there are generally fewer frames than pages. Also, in the presence of aliases there may be more than one page-table entry per frame; if a read-write lock is associated with each page, a processor would have to obtain multiple write locks in order to modify the mapping of a frame. If a read-write lock is associated with each frame, rather than with each page-table entry, a TLB reload is accomplished by first consulting the page-table entry for a frame number and then obtaining a read lock on that frame. Then the frame number is read again from the page-table entry, this time with the read lock in force, and is compared with the previously read frame number. If they differ, *i.e.*, the page-to-frame mapping has changed in the meantime, then the read lock is released and the process is repeated. Otherwise, the GC is read from an appropriate validation table and the read lock is released. Note that in reading the page-table entry, the translation information was accessed and, thus, the TLB reload is complete once the information is loaded into the TLB and the valid bit is set. Write locks are obtained similarly. This method of obtaining locks on a frame, rather than pages, can also be used to implement Read-locked TLBs.

Using the two-counter algorithm of Section 5.2.1.2, we can simplify the above procedure and make the locks implicit. This is done by associating two GCs with each frame: the GC in the validation tables and the GC in the related page-frame table entry (in this case, we assume that each page-frame table entry has a field to store the GC assigned to each frame). The GC in the validation tables, corresponding to C_2 in the algorithm, is used, as before, by the memory module to validate incoming memory requests. However, a TLB reload copies the GC from the page-frame table, corresponding to C_1, instead of an appropriate validation table. The GCs of the page-frame table and all the validation tables of the frame's memory cluster agree except while translation information is being modified. A processor that desires to make an unsafe change executes the writer portion of the two-counter algorithm by first updating the GCs in the validation tables (C_2), then modifying the page-table entry, and finally updating the GC in the page-frame table (C_1). A TLB reload obtains a trial frame number from the page-table entry, the GC from the page-frame table entry (C_1), and then the actual frame number is read again from the page table. This process is repeated until the trial and actual frame numbers agree; at that time the GC and frame number are loaded into the TLB. The GC sent with the memory request is validated by comparing it to the GC stored in the validation table of the targeted memory module (C_2). This completes the reader portion of the two-counter algorithm.

The modifying processor's participation in Validation is independent of the other processors. Since all solutions to the TLB consistency problem in multiprocessors with multi-stage interconnection networks must employ some type of locking on page-table entries in order to effect unsafe changes, the time expended by the modifying processor in executing that part of the writers portion of the two-counter algorithm that updates the GC in validation tables and the GC in a page-frame table is considered to be comparable to the time expended in any other algorithm.

Processors other than the modifying processor participate in the algorithm on which Validation is based when they use a stale TLB entry and when they perform a TLB reload. A stale entry is detected when a memory request is tested for validity at the targeted memory module. Stale translation information is updated by performing a TLB reload. Thus,

$$OH_{Val} = S \times (T_{reload} + t_{stale}) + (R - S) \times T_{reload},$$

where S is the number of stale TLB entries used by the N processors during program execution, T_{reload} is the processor time expended in an N-processor system to execute the TLB-reload routine associated with Validation, a sample of which is given in Section 7.5.3.4, t_{stale} is the time for one round trip through the network interconnecting N processors with memory that is required when stale translation information is used to access memory, and

R is the number of TLB reloads that occur during program execution. Again, note that T_{reload} is defined differently depending on whether we are making comparisons among our own solutions or among our own and other solutions. It is the processor time expended to execute the TLB-reload routine in the former case and the processor time expended to execute the instructions unique to the TLB reload code in the latter case. Since we believe that S will be small, as illustrated in the simulation results presented in Chapter 8, we feel that OH_{Val} is fairly represented by $R \times T_{reload}$.

If the number of TLB reloads, R, remains fairly constant as N increases, OH_{Val} is $\Theta(\log N)$ and, thus, remains relatively constant as N increases. If R grows linearly with N, then OH_{Val} grows asymptotically as $N \log N$.

There are several side effects of adopting Validation that should be considered. If the cluster size remains constant as N increases then T_{change} remains relatively constant as N increases. The multicast of the GC to the memory modules of a cluster has a relatively constant cost and the latency increases as *log N* increases. If the cluster size increases with N then more network contention may result, possibly increasing T_{change} and the performance of this solution. If bus monitors are associated with each validation table and the tables of a cluster are interconnected via a bus, then whether or not the cluster size increases with N the cost with respect to network transit remains relatively the same. Rather than multicasting a message to all the memory modules of a cluster, when a processor makes an unsafe change one message is sent to one memory module. This memory module then forwards the message to the other memory modules in its cluster via the cluster bus. This organization mirrors that of the snoopy TLBs used to implement Memory-based TLBs, discussed in Section 5.2.4, and, thus may exhibit some of the same side effects described therein.

As discussed in Chapter 4, Rosenburg [1989] addresses a difference in semantics between his algorithm and that of Black, *et al.*, [1989]. A similar issue arises with respect to Validation. The synchronization employed by an algorithm determines the relative times at which translation information is modified and related TLB entries become consistent with the system page tables. When an unsafe change and the invalidation of related inconsistent TLB entries are tightly synchronized, *e.g.*, by suspending the execution of processors or the serialization of global events inherent in a bus-based architecture, after the page-table modification has been made, all processors use the updated translation information. On the other hand, when an unsafe change can occur concurrently with the use of potentially stale copies of the translation information, it is possible for stale TLB entries to be used after the related translation information has been modified, *i.e.*, a time interval may result in which multiple processors referencing the same page use different page-to-frame mappings—one processor uses the old version of translation information while another uses the new version. After a certain point in time, however, all processors use the same mapping for a page. An example of these semantics used by Rosenburg occurs when a process executing in parallel with other processes within the same address space deallocates two pages of its address space. In this case, once the process modifying the page table completes its part of the algorithm, no process will be able to access the deallocated pages. However, while the algorithm is in progress, another process may find one of the pages invalid and the other page accessible. In addition, one process may continue to access a page after another process has found it to be invalid. In this case the first process is running on a processor that has not yet received the related interrupt while the other process is running on a processor that has received the interrupt. Rosenburg suggests that operating systems semantics should make such behavior explicitly undefined. The semantics of this algorithm are similar to those of Rosenburg's algorithm. After the GC for a page/frame pair is updated at the validation tables, the associated page cannot be accessed. References being made concurrently with the broadcast of the updated GC may or may not be able to access memory, depending upon when they reach memory with respect to the broadcast.

If the size of the generation count tag increases with N then unless the path from the processors to memory is large enough to accommodate the larger tag, the number of packets in a message will increase. According to Kruskal, Snir, and Weiss [1986], larger messages may dramatically increase network delays and thus increase execution time.

To implement Validation care must be taken when updating the GC of a frame. When a GC is incremented in order to change its value, a problem arises when the GC wraps around: There may be an inconsistent TLB entry associated with the newly updated GC value and this inconsistent TLB entry will appear to be consistent! This problem can be solved in several ways.

1. When wrap-around occurs use a TLB Shootdown-like algorithm to invalidate relevant TLB entries.
2. Bound the time interval during which a TLB entry is guaranteed to be safe, with respect to GC wrap around, and invalidate the entry before this time interval expires.
3. Maintain a set of counters for each frame, one for each allowable GC value. Each counter has N bits, where N is the number of processors in the system. A TLB reload increments the counter of the frame corresponding to the GC stored in the TLB entry. An invalidation or replacement of a TLB entry decrements the corresponding counter. A frame's GC is updated by assigning it a value associated with a zero counter. If there are N counters per frame, there is always a zero counter because the modifying processor can make one zero. If small GCs are used, however, and if each allowable value of the GC of a frame is present in at least one TLB (which is unlikely since stale TLB entries are likely to be replaced during the execution of a program), then a TLB Shootdown-like algorithm must be employed in order to create a zero counter.

Validation exhibits many of the ideal characteristics of the solutions for bus-based architectures. The participation of a processor is completely independent of that of other processors. Its implementation does not require interprocessor communication or synchronization. And, the computational overhead per unsafe change is small and relatively constant. The participation of a processor can occur in parallel with that of other processors. And, as many as N TLB updates to maintain TLB consistency can be performed concurrently, *i.e.* Validation affords unbounded parallelism in maintaining TLB consistency. The side effect of potentially increasing network latency is the only significant drawback of this solution. Thus, if the bandwidth of the interconnection network is large enough to accommodate the transmission of a GC with each memory request, then Validation is a fully general solution to the problem of TLB consistency that may prove to be efficient in large-scale multiprocessor architectures. Note that if a write-back, physical-address, general-purpose cache is associated with each processor, then coordination is needed between the caches and TLBs to ensure that data write-backs are targeted to the correct frame.

5.2.4. Memory-based TLBs

Using ideas inherent in the solutions described in Chapter 3, Memory-based TLBs is a solution proposed for large-scale multiprocessors, where the processors and memory modules are interconnected by a multi-stage network. As depicted in Figure 14, rather than locate the TLBs at the processors, Teller, Kenner, and Snir [1988] suggest that a TLB, designed similar to a snoopy cache, be associated with each memory module.

Since the TLB is a special-purpose virtual-address cache, virtual rather than physical addresses are transmitted through the network. This requires each memory cluster to be a distinct paging arena. A page, when resident, is assumed to be stored in a specific cluster of memory modules, the cluster being defined by the virtual page number and the memory module being defined by the referenced virtual address. In addition, if a general-purpose cache is associated with each processor then these caches must be addressed by virtual addresses. With address translation at the memory, an additional delay may be experienced. However, using a technique due to Sun Microsystems, Inc., which is referred to in Thakkar and Knowles [1986], memory access can be overlapped with virtual-to-physical address translation performed by the TLB. This technique, taking advantage of the fact that dynamic random access memories (DRAMs) are accessed by first selecting a row with the low-order bits of an address and then selecting a column with the high-order bits of the address, presents the displacement defined by the referenced virtual address as the DRAM's row address while performing the page-to-frame translation. After a successful translation is performed the DRAM is presented with the frame number as the column address. The translation information for the referenced page is accessed from the TLB, which is implemented as a snoopy cache that stores copies of portions of page-table entries and can be designed as proposed in Goodman [1987]. As depicted in Figure 14, within each memory cluster the snoopy TLBs and the cluster page table are interconnected by a bus. Bus management, similar to a snoopy-cache protocol, ensures consistency among the TLBs and the page table of a cluster. A cluster's page table is accessible only via the cluster's bus, making all unsafe changes visible to the snoopy-TLBs and serializing page-table modifications. Atomicity of address translation and memory access guarantee that a TLB will not respond to a bus request while a memory access is in progress. This solution may be a viable candidate for a large-scale multiprocessor system, although it does not solve the TLB consistency problem in its fullest generality when multiple paging arenas are assumed. This is because if multiple paging arenas are assumed then pages that map to the same frame are restricted to have page numbers that map to the same memory cluster.

The semantics of this solution differ from those of the schemes described in Chapter 3 since the interconnection between processors and memories is no longer a bus that serializes transactions. Now only transactions that involve page-table entries are tightly synchronized and only within a cluster. All TLBs in a cluster are updated simultaneously, but

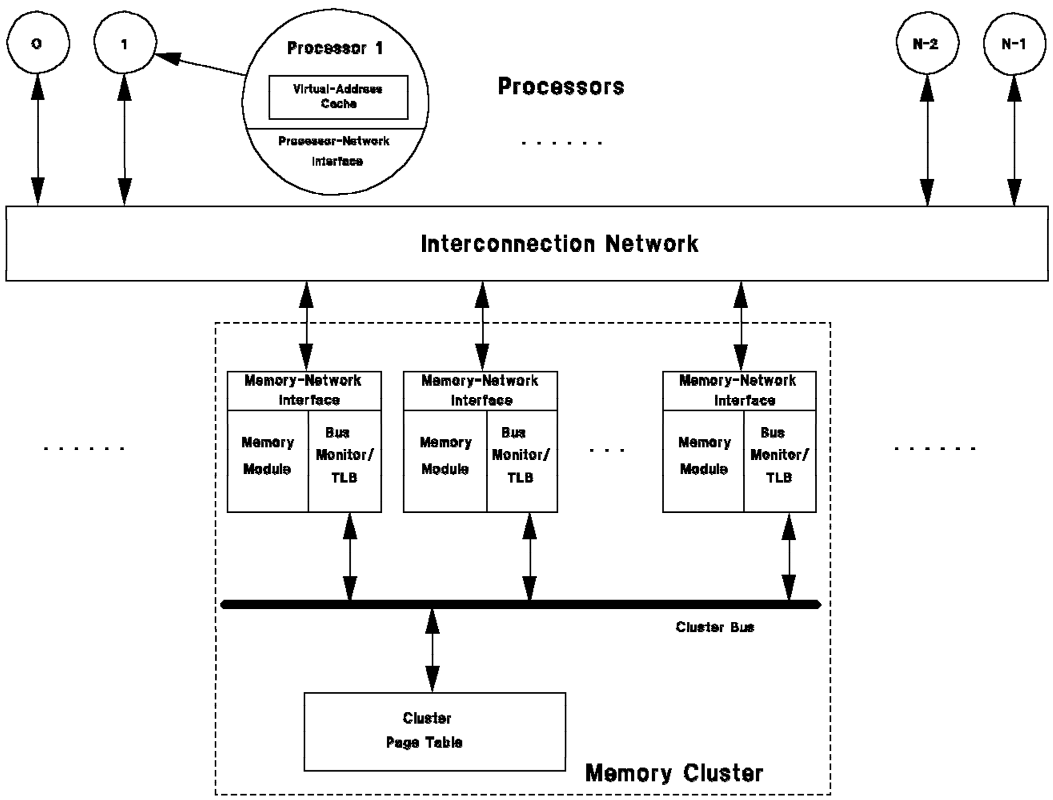


Figure 14. A Multiprocessor with Hardware Support for Memory-based TLBs

while they are being updated, accesses to resident pages can be taking place. Remapping of virtual or physical memory is transparent to the processor but logic associated with the TLBs at memory must alert the processor to other unsafe changes that have been made and of which it is not aware. Upon accessing memory, the processor must be informed if the referenced virtual page is not resident, if the access permission for the referenced physical page has been reduced, or if the referenced virtual address has been deallocated. In response, the processor then executes a corresponding trap routine.

Since multiple, possibly independent, processes may be accessing the same TLB, an address space identifier must be included as part of the address presented to a TLB or a single global address space must be assumed. In addition, virtual pages that map to the same physical frame must have some common identifying feature, such as common address bits, that identify the memory cluster to which they are mapped. Thus, the flexibility of shared memory is compromised by these assumptions since a page is always mapped to the same cluster and, therefore, pages that map to the same frame are restricted to be pages that map to the same memory cluster. Because of this restriction Memory-based TLBs does not solve the problem in its fullest generality.

The computational overhead of Memory-based TLBs appears to be small, like that of the solutions described in Chapter 3. A processor effects an unsafe change independent of other processors and the readers-writers semantics that ensure that unsafe changes are effected correctly are encapsulated in the protocols associated with cluster buses. Thus, as with our other solutions, Memory-based TLBs' overhead, OH_{Mem}, is represented by the cost of TLB reloads. In addition, in our simulations OH_{Mem} includes delays incurred by assuming blocking, rather than lockup-free TLBs. Accordingly,

$$OH_{Mem} = R_1 \times T_{reload_1} + R_2 \times T_{reload_2} + T_{\text{busy-TLB}} + T_{\text{busy-bus}} + T_{pgexcpt},$$

where R_1 and T_{reload_1} are the number of TLB reloads satisfied by TLBs in a cluster and the contention-free cost of each, R_2 and T_{reload_2} are the number of TLB reloads satisfied by a cluster's page-table memory and the contention-free cost of each, $T_{\text{busy-TLB}}$ is the time that memory references are delayed while waiting for TLBs that are busy servicing TLB misses, $T_{\text{busy-bus}}$ is the time that TLB-miss servicing is delayed as a result of a busy cluster bus, and $T_{pgexcpt}$ is the network transit time related to informing processors about exceptions caused by memory references that they generated. The total number of reloads is referred to as $R_{Mem} = R_1 + R_2$ and the total contention-free reload time is equal to $T_{reload_1} + T_{reload_2}$.

The values of $T_{\text{busy-TLB}}$ and $T_{\text{busy-bus}}$, which occur while delayed in the targeted cluster's TLB subsystem, may vary with the cluster size and depend on the design of the TLB subsystem. Even though we assume that on a TLB hit translation is overlapped with memory access, if a TLB cannot be accessed, processors can be delayed. In our simulations the design of the TLB is rather basic, in the sense that it is a blocking cache, rather than a lockup-free cache [Kroft, 1981], *i.e.*, it does not provide buffering for TLB misses that are being serviced. Without buffering a TLB is considered busy until an outstanding TLB miss is satisfied. Thus, a TLB miss blocks the memory-network interface from receiving a memory request from the network until the miss is satisfied. Note, however, that if TLB misses are buffered then the subsystem must be sophisticated enough to recognize misses on a page that are subsequent to a miss that is buffered for servicing. When designing the simulator, it was our belief that buffering would not be necessary since the cost of a TLB reload in this case is very small (5 or 7 cycles depending on whether the translation information is accessed from another TLB or from the memory that stores the page table) and accesses of shared variables would not be so very frequent. (For more details see Section 7.5.3.6.) Our simulation results, described in Chapter 8, however, indicate that buffering is necessary.

At first glance, it appears that OH_{Mem} is small in comparison to OH_{Val} and OH_{Lock}. However, as mentioned above, other factors may affect the contributions of $T_{\text{busy-TLB}}$ and $T_{\text{busy-bus}}$ and, thus, affect the magnitude of OH_{Mem}, for example:

- Since the TLBs are interconnected by a bus, the number of memory modules per cluster is limited. However, due to the infrequency of page-table modifications, in comparison to the total number of memory references, the available bus bandwidth limits the number of snoopy TLBs that can be interconnected in this way to a much larger number than the number of processors that can be interconnected in a bus-based multiprocessor architecture. If the number of memory modules per cluster increases with N, the number of processors, then the number of TLBs in the cluster grows. At some point the intra-cluster rate at which unsafe changes and/or TLB misses occur may rise, thus causing an increase in cluster bus traffic. An increase in cluster bus traffic may result in a rise in the cost of a TLB reload and, accordingly, memory requests may take longer to satisfy and the effective pipeline of requests from the processors to the memory may be delayed. This increase in network latency can increase execution time.

- As N increases, a larger virtual address may be needed. If the path between the processors and memory is limited then the size of the virtual address may affect m, the average number of packets comprising a message. According to Kruskal, Rudolph, and Snir [1986], larger messages may dramatically increase delays in all but very lightly loaded networks. For a fixed traffic intensity, the average waiting time increases linearly in m, and the variance increases quadratically in m.

- Assuming a constant size memory cluster, an increase in N results in an increase in the number of memory clusters and a possible change in the mapping of virtual pages to memory clusters. Since this mapping determines the set of pages that compete for residency in a cluster, a change in the mapping may have an adverse affect on the page fault rate and, thus, increase processor execution time.

- Processor-based TLBs are generally accessed by one process at a time. Memory-based TLBs are accessed by concurrently executing processes, as are processor-based TLBs in multiprocessor architectures that attempt to mask communication latencies and synchronization delays by associating multiple processes with each processor so that the execution of these processes can be multiplexed (for example, see [Agarwal, *et al.*, 1990 and Smith, 1978 and 1989]). Thus, these TLBs are subject to TLB-entry thrashing. In addition, in order to access an entire page, its translation information must be replicated at each TLB in it's cluster. As a result of this and TLB-entry thrashing a memory-based TLB may have to be larger than a processor-based TLB, which is accessed by one process for a substantially longer interval of time.

In order to abstractly compare the hit rates of memory-based TLBs with processor-based TLBs, assume a fixed-sized TLB and consider the following argument. One would expect that cooperating processes concurrently executing on distinct processors would reference a common set of pages. Assuming that the number of processors equals the number of memory modules, let a be the average number of active pages per processor (*i.e.*, the average size of a processor's working set), s the average number of processors sharing a page, and c the cluster size. Then, the average number of active pages per memory module is equal to $\frac{\hat{a}}{s}$. Thus, if the average amount of page sharing is larger than the cluster size, TLBs located at the memory modules may enjoy better hit rates than those located at the processors. If a processor's TLB must be invalidated when a new process becomes processor resident then this also favors locating TLBs at the memory modules. In addition, with an adequate level of page sharing, TLB miss rates should be smaller if TLBs are located at memory rather than at the processors. Our simulation study shows that this may be the case for systems with small numbers of processors and for systems with multiple paging arenas.

OH_{Mem} appears to be small since it does not require network transit. Of course, it remains to be seen whether it is or not. Our simulation results, presented in Chapter 8, indicate that OH_{Mem} with one paging arena is smaller than OH_{Val} for reasonably small numbers of processors but as N increases, it becomes larger. The magnitude of OH_{Mem} is due in part to the fact that the simulated TLB subsystems do not provide buffering for TLB misses and that larger TLBs or multiple paging arenas are required at memory in order for Memory-based TLBs to be competitive with or outperform Validation for larger N. When comparing Memory-based TLBs with solutions that do not have the major portion of their cost in T_{reload}, it is necessary to include the cost of TLB reloads in OH of the other solutions. Regardless, Memory-based TLBs affords a large degree of parallelism in maintaining TLB consistency. If the side effects of adopting Memory-based TLBs is not significant then Memory-based TLBs may be a viable solution for large-scale multiprocessor systems. Recall, however, that Memory-based TLBs is a partial solution for systems with multiple paging arenas because pages that share a frame must map to the same memory cluster/paging arena. If general-purpose virtual-address caches at the processors contain protection information then coordination between cache and TLB management is necessary when a page's protection is increased. Without protection information in the caches, illegal writes will not be recognized until they are written back to memory.

5.2.5. Comparison

In order to easily compare the solutions described in Chapters 4 and 5 we include Table 1 and Table 2 which, for each solution, list its:

1. strategy,
2. required hardware,
3. memory sharing limitations, and
4. performance overhead.

We assume that broadcasts and multicasts from processors to memory and from memory to processors are provided and, thus, we do not account for the cost of broadcasting or multicasting.

Three of the solutions are fully general: TLB Shootdown, Modified TLB Shootdown, and Validation. Memory-based TLBs with one paging arena is also a fully general solution. Of these only the original TLB Shootdown algorithm provides a fully general solution to the TLB consistency problem that is essentially hardware independent and, thus, can be implemented on all existing multiprocessor architectures. However, this solution, as well as modified TLB Shootdown and Lazy Devaluation, can incur high overheads in large-scale systems since the algorithms on which they are based are of $O(P)$, and potentially $O(N)$, complexity. As mentioned in Rosenburg [1989], the original TLB Shootdown algorithm is acceptable for machines where the number of processors is in the tens. Rosenburg's algorithm, simpler and less synchronous, may be appropriate for machines with a few hundreds of processors but is not suitable for large-scale multiprocessor architectures, where a page is shared among many processes. This is also true for Lazy Devaluation since it reverts to a TLB-Shootdown-like algorithm at times. All three of these solutions allow only limited parallelism in maintaining TLB consistency.

The major part of the computational overhead of our solutions to the TLB consistency problem is incurred during the course of reloading TLBs, while the overhead of the other solutions, which were presented in Chapter 4, is incurred during the course of making unsafe changes. In the case of our solutions, the cost of making the unsafe change itself is rather inconsequential. And, as illustrated in Chapter 7, there is very little that is unique for the different TLB-reload routines associated with our solutions.

Since all solutions to the TLB consistency problem in shared-memory multiprocessors with multi-stage interconnection networks must use either implicit or explicit locking to effect unsafe changes, for our solutions we do not consider the cost of making an unsafe change and for TLB Shootdown algorithms we consider only that part of T_{change} that is related to the modifying processor waiting until the participating processors have completed their parts of the algorithm. Thus, in our simulation study, described in Chapters 6, 7, and 8, we do not simulate the cost of unsafe changes. The major cost of our solutions is embedded in the cost of TLB reloads; a few extra instructions are added to a generic TLB-reload routine in order to implement our solutions. The TLB-reload routine for Read-locked TLBs modifies two counters. Validation's reload routine explicitly locks a page-table entry and modifies a counter. And, Memory-based TLBs incurs delay time due to the TLB subsystem; delays may be incurred by memory requests that will have TLB hits.

Validation has many of the characteristics of the solutions for bus-based architectures and it is a fully general solution that permits unbounded parallelism in maintaining TLB consistency. Thus, it may prove to be an effective solution for large-scale multiprocessor systems. The only drawback to Validation is that a tag must be transmitted with each memory request. If the interconnection network does not provide adequate bandwidth, network latency, and thus Validation's overhead, may increase as a result.

Like Validation, Memory-based TLBs and Read-locked TLBs address the TLB consistency problem with respect to large-scale shared-memory multiprocessors. However, Read-locked TLBs is not always effective by itself and Memory-based TLBs requires the equivalent of a global address space and, if more than one paging arena is assumed, then it restricts memory sharing by not permitting pages in two different clusters to be mapped to the same frame. These solutions do not solve the problem in its fullest generality, but, like Validation, they have many of the characteristics of the solutions of Chapter 3 and they afford parallelism in maintaining TLB consistency. All three solutions do not require interprocessor communication or processor synchronization. But all three generate additional network traffic, albeit a small amount. Validation does so to update GCs and as a result of using a stale TLB entry. Memory-based TLBs does so as a result of referencing pages that have been evicted or deallocated or that have had their protection increased. Read-locked TLBs does so to update counters on each TLB reload, invalidation, and replacement.

Similar to Validation, Memory-based TLBs may increase network latency if the memory bandwidth is not sufficient to transmit the virtual, rather than the physical, address as part of a memory request. Although not fully general, this solution offers an interesting alternative of locating TLBs at the memory and ensuring their consistency via a snoopy-cache technique, while providing shared-memory functionality that is not severely restrictive. TLBs may need to be larger to accommodate multiple localities of reference and still provide a good TLB hit rate, but this cost may be worthwhile since this solution requires very little network traffic since TLB reloads are executed exclusively at the memory.

To be effective in most multiprocessor systems Read-locked TLBs must be coupled with another solution. If the cost of maintaining a counter for each resident page can be subsumed by the implementation of the page replacement policy then it seems reasonable to consider this solution as an optimization that could be incorporated in any other solution. Along the same lines, other techniques can be combined. For example, techniques used in the TLB Shootdown algorithms and in Lazy Devaluation might improve the performance of other solutions. An optimization used in TLB Shootdown avoids invoking the TLB consistency algorithm if no processor has the related page mapped in. If, as in the MIPS R2000, the TLB provides a field for an address space identifier, then some invalidations could be postponed as was shown in Lazy Devaluation. Rosenburg's improved version of TLB Shootdown demonstrates that in some cases synchronization overhead can be reduced and explicit locking of data structures eliminated by the use of fetch-and-ϕ operations and the encoding of data structures into one word, so that they can be atomically modified.

If we just compare $T_{participate}$ to T_{reload} then the TLB Shootdown algorithms do not look good in comparison to our solutions. However, the entire story can only be told if we know the behavior of both β, P, and R as N increases. Assuming a particular multiprocessor architecture and operating system, it is important to remember that the performance of our solutions can be improved by assuming a larger TLB size, which will lower the value of R, whereas the performance of the other solutions does, in fact, depend on β, the value of which cannot be altered.

5.3. Summary

Parts I and II of this thesis have discussed the TLB consistency problem and qualitatively compared solutions to this problem that have appeared in the literature. Each solution demonstrates an effective technique for solving the problem for a set of multiprocessor systems. This set can be defined by:

- the interconnection between the processors and memory,
- the scalability of the architecture, and
- the function of shared memory.

Special attention was given to the generality with which a solution solves the problem and the performance of a solution in a large-scale multiprocessor environment.

Good methods of ensuring the consistency of the general-purpose caches of bus-based systems have been proposed, and many have been implemented. In these systems, the bus interconnecting processors and memory allows efficient broadcasting and, thus, facilitates the solution of the consistency problems associated with general-purpose caches and TLBs.

The general problem of cache consistency is much more difficult to solve in large-scale multiprocessor systems where the processors and memory modules are interconnected by a multi-stage network. For these systems cache consistency may be ensured by directory-based schemes or by software-assisted cache management. The efficiency of a directory-based scheme for cache consistency is being explored ([Agarwal, *et al.*, 1988; James, *et al.*, 1990; Lenoski, *et al.*, 1990]). It it proves to be efficient, then it may be suitable for ensuring TLB consistency. If software-assisted cache management is employed, then it is certainly true that the TLB consistency problem must be solved via a different mechanism.

TLB consistency solutions have been proposed in the literature. A solution for a particular architecture must support the defined shared-memory functionality and should not degrade the achievable performance of application programs executed on the machine. Thus, for large-scale multiprocessor architectures that are intended to run highly-parallel application programs exhibiting large degrees of memory sharing among processes, the solution's cost should remain relatively constant as the number of processors increases. The TLB Shootdown algorithms and Lazy Devaluation seem suitable for medium-scale multiprocessors, with perhaps a few hundred processors. In large-scale systems, they may be ineffective due to the interruption of processor execution, potentially high synchronization costs, and limited inherent parallelism.

The following issues confront us:

1. None of the proposed solutions have been implemented on large-scale systems.
2. It is not known how the number of unsafe modifications or the number of TLBs that will be affected by these modifications change with *N*.
3. It is not known how much parallelism in consistency-ensuring TLB management is required.

We believe that the number of invocations of the TLB Shootdown algorithm and the number of processors that will participate in the algorithm will increase with *N*. Therefore, inherent interruption of processor execution, interprocessor synchronization and communication, and limited parallelism in maintaining TLB consistency are likely to have adverse effects upon performance. Solutions that do not incur these types of overheads may be better suited for large-scale systems. Read-locked TLBs, Memory-based TLBs, and Validation are examples of such solutions. Validation is the only one of these solutions that solves the problem in its fullest generality. Memory-based TLBs does so when one paging arena is assumed but even when this is not the case, the function of shared memory is not restricted significantly. These two solutions, however, may increase network latency if the network does not afford sufficient bandwidth. If the path between a processor and a memory module is sufficiently wide then these solutions may prove to be effective ways of ensuring TLB consistency at a cost that remains relatively constant as the number of processors increases. Note that future interconnection networks may provide much greater bandwidth between a processor and a memory module than is needed.

Since it is clear that there is a need for TLB consistency solutions with overheads that do not increase with *N*, we have examined our solutions more closely by simulating their performance in different size multiprocessor architectures. The simulation system we built for this purpose is described in Chapter 6. The experiments we ran are described in Chapter 7. And, the results of these experiments are discussed in Chapter 8.

Solution (Strategy)	**Required Hardware**	**Memory Sharing Limitations**	**Performance Overhead**
TLB Shootdown (Invalidate)	Interprocessor interrupt and architected TLB entry invalidation.	None.	$OH \approx \beta \times (g(P,N) + (P+1) \times t_{slowest})$, where β is the number of unsafe changes, $g(P,N)$ is the modifying processor's time actively engaged in the algorithm, and $(P+1) \times t_{slowest}$ represents the average time that participating processors are engaged in the algorithm and the modifying processor's time inactively engaged. Additional potential overheads: serialization of unsafe changes to page tables a page table's modification and use.
Modified TLB Shootdown Invalidate)	High-priority interprocessor interrupt, architected TLB entry invalidation, software-implemented page-usage information update, and hardware support for atomic operations.	None.	$OH \approx \beta \times (g(P,N) + t_{slowest} + P \times t_{average})$, where β is the number of unsafe changes, $g(P,N)$ is the modifying processor's time actively engaged in the algorithm, $t_{slowest}$ is the modifying processor's time inactively engaged in the algorithm, and $P \times t_{average}$ represents the average time that participating processors are engaged in the algorithm. Additional potential overheads: serialization of unsafe changes to page tables.
Lazy Devaluation (Delay & Invalidate)	Extra TLB field, interprocessor interrupt, and architected TLB entry invalidation.	No parallel execution in same address space or remote address space modification.	Stalled remaps and memory deallocation: inconsequential; protection increases: depends on the # TLB entries flushed; otherwise, TLB Shootdown-like overhead.

Table 1. Summary of Other Solutions for Large-scale Systems

Solution (Strategy)	**Required Hardware**	**Memory Sharing Limitations**	**Performance Overhead**
Read-locked TLBs (Avoid & Postpone)	Interprocessor interrupt and architected TLB entry invalidation.	Not always effective for unsafe changes other than eviction. Copy-on-write sometimes forfeited.	$OH_{Lock} = R \times T_{reload} = O(R \log N)$, where R is the number of TLB reloads and T_{reload} is the average TLB-reload cost. Additional potential overheads: possible increase in page-ins and page faults, and delays due to postponed unsafe changes.
Validation (Detect & Correct)	Validation tables, extra TLB field, comparators, and sufficient network bandwidth.	None.	$OH_{Val} = S \times (T_{reload} + t_{stale}) + (R - S) \times T_{reload}$, where S is the number of stale TLB entries used, T_{reload} is the average TLB-reload cost, t_{stale} is the extra cost incurred by stale entry use (one network round trip per use), and R is the number of TLB reloads. Additional potential overheads: generation-count transmission with each memory request and generation-count wrap-around solution.
Memory-based TLBs (Invalidate)	Bus-connected TLBs w/bus monitors, virtual-address caches, sufficient network bandwidth.	Address-space identifier or single, global address space. If multiple memory clusters, pages in different clusters cannot map to same page.	$OH_{Mem} = R_1 \times T_{reload_1} + R_2 \times T_{reload_2} + T_{\text{busy-TLB}} + T_{\text{busy-bus}} + T_{pgexcpt}$, where R_1 and T_{reload_1} are the number of reloads satisfied by TLBs in a cluster and the contention-free cost of each, R_2 and T_{reload_2} are the number of reloads satisfied by a cluster's page-table memory and the contention-free cost of each, $T_{\text{busy-TLB}}$ is the time that memory references are delayed by busy TLBs, $T_{\text{busy-bus}}$ is the time that TLB-miss servicing is delayed by busy buses, and $T_{pgexcpt}$ is the network transit time related to informing processors about exceptions.

Table 2. Summary of Our Solutions for Large-scale Systems

Part III: Performance Evaluation via Simulations

This part of the thesis includes:

- an overview of our simulation system,
- a description of our trace compression techniques,
- a description of our simulation model,
- a classification of the software built to implement the simulation system,
- a definition of the simulation input, and
- a discussion of the results of our simulation study.

Chapter 6. Simulation Methodology

This chapter describes the simulation methodology that we used to design and implement a trace-driven simulator for evaluating the performance of TLB consistency solutions in scalable multiprocessor architectures. In Chapter 5, we merely conjectured about the performance of our solutions in such systems. Since such qualitative arguments are not conclusive, using this simulator we have obtained data that allow us to quantitatively compare their performance in scalable multiprocessor architectures. The parameters and the program address traces used in our simulations are discussed in Chapter 7. And, the results of these simulations are presented in Chapter 8.

Simulation studies are carried out by first building a simulation model of a system, and then driving the model with external stimuli. Rather than drive the simulation model with a random number generator, a *trace-driven simulation* drives the simulation model with a program address trace. A *program address trace* is a trace of the sequence of virtual addresses generated by one or more programs. Alan Jay Smith believes that trace-driven simulation is a good method to study the performance of caches. Smith [June 1985] states that:

> it [trace-driven simulation] is superior to either pure mathematical models or random number driven simulation because there do not currently exist any generally accepted or believable models for those characteristics of program behavior that determine cache performance; thus it is not possible to specify a realistic model nor to drive a simulator with a good representation of a program. A trace properly represents at least one real program, and in certain respects can be expected to drive the simulator correctly.

As explained by Smith, there are other good reasons for using trace-driven simulation. It takes much less time to build a simulator than to build a prototype design. And, a simulated design can be varied, often by changing the value of an input parameter. Consequently, the majority of research directed towards the study of cache design and performance uses trace-driven simulation as a measurement tool. Accordingly, since TLBs are special-purpose caches, we believe that trace-driven simulation is an appropriate way to study multiprocessor performance of TLB consistency solutions.

The results of trace-driven simulation are, however, subject to criticism. As discussed in more detail in Chapter 8, our trace-driven simulations, and others like them, have several drawbacks, for example:

- A trace-driven simulation of ten million memory addresses represents about $\frac{2}{3}$ second for a 10 MIPS microprocessor, and much less than that for a multiprocessor composed of many such processors.
- Operating system instructions are not traced.
- The instruction set and the compiler of the machine on which the program was traced are not necessarily the same as for the machine being simulated.

Despite these drawbacks, the results of trace-driven simulations of this class have proven to be effective performance predictors. For example, see [Alexander, *et al.*, 1986] and [Smith, June 1985]. Accordingly, we believe that our trace-driven simulations provide data that predict the relative performance of the three solutions to the TLB consistency problem proposed in Chapter 5.

This chapter is divided into four sections. The first section gives a conceptual overview of our simulation system, including an explanation of how the address traces used in this thesis were generated. Section 6.2 describes the compression techniques that were applied to these traces. These compression techniques significantly reduced the sizes of the traces, and it is our belief that this was done without compromising the accuracy of our simulations. Next, a high-level specification of our simulation model is presented in Section 6.3. And, finally, Section 6.4 catalogs and briefly describes the software we built to implement the simulation system.

Some of the concepts introduced in this chapter are clarified in Chapter 7, where we define the input parameters used in our simulations and follow through the computation of the various service times used in our simulation model. As will be seen, our strategy is quite flexible, allowing us to simulate different models of computation, various architectures, and uniprogramming and multiprogramming environments, but without the overhead of unwieldy amounts of trace data.

6.1. Simulation System

Given input parameters that define a multiprocessor system and a uniprocessor trace of the execution of a parallel program, a simulation generates data that allow us to compare the performance of our TLB consistency solutions in the defined system with respect to four criteria:

1. number of TLB reloads,
2. processor time attributable to TLB management,
3. number of page-ins, and
4. affect on overall program execution time.

In addition, data are generated that can be used to study the performance of demand paging and paging I/O, and to characterize parallel programs.

In order to generate this data, instead of executing the program, a simulation models the processing of the instructions and memory references that comprise tasks and the management of the caches, TLBs, physical and virtual memories, and paging I/O subsystem in the defined multiprocessor system, which is assumed to have a demand-paged virtual memory. Unsafe changes are generated when pages are evicted from physical memory. The frequency with which unsafe changes occur is varied by changing the physical memory size.

During a simulation, we introduce service times for instruction execution, cache and memory accesses, cache and TLB misses, page faults, page-ins, and context switches due to page faults. An architectural change may alter service times and, thus, alter the scheduling of tasks to processors. Thus, a multi-stream virtual-address trace would be of limited use in driving such a simulation model because the scheduling of tasks to processors is fixed.

Instead, our simulation model is driven by a trace of an instrumented parallel program run on a uniprocessor. Such a trace, although statically defined, contains sufficient information to allow many different multi-stream virtual-address traces to be dynamically created from it. In this way, one trace can be used to simulate different multiprocessor architectures.

As shown in Figure 15, and as described in Section 6.1, our simulation system is divided into three parts, the

1. preprocessor,
2. scheduler, and
3. modeler.

A compressed version of the augmented uniprocessor address trace of a parallel program is input to the preprocessor. The preprocessor divides the trace into a sequence of basic scheduling units, called *tasks*, and produces, as output, a task queue with an embedded process flow graph of the program, which is input to the scheduler. The scheduler dynamically creates a multi-stream virtual-address trace from the augmented task queue. This is done by assigning tasks to the simulated processors of the defined multiprocessor system and dynamically delaying task processing in accordance with feedback from the modeler. During a simulation time step, each of the simulated processors generates zero or one memory references. The number of references generated depends on whether or not a processor has a resident task and, if a processor has a resident task, the nature of the portion of the trace that comprises the task and the unexpired delay time associated with the task. The sequence of memory references created at each time step is input to the modeler, which realizes the simulation model by

- managing the caches, TLBs, physical and virtual memories, and paging I/O subsystem;
- introducing service times for instruction execution, cache and memory accesses, cache and TLB misses, page faults, page-ins, and context switches; and
- recording data that can be used later to study the performance of the simulated system.

This section is divided into four subsections. After explaining how the address trace input to the preprocessor is generated, the next three subsections present a conceptual description of the functions of the preprocessor, scheduler, and modeler, as well as the interaction between them.

6.1.1. Address Trace Generation

The augmented uniprocessor address trace that is input to the preprocessor is produced by executing a parallel program on one processor under a modified version of IBM's PSIMUL [So, *et al.*, 1986]. PSIMUL allows simulation of the entire execution of a parallel program,

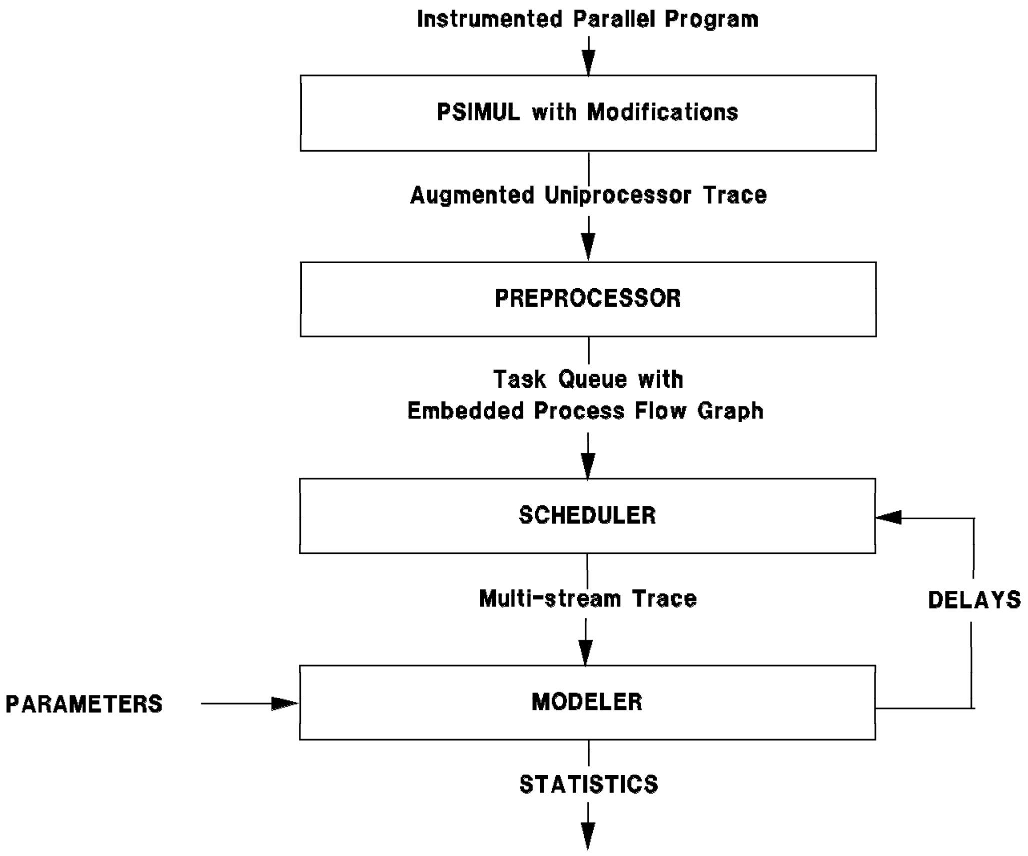


Figure 15. Simulation System

not only specific program sections or simple kernels. The simulation generates a trace of the virtual addresses referenced by the user program. References made by the operating system are not traced.

PSIMUL is built on top of a S/370 instruction-level simulator called SEMUL [Liles and Paul] and is supported by a VM parallel-programming environment called VM/EPEX ([Darema-Rogers, *et al.*, 1985; Stone, *et al.*, 1985]), referred to as EPEX. VM is shorthand for Virtual Machine/System Product (VM/SP), which is an IBM program product that controls virtual machines. According to IBM [1988], "a virtual machine is the functional equivalent of a real computer that you control from your terminal, using a command language of verbs and nouns".

EPEX allows parallel execution of a single program on multiple virtual machines, executing on a time-shared basis. This is done by preassigning n processes to a job (where n is assumed to be less than or equal to the number of processors in the simulated parallel system) and associating each process with a virtual machine. These processes remain in existence from the start of program execution until program termination. When coupled with PSIMUL, the parallel execution of the n parallel processes is simulated by enabling SEMUL for each one of them. In this way, an address trace is generated for each process.

Under EPEX all data that are to be shared by multiple processes are loaded into an address space shared by all the processes. This is implemented under VM by using writeable shared segments to which multiple users can have read/write access. Thus, shared data are referenced as if a single, global address space were defined. Program instructions and local data are considered private to a process. In general, programs run under EPEX make shared read-only data private to a process by copying it to private variables after its initialization.

The program model provided by EPEX is the SPMD, Single-Program-Multiple-Data, computational model [Darema-Rogers, *et al.*, 1986]. EPEX parallel-programming constructs are added to a program in order to distinguish between shared and private data, and to identify different types of parallelism and barriers. The EPEX preprocessor inserts synchronization code in the program to enforce the precedence constraints implied by the the programming constructs. These code segments use both private and shared data.

We modified PSIMUL to place markers in the address trace of a program. These markers are used to identify the individual tasks that comprise the program, code segments introduced by EPEX, and precedence constraints. In addition, they are used to identify the various sections of a program, *i.e*, initialization, computation, and termination, different iterations of the computational section of the program, and differences in the flow of control of the computational section from one iteration to another. As explained in the next section, the markers allow us to use one trace to drive our simulation model for various multiprocessor architectures and, as described in Section 6.2, they allow us to substantially compress the traces of certain types of parallel programs.

In summary, the trace that is input to the preprocessor is a compressed version of the trace of a parallel program augmented by EPEX code segments and markers that enable the preprocessor to identify tasks, EPEX code, and precedence constraints. Given such a trace as input, the preprocessor produces a task queue with an embedded process flow graph, which is input to the scheduler.

6.1.2. Preprocessor

Any parallel program can be represented by a *process flow graph*, an acyclic, directed graph in which nodes represent task groups and edges represent execution constraints [Shaw, 1974]. The preprocessor constructs a process flow graph of a program from the augmented address trace produced by our modified version of PSIMUL. This flow graph and its associated task queue are encapsulated in a set of files created by the preprocessor. Each file represents a different section of the program.

A *task group* is a set of tasks, not necessarily maximal, sharing the same precedence relationship. Multiple tasks (*e.g.*, separate iterations of a parallel loop) intended for parallel execution are in the same task group. A task group that contains no tasks serves solely as a synchronization point in the program. Each task group may have an associated *import task*, which often updates private variables. Each task of a task group at the source of an edge of the process flow graph must be completed before any task in the task group at its sink can be executed. A process must execute all import tasks encountered in its execution of the program, and the import tasks associated with a node of the process flow graph must be executed by each process prior to executing any other tasks belonging to that node. Therefore, such a node could be expanded to two nodes joined by an edge, where one node only contains the import tasks, and the other, the successor of the first, contains all the other tasks of the group.

As mentioned above, markers in the input trace enable the preprocessor to identify tasks and segments of the reference stream corresponding to the execution of EPEX parallel-programming constructs. The collection of the tasks into task groups and the precedence relationships among them are determined by the types of markers associated with the tasks and the semantics of the associated parallel-programming constructs. This method was adopted by So, *et al.* [1987] to implement a speedup analyzer called SPAN that analyzes the process flow graph of a program and provides information to assist the user in attaining maximal speedup.

Some of the tasks delimited in a trace correspond to synchronization code inserted by EPEX. These are deleted by the preprocessor, along with the trace segments that correspond to the execution of EPEX parallel-programming constructs. The markers that identify such code segments can be used to designate positions in the trace where the scheduler can add address sequences that correspond to predefined code segments. For example, suppose we wish to model a self-scheduling system, where a global queue of tasks that are ready for processing is maintained. In this case, processors access and manipulate the queue in order to get tasks to process. In our context, this constitutes an import task which could be associated with each member of every task group. Thus, to model a self-scheduling system, the scheduler could add the address sequence that corresponds to this import task to the beginning of the trace segment associated with each task. The preprocessor provides a list of the synchronization addresses used by the EPEX code which, in turn, can be used in such address sequences.

The parallel programs used in this study employ the following types of tasks:

serial-wait: A task that is only executed by one process; an edge emanates from its associated node of the process flow graph.

serial-nowait: A task that is only executed by one process; it does not carry any inherent precedence constraints.

conditional-serial-wait: A serial-wait task that is only executed when a certain condition holds.

parallel-wait: A task intended for concurrent execution with other tasks; an edge emanates from its associated node of the process flow graph.

replicate: A task that is executed by all processes of the related job; it is, in fact, an import task and, therefore, carries inherent precedence constraints as described above.

barrier: A possibly null task that is executed by all processes; it can either be a task group with no tasks that serves solely as a synchronization point in the program, or an edge in the process flow graph.

Other task types are supported by the simulator, in particular:

parallel-nowait: A task intended for concurrent execution with other tasks; it does not carry any inherent precedence constraints.

conditional-serial-nowait: A serial-nowait task that is only executed when a certain condition holds.

As described above, the preprocessor creates a set of files that encapsulate the process flow graph and associated task queue of a program. These are input to the scheduler, which, in turn, drives the simulation model.

6.1.3. Scheduler

The scheduler dynamically creates a multi-stream virtual-address trace from the process flow graph and associated task queue of a program. This multi-stream trace is input to the modeler, *i.e.*, it drives the simulation model.

The process flow graph of a program directs the scheduling of its tasks. The scheduler can simulate different ways of enforcing precedence constraints. In our simulations the operating system assigns a task to a processor only after the constraints of the task's group have been satisfied. However, other computational models can be used.

For instance, processors can busy-wait until the constraints have been met. Busy-waiting can be implemented using various techniques. Two of these associate a shared synchronization variable with each node. In one case, a task waiting to run inspects all the synchronization variables in its predecessor nodes, which are updated as tasks in those nodes complete. In the other case, completing tasks update synchronization variables in all the successor nodes, and each waiting task needs to check only its own node's variable. The scheduler can model these possibilities by adding to each node an import task that is a predefined code sequence that waits for all tasks of predecessor nodes to complete. Predefined task exit code is added to all tasks in each group; for null task groups, a dummy task containing just this exit code is created.

In addition, task preemption or various *back-off* strategies for busy-waiting can also be implemented. And, considerable flexibility is possible in the selection of the next task within a group to be executed. For example, consecutive iterations of a loop can be assigned to a single processor, implementing *chunking*.

As mentioned above, in our simulations we assume that the operating system enforces precedence, therefore, the scheduler assigns a task to a processor only after the precedence constraints of the task's group have been met. At the beginning of each simulation time step the scheduler assigns allocatable tasks to inactive processors, and then generates a sequence of references that is input to the modeler. The sequence of references generated by the scheduler consists of zero or one references from each processor. A processor generates no reference if it is idle, if it is waiting for a memory reference to be satisfied, or if it is being delayed by the modeler.

A simulation time step is a variable amount of time. The length of a time step is dependent on feedback from the modeler, which introduces delays in task processing (*i.e.*, service times associated with system overheads). Thus, at a specific point in a simulation, the length of the next time step depends on the processor-resident task with the minimum unexpired time delay. Analogous to discrete event simulation [Heidelberger and Lavenberg, 1984], events in the simulation can only occur at a countable number of points in simulated time and not continuously.

6.1.4. Modeler

At each simulation time step, given a sequence of references and the length of the time step, the modeler realizes our simulation model by processing the memory references and updating the states of the simulated processors, caches, TLBs, physical and virtual memories, and paging I/O devices accordingly. At the end of each time step, the modeler transmits information to the scheduler that may affect the scheduling of tasks and the processing of the address trace. In addition, the modeler maintains a record of events, such as TLB reloads and page-ins, that can be used later to produce performance statistics.

The processing of a memory reference may result in a cache or TLB miss and, possibly, a page fault. Thus, the processing of a task may be delayed until a memory reference, cache miss, or TLB miss is satisfied. If a page fault is encountered, task processing is delayed until the referenced page is memory resident. Consequently, the faulting task's processor may start the I/O necessary to make the page resident, busy-wait, or context switch.

The modeler can also introduce random delays, to simulate the delays incurred in the interconnection network and to cause the system to be asynchronous. And, a multiprogramming environment can be modeled by passing the scheduler a process flow graph consisting of disjoint subgraphs, each of which is the process flow graph of a different program.

The processing that occurs during a time step is described in more detail in Section 6.3, where we discuss our simulation model. In this discussion, the distinction between the scheduler and modeler is blurred. This is, in some part, due to the fact that instead of the trace input being a stream of virtual addresses, it is compressed into a stream of records, called events. The processing of events, in turn, affects the design of the simulation system. Thus, before describing our model, we describe the techniques we used to compress the address traces used in this thesis.

6.2. Trace Compression

Compression techniques will be more and more important as larger parallel systems are simulated. The processing of less trace data can reduce the execution time required to effect a simulation and the processing of a trace stored on disk rather than on a magnetic tape medium can also reduce simulation time. For simulations that model a multiprogramming environment such compression techniques are even more important. Compression techniques have gained more attention in recent times, especially with respect to studying the behavior of caches. This is because as cache sizes become larger and larger, longer traces are required. For more information, see [Stone, 1987] and [Laha, Patel, and Iyer, 1988].

Two techniques were employed to reduce the size of the address traces used in this thesis. The first technique depends on the nature of the parallel programs used to generate the traces. Each of these programs is based on an iterative, grid-based algorithm that has one of the following general structures:

```
Initialization

FOR i = 1, N DO IN PARALLEL
   { Computation }
END FOR

Termination
```

or

```
Initialization

WHILE ( x < y ) DO IN PARALLEL
   { Computation }
END WHILE

Termination
```

The flow of control of the computational section of the program differs very slightly or not at all from one iteration to the next. One of the programs we used has only two distinct flows of control: one being predominant, the other occurring at predetermined intervals. The other program has only one distinct flow of control.

Thus, for these programs, as well as other application programs that have these characteristics, it is only necessary to trace each different section of the program. Iterations of the computational section that differ with respect to the flow of control followed during execution of that section of the program are considered different sections of the program. Thus, besides tracing the initialization and termination sections of the program, one iteration of each different execution of the computational section is traced. Markers in the trace created by our modified version of PSIMUL enable the preprocessor to identify each different section of the program, and to create a file for each different section that encapsulates its subgraph of the process flow graph of the program and the associated portion of the program's task queue. Scheduler input parameters describe the process flow graph of the entire program and, therefore, control the sequencing from one subgraph to another. For example, after scheduling the tasks that comprise the initialization section of the program, input parameters may direct the scheduler to next schedule the tasks comprising one type of iteration of the computational section of the program, and, afterwards, to schedule the tasks of a different type of iteration, possibly for a specific number of times. This technique allows the simulation of a longer execution period than that represented by the address trace, and enables one to study variations between iterations of the main computational loop due to different page and TLB resident sets. In addition, this technique can be used to easily construct multiprogramming mixes for input to the simulation model.

The other way in which we compressed the size of the traces used in this thesis was implemented by modifying PSIMUL to fully trace only references to shared read/write data and to count and record merely the number of references made to private and shared read-only data between each pair of references to shared read/write data. For a reference to shared read/write data, we record the referenced virtual address and the type of reference (*i.e.*, a load or a store). This served our purposes for several reasons:

- We assume that shared read/write data are not cacheable, while private and shared read-only data (including program instructions) are cacheable.
- Consistency-ensuring TLB management is targeted mainly at shared memory regions.
- The only unsafe changes that occur in our simulations are the result of the eviction of shared pages.

As illustrated in Figure 16, a noncacheable reference together with the cacheable references that occurred between it and the preceding noncacheable reference is defined as an *event*. Thus, a task is comprised of a sequence of events. The cacheable references that occur between two noncacheable references is called an *interval*. Therefore, an event is defined as a noncacheable reference together with the preceding interval, and the *event length* is equal to the *interval length*, the number of cacheable references in the interval, plus one. Accordingly, the trace input to our simulation model is a sequence of events, rather than a sequence of virtual addresses. The following description of our model is based on this premise.

6.3. Simulation Model

We describe our simulation model by first defining its input and output, and then describing the way in which we model a defined multiprocessor system. Accordingly, this section is divided into three parts. Although both of the techniques discussed in the last section were used to compress the size of the traces used in this thesis, only the latter one changes the nature of the trace that is produced by the scheduler. Thus, in the rest of this thesis we rarely refer to the former technique. We do, however, use terms defined with respect to the latter technique in the remainder of this chapter and in Chapter 7.

6.3.1. Input

As described above, our simulation model is driven by a dynamically created multi-stream trace of events. Besides this input trace, input parameters define the multiprocessor system to be modeled. These parameters specify the size and configuration of the high-level architecture, the architectures of the processors and interconnection network, and the characteristics of the TLBs and memory and I/O subsystems. Input parameters also describe characteristics of the address trace that drives the model, and performance characteristics of the processors.

The high-level architecture is defined by the following input parameters:

- number of processors
  (which is also the number of memory modules),
- number of paging arenas,
- number of I/O units per paging arena,
- size of physical memory,
- size of a paging arena, and
- size of a frame
  (which is also the size of a page).

The architecture of the processing elements, assumed to be RISC processors in our simulations, is defined by specifying the

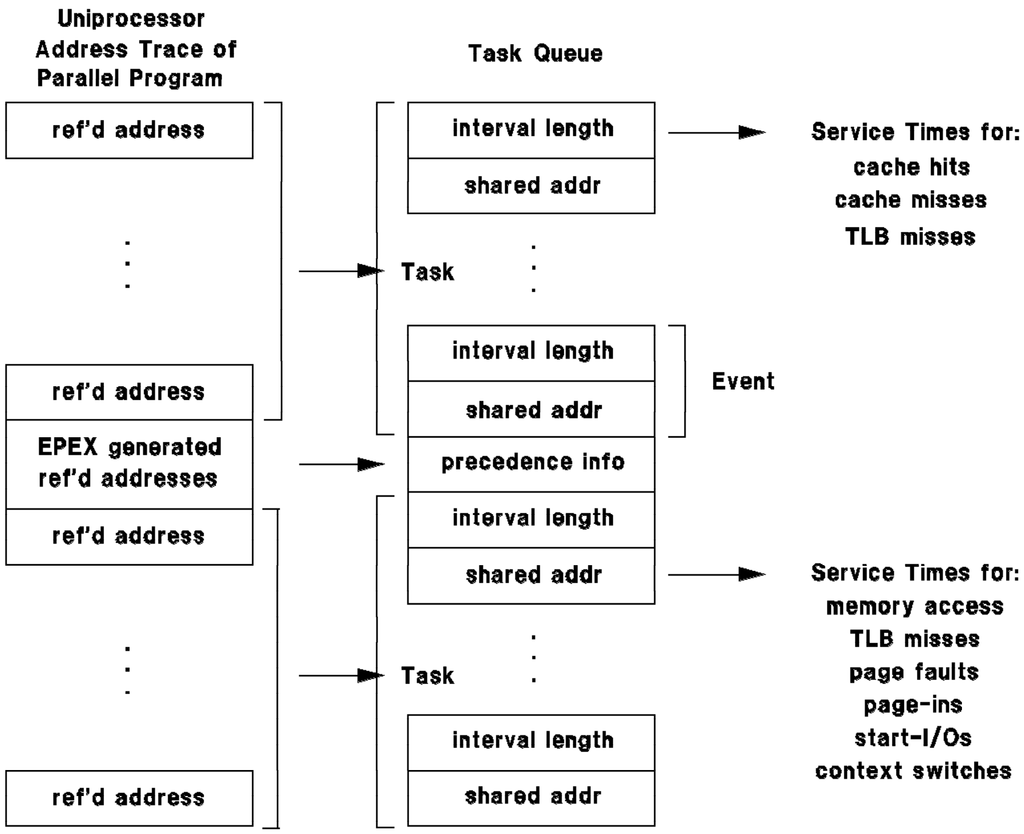


Figure 16. Trace Compression Technique

- MIPS rating,
- cache line size, and
- cache miss request size.

Our model assumes that the processors and memory modules are interconnected by an Omega network. (This can be changed easily by modifying the software module that defines the interconnection between processors and memory.) The architecture of the Omega network is specified by input parameters that define the switch size and the number of packets per message.

Since the main purpose of our simulations is to compare the performance of our TLB consistency solutions, the following input parameters characterize the TLBs and their management:

- number of TLB entries for noncacheable data,
- time to effect a TLB reload under the

adopted TLB consistency scheme, and

- TLB replacement policy.

Our model assumes that a TLB has four partitions: user cacheable, user noncacheable, system cacheable, and system noncacheable. Likewise, virtual memory is viewed as being partitioned in the same manner, and it is assumed that pages other than those in the user-noncacheable partition are locked in main memory. Thus, demand paging is modeled with respect to only user-noncacheable pages, and unsafe changes are the result of the eviction of user-noncacheable pages. Accordingly, only the user-noncacheable partition of TLBs is modeled. Limiting our research in this way seems reasonable since the ratio of user-cacheable to user-noncacheable data is small for large runs of the programs used in this thesis (as shown in Chapter 7), and the number of pages associated with operating system code and data structures is small in comparison to the number required to store user-noncacheable data. Also, it is conceivable that the operating system and a number of private pages would be locked in memory.

Since the multiprocessor system being modeled is assumed to implement demand paging, at least with respect to user-noncacheable pages (as explained above), the following parameters define some characteristics of the memory and paging I/O subsystems:

- main memory access time,
- mapping of pages to frames,
- page replacement policy,
- I/O transfer rate of auxiliary memory, and
- interleave factor, *i.e.*, the way in which a frame is interleaved across a cluster of memory modules, *e.g.*, on a two-word basis.

Since the elements of the trace input to our model are events rather than simple virtual addresses, we use input parameters that characterize the performance of the processors to, in a sense, reconstitute the references to cacheable data that are represented by the interval lengths associated with events. These parameters represent estimates of the average number of references made per instruction and the ratio of loads to stores, as well as the mapping from a reconstituted S/370 interval to an interval for the type of RISC processor we assume for the processing elements of the defined multiprocessor system. Using this information plus the average length of an event in the computational section of the program trace and estimates of cache and TLB miss rates, the average network load for the simulation is estimated. Since the architecture of the interconnection network is defined, the model then computes the average network waiting time for main memory requests of various lengths. This average network waiting time is used to determine the service time (in processor cycles) for a memory request that is transmitted through the network. The network load is assumed constant throughout a simulation. Network traffic due to page-in service is not taken into consideration in the computation of the average offered network load.

6.3.2. *Output*

Since a simulation takes a significant amount of computer time, our model was designed to collect much more data than was needed to achieve our main goal of being able to make a relative comparison of the performance of our solutions to the TLB consistency problem. For example, the data we collected includes the following:

- processor time,
- number of TLB misses and reloads,
- number of page-ins and page faults,
- number of cacheable and noncacheable references,
- number of load and stores (cacheable and noncacheable), and
- I/O time.

Processor time is broken down into time attributable to

- accessing cacheable data,
- accessing noncacheable data,
- TLB management,
- context switches,
- I/O startup for page-ins,
- busy-waiting for faulted pages, and
- idleness.

This data are further classified in terms of related program sections and, in many cases, in terms of related task types.

Having defined the input and output of our simulation model, we now describe the model, itself.

6.3.3. *System Model*

We describe the way in which we model a defined multiprocessor system by first describing what occurs during a simulation and then indicating how the relevant system components are modeled. As mentioned above, our model's input trace is a sequence of events, rather than a sequence of virtual addresses. When an event generates a memory request, cache miss, TLB miss, or page fault, the associated service time and the event's noncacheable reference are recorded as part of the task's state. The delay associated with the event (initially equal to the assessed service time) is reduced at subsequent simulation time steps, according to the size of the time steps. Only events associated with tasks that are assigned to processors and are not being delayed can be processed during a simulation time step.

6.3.3.1. Simulation Model

To describe our model we start at the beginning of a simulation. A simulation begins with a cold start, *i.e.*, the processors and I/O devices are idle, and the TLBs and main memory are unpopulated. The first tasks to be assigned to processors are those that do not have any precedence constraints. Since each task is a sequence of events, a simulation proceeds by processing the events of tasks assigned to processors until there are no more tasks to assign. At this point, the simulation is complete. Thus, by describing how an event is processed and how a time step is defined, we, in effect, describe our simulation model.

Recall that an event represents a noncacheable reference and the preceding interval of cacheable references. An event is processed by first processing its reconstituted cacheable references and then processing its noncacheable reference.

Since the only information gathered regarding cacheable references is the length of the interval associated with an event, input parameters to the simulation model allow processor time to be accounted for as if more information were available about the nature of each interval. An interval of cacheable references is, in effect, reconstituted by estimating the mix of instruction and data references in the interval and the number of data references that are loads. In addition, as described in Chapter 7, in our simulations a reconstituted S/370 interval is mapped to a corresponding interval for a RISC processor.

Given a reconstituted interval and the input cache and TLB miss rates for cacheable data and instructions, a service time, in terms of processor cycles, is assessed on the interval. This is done by computing the time that would be required to execute the instructions in the interval, including the time to satisfy cache and TLB misses, which require memory accesses and network transit. The information required to compute the cost of an interval either is defined by input parameters or, like the average network waiting time, is computed using input parameters at the beginning of a simulation. In Chapter 7, we follow through an example that computes the service time of an interval. The processing of the noncacheable reference of the event and the subsequent events of the task is delayed until the processing time for the interval has elapsed.

The processing of a noncacheable reference can result in a

- TLB hit,
- TLB miss on a resident page, or
- TLB miss on a nonresident page.

A service time, which corresponds to the processing that would occur in a real system, is associated with each of these possibilities.

We assume that a fixed TLB access time is hidden in the instruction cycle or memory access time. Thus, the service time for a TLB hit is composed of the time for transmitting a memory request from the processor to a memory module, memory access, and transmitting the memory response to the processor. As is shown in Chapter 7, in our simulations loads and stores have different service times. Processor time is accounted for accordingly, delaying further processing of the assigned task. After this delay time has elapsed, the memory request is considered satisfied. During the next simulated processor cycle the reference is considered processed. At this point, if the resident task is not complete, the processing of the task's next event begins at the following processor cycle. If the task is complete, then the processor is considered idle and is assigned a task at the next simulation time step, if one is available.

The service time for a TLB miss on a resident page is the service time for a TLB reload, which is an input parameter. In Chapter 7 we show how we arrived at the service time for a TLB reload for each of our TLB consistency schemes. After the service time for a TLB reload has elapsed, if the referenced page is still memory resident, the appropriate TLB entry is loaded, evicting another TLB entry if necessary. The noncacheable reference is then reprocessed. If the processor had been delayed by a page fault, the page is now resident in memory. At this point, unless the loaded TLB entry has become invalid, a TLB hit ensues. Otherwise, another TLB miss is processed.

A TLB miss on a nonresident page can result in the processor

- starting page I/O,
- context switching to another task, or
- busy-waiting until the page is memory resident.

Input parameters define the services times for starting I/O and context switching, as well as service time for reading a page from or writing a page to auxiliary memory. If a processor faults on a page that is not currently being paged in, it starts the I/O necessary to make the page resident, *i.e.*, allocates a frame to the page, schedules the I/O for the page on the I/O unit with the minimum waiting time, and updates the necessary data structures used to manage the paging system.[4] If the allocated frame stores a dirty page, then this involves writing the dirty page to auxiliary memory before reading the referenced page into main memory. After starting I/O, the processor can either context switch or busy-wait until the page is memory resident. If the processor faults on a page that is being paged in, it either context switches or busy-waits until the page is memory resident. The service time for page faults is hidden in the idle time experienced by processors waiting for the pages to become memory resident, the context switch service time, or in the I/O start service time.

A processor context switches only if there is an allocatable task for it to process and if the time until the referenced page is to be memory resident is greater than a lower bound specified by an input parameter. When a processor context switches, it puts its resident task on the list of blocked tasks and then, after the context-switch service time (defined by an input parameter) has elapsed, it is eligible to execute another task.

Thus, at a specific point in a simulation, the delay time associated with a processor-resident task is attributable to

- a sequence of cacheable references (including the cache misses and TLB misses generated by them),

[4] A cluster controller associated with each paging arena could take on the responsibility of starting page I/O. In this case, network traffic due to the execution of the page fault servicing routine would be minimized, but page-fault handling could become a bottleneck if one controller is the sole page handler for a cluster.

- a memory access triggered by a noncacheable reference,
- a TLB miss on a noncacheable reference,
- a page fault on a noncacheable reference, or
- a context switch.

The service time associated with an interval of cacheable references is a function of the input parameters that describe

- the multiprocessor architecture being modeled,
- the processor architecture,
- the interconnection network architecture,
- cache miss and TLB miss rates, and
- the characteristics of the program trace and the architecture of the machine on which it was generated.

The service time associated with a noncacheable reference is a function of the input parameters that describe

- the multiprocessor architecture being modeled,
- the processor architecture,
- the interconnection network architecture,
- the assumed input/output facilities,
- context-switch overhead, and
- TLB management overhead.

As can be gleaned from the above discussion, the states of processors, TLBs, I/O devices, and tasks, as well as the states of the physical and virtual memories, are continuously updated during a simulation. These state transitions are the subject of the next section, where we discuss the way the relevant system components are modeled.

6.3.3.2. System Component Models

Task processing propels a simulation and, as described above, a simulated processor is a vehicle for task processing. A processor is modeled by a data structure that stores state information directing the execution of the simulation and collected data that can be used later to study the performance of the simulated system.

A processor can be in either an active or inactive state. If active, it has a resident task that it is processing; if inactive, it does not have a resident task and, therefore, can be assigned work. Initially all processors are inactive. At each simulation time step, allocatable tasks are assigned to inactive processors (processors that are assigned tasks move to the active state) and the unelapsed delay times of active processors are reduced by the length of the time step. A processor becomes inactive when it completes the processing of a task or when its task is blocked on page I/O and, as a result, is removed from the processor and put on a queue of blocked tasks. A simulation is complete when there are no more tasks to assign to processors and all assigned tasks have been completely processed. As in discrete event simulation, the list of active processors is maintained in order of increasing unelapsed delay time, and the active processor at the head of the list determines the length of the next simulation time step. Thus, when multiple processors have the same amount of delay time left, a simulation time step may correspond to an empty time interval. In this case, the processors are simulated according to the order in which they are on the list. The list of inactive processors is managed as a queue, *i.e.*, first-in first-out.

As described above, the processing of a task can initiate a TLB miss and possibly a page fault. Each entry of the user-noncacheable portion of the system's TLBs is modeled explicitly. Chapter 7 includes a description of the TLB entry that we assume in our simulations. Dirty and reference bits are associated with each TLB entry and, thus, during a simulation, we update this information accordingly. In our simulations, TLBs are fully associative and are managed using a least-recently-used (LRU) replacement policy. Thus, each TLB is modeled as a list of entries maintained in LRU order.

Since page faults and the resulting page-ins are modeled, the state of the devices that comprise the I/O subsystem are also updated at each simulation time step, in addition to the states of the virtual and physical memories. The initial fault on a page initiates its page-in process. When a page-in is initiated, a frame is allocated to the page and the I/O device with the smallest workload is assigned the first I/O task associated with making the referenced page memory resident. Thus, the data structures that represent the I/O devices associated with each paging arena comprise two lists: a list of active devices and a list of inactive ones. The list of active I/O devices is ordered by increasing workload and a queue of I/O tasks is associated with each I/O device. When a device is assigned an I/O task, its workload (in processor cycles) is incremented by the time it will take to process the task. If multiple I/O tasks are required, *e.g.*, the page stored in the allocated frame must be written to auxiliary memory, the completion of one I/O task triggers the scheduling of the next. In our simulations, if the page stored in the frame is referenced while it is being written to auxiliary memory, we allocate another frame to the referenced page and then schedule the necessary I/O. In addition, in our simulations, we adopt a LRU page replacement policy. Thus, the data structures that represent frames are managed as two lists that represent allocated and unallocated frames. The allocated frames are maintained in LRU order, while the list of unallocated frames is maintained as a queue. (The data structures that represent virtual pages are maintained as an array.) Since the I/O devices are used only for paging I/O, the state of a device is summarized by the page it is reading from or writing to main memory, the time remaining until the current I/O task is complete, and its backlog of work. Thus, updating the states of these devices involves reducing the remaining processing time of currently active I/O tasks, starting up queued I/O tasks as active I/O tasks complete, making pages associated with completed I/O tasks memory resident. Data structures that record the relationship between frames and pages are updated accordingly and tasks that are on blocked task queues as a result of faulting on now memory-resident pages are moved to queues of tasks that can be assigned to processors.

The task is the component of the simulation model that is the common thread among the other components. Tasks fall into two major categories:

1. those that belong to tasks groups with precedence constraints that have not been satisfied and
2. those that belong to tasks groups with either no precedence constraints or with precedence constraints that have been satisfied.

Consider a job to be the processing of a program task queue, and a job task queue to be that portion of a program's task queue (created by the preprocessor) that contains tasks that have not been assigned to processors. Then, a task in the first category is a member of a locked job task queue, while a task in the second category is a member of an unlocked job task queue. A job task queue is locked if the precedence constraints of the task group at the head of the queue are still to be satisfied or if the task at the head of the queue is a barrier task. Thus, the synchronization of the processors assigned to a job can be implemented by locking the job's task queue until all its tasks that have been assigned to processors have been completely processed. Precedence constraints, in accordance with the parallel-programming constructs supplied by EPEX, are implemented in this fashion.

A task is considered allocated once it is assigned to a processor and, thus, processor resident. Tasks are initially allocated from an unlocked job task queue. Once allocated, a task can be in one of four states: It can be resident on a

- processor,
- processor's replicate task queue,
- job's blocked queue, or
- job's ready queue.

When a replicate task (a task to be executed by all processors assigned to the corresponding job) is allocated, a copy of the task is, in essence, added to a replicate task queue of each processor assigned to the corresponding job. Thus, in allocating a replicate task, N tasks are actually allocated. A processor is only assigned one of a job's tasks, other than a replicate task, if its replicate task queue associated with the job is empty. Otherwise, it is assigned a task from its replicate task queue.

A processor-resident task moves to a job's blocked queue on a page fault or after starting page I/O, if the time until the referenced page is to become memory resident is greater than some lower bound and there is an another allocatable task that can be assigned to its processor. The lower bound is specified by an input parameter that represents the point at which context switching seems reasonable in this situation. In our simulations, this lower bound is twice the time we estimate for context switching and page/frame table management in user mode. Since a task on a job's blocked queue is allocatable, when there are inactive processors and no other allocatable tasks to assign to processors, tasks on blocked queues are assigned to them. In this way, if processors are, in effect, idle as a result of page I/O, processor time is accounted for as such. Additionally, this ensures that a simulation time step is defined by the active task with the minimum unelapsed delay time.

A task moves from a job's blocked queue to its ready queue when the page becomes memory resident or when additional I/O is required, *i.e.*, subsequent I/O tasks must be processed to make a page memory resident. Thus, inactive processors are assigned tasks from their respective replicate task queues, ready queues, unlocked job task queues, and blocked queues, in that order. As described above and as shown in Figure 17, once allocated, a task can migrate.

As can be seen, processors and I/O devices execute in parallel in our simulation, and context switching on page faults magnifies this concurrency. During a simulation any one of the three TLB consistency schemes described in Chapter 5 can be modeled.

6.4. Software

This section catalogs the software we built to implement the simulation system described in this chapter. Since Section 6.1.1 describes the modifications we made to PSIMUL, we do not discuss the software for address trace generation here. We do, however, discuss the software related to the preprocessor, scheduler, and modeler, as well as the software that we built to postprocess the resulting data files. All our software was written in C and has been run on IBM S/370 systems and on IBM RTs.

The preprocessor consists of one program, called **readtap.c**, and two include files, **common.h** and **readtap.h**. Given a file that is a program's address trace produced by our modified version of PSIMUL and stored on an inputted number of tapes, a number of files, the names of which are specified by input parameters, are created. These output files include:

- a task queue associated with the initialization section of the traced program,
- a task queue associated with each different computational section of the program,
- a task queue associated with the termination section of the program,
- the synchronization addresses used by EPEX-inserted code segments, and
- the smallest and largest referenced virtual addresses (used by the program that implements the scheduler and modeler to define data structures).

These files are input to the program that models both the scheduler and modeler. This program is comprised of eleven modules:

contexts.c: services a processor's request to context switch.

job.c: creates and initializes the data structures to model jobs.

memory.c: creates and initializes the data structures to model paging arenas, I/O devices, virtual memory, and physical memory.

network.c: computes the average network load for a simulation and the average network transit times (including queueing time) for various sized messages.

page.c: services page faults and page-ins, including I/O device management.

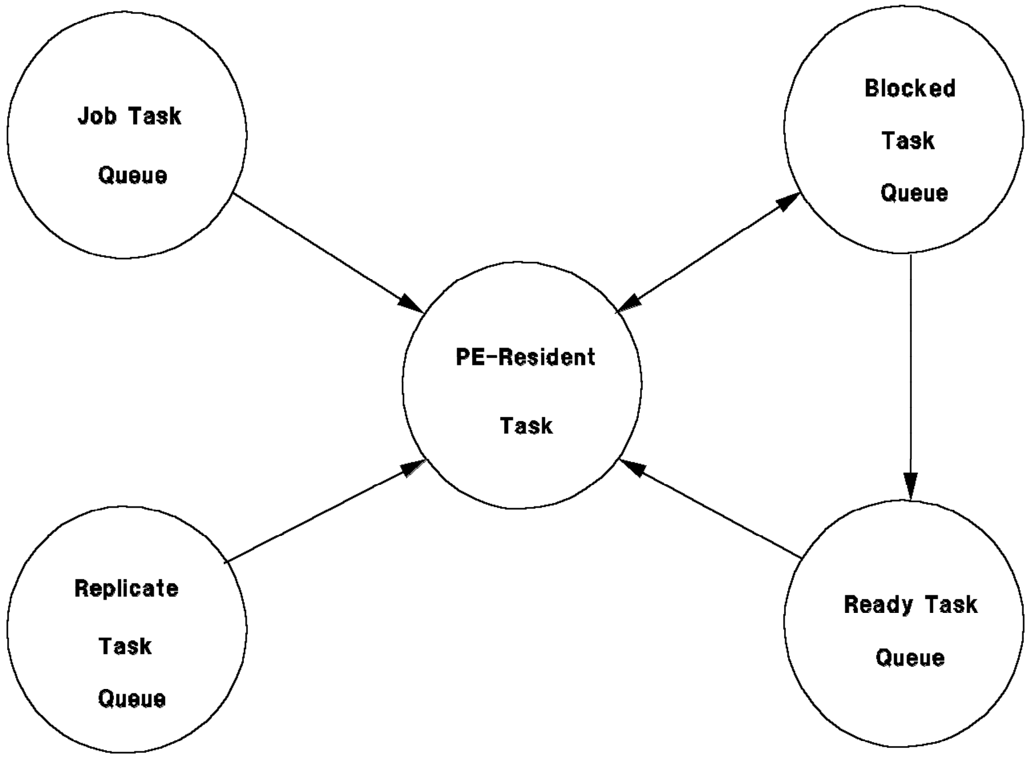


Figure 17. Migration Paths of Tasks

results.c: outputs the simulation results with textual interpretation.

sched.c: main module; schedules tasks to processors and directs the flow of control of a simulation.

statedum.c: on error, outputs the states of system components.

task.c: manages the task queues and accesses task events.

tlb.c: creates and initializes the TLB data structures, services TLB hits and misses, and implements the adopted TLB consistency scheme.

writedat.c: writes the raw output data to the specified file.

An include file is associated with each of these modules which defines other include files, compiler directives, macros, internal variable definitions, and external variable definitions. The include files shared by modules are:

archdefs.h: defines the high-level architecture and processor architecture.

conswdef.h: defines the parameters associated with a context switch and a start-I/O task.

globalde.h: defines the global constants.

jobdefs.h: defines the data structure that models a job.

listmacro.h: defines the macros used for list manipulations.

memoryde.h: defines the data structures that model a page, a frame, a TLB, and an I/O device.

pedefs.h: defines the constants used in managing the data structures that model processors.

taskdefs.h: defines the data structures that model a processor, a task, and entries for each type of task queue.

tlbdefs.h: defines the data structure that models a TLB.

To conserve storage space, the results of a simulation are usually output only as a raw data file, *i.e.*, without textual interpretation. Another program, which is comprised of two modules, **readdata.c** and **genresul.c**, and their corresponding include files plus the include files **readmemd.h** and **genresul.h**, transforms the raw data file into a file that presents the data with textual interpretation. Specific data can be extracted from the raw data file in order to plots graphs. This is done by using yet another program, called **readgraph.c**.

Chapter 7. Simulation Input

As described in Chapter 6, a simulation is propelled by the processing of events by processors. Recall that a sequence of events comprise a task, which is the basic scheduling unit in a simulation. And, an event is defined as a noncacheable reference and the preceding interval, *i.e.*, a count of the cacheable references between the event's noncacheable reference and that of its task's preceding event. Service times associated with the processing of events allow us to simulate the processing of a program on the defined multiprocessor architecture. An event's service time is the sum of the service times assessed on its interval and noncacheable reference. An interval's service time consists of the time required to access and process cached instructions and data and to satisfy cache and TLB misses. The service time associated with the processing of a noncacheable reference consists of the time required to access main memory, process the accessed data, and possibly satisfy TLB misses and page faults.

The simulation input parameters determine the service times for cache and memory accesses, cache and TLB misses, page faults, page-ins, and context switches. Input parameters define architectural characteristics of the processors which, in turn, define the instruction cycle time and cache and TLB access times. The specification of the I/O subsystem, the page replacement policy, and the mapping of pages to frames determine the service times for page faults and page-ins, while input parameters define the overheads associated with starting page I/O and context switching. The service times for cache and TLB misses and noncacheable data accesses are dependent on the service time for satisfying a main memory request. In turn, the service time for a main memory request is dependent on, among other things, the size of the simulated multiprocessor architecture and the architecture of the interconnection network.

The architectures of the multiprocessor and network dictate the length of the path from a processor to a memory module and the network transit time, assuming that there is no network contention. Given data regarding the characterization of an average interval of the input program address trace, an average offered network load is computed that is used to determine the average amount of time a main memory request will be delayed in the network. With an average network waiting time and the memory access time defined, the service times for various kinds of network requests are computed. Accordingly, given the sizes of cache lines and cache miss requests, the services times for cache misses are computed; and given the routines used to perform TLB reloads, the service times for TLB misses are computed.

Since the goal of our simulations was to attain performance measurements that would allow us to make a relative comparison of our proposed TLB consistency methods, we varied only the sizes of the modeled multiprocessor architecture, the TLBs, and the physical memory, and, of course, the TLB consistency algorithm. In the majority of cases, one paging arena was assumed. When two were assumed, each was of the same size. To limit the influence that the I/O performance has on the simulation results, the number of I/O devices per paging arena was defined to be equal to the maximum parallelism that the traced program could utilize. All other input parameters remained constant during the simulations.

In this chapter, after indicating which parameters were varied during the simulations and how they were varied (see Section 7.1), we show how the service times for cache and memory accesses, cache and TLB misses, page faults, page-ins, and context switches are derived. While deriving these service times in Sections 7-2 – 7-8, we define the input parameters that were kept constant during our simulations and justify the assigned values. To illustrate the ideas presented in Sections 7-2 – 7-8, while deriving the average network load for a simulation, Section 7.8.1 follows through the computation of the service time for a sample event. Finally, in Sections 7.9 and 7.10, we summarize the input parameters used in our simulations and describe the programs that were used to generate the input traces.

7.1. Variable Parameters

Our simulations were for multiprocessor systems where each memory module is equidistant from each processor and an Omega network interconnects processors and memory. The simulated multiprocessor systems range from a one-processor system (without an interconnection network) to a 256-processor system.

A set of simulations were run for each different sized multiprocessor system. For each of two program traces and for each of the three TLB consistency methods described in Chapter 5, simulations were run with three different TLB sizes and two different physical memory sizes. Assuming a 4-Kbyte page size, the noncacheable partition of the TLBs consisted of 16, 32, or 64 (page table) entries. The number of frames for shared read/write (noncacheable) data was set to either 33% or 50% of the number of shared virtual pages. A fluid dynamics application and a hydrodynamics application, respectively called Peskin and Simple and described in Section 7.10, were used to generate the input address traces.

As stated above, most of the simulations assumed one paging arena. Simulations that adopted Memory-based TLBs, were run twice: once with one paging arena and another time with two paging arenas, where each arena was of equal size. In order to minimize the effect of I/O performance on the service times for page faults and page-ins, the number of I/O devices per paging arena was fixed to be equal to the maximum parallelism that the traced program could utilize. The number of I/O devices per paging arena for Peskin was fixed to be 512; for Simple, the number was fixed to be 1024. Other than the service times for TLB misses, which, as described in Section 7.5, vary with the adopted consistency scheme, all other input parameters were kept constant. Their assigned values are defined and justified in the following sections.

7.2. Instruction Cycle Time and Cache and TLB Access Times

The definition of the processor architecture determines the processor cycle time and the number of processor cycles required to access and process a cached reference. In our simulations each processor of the multiprocessor architectures studied is assumed to be a pipelined RISC (reduced instruction-set computer) processor (*cf.*, [Hennessy, 1984; Patterson, 1985]) with multiple register sets, idealized so that all instructions execute in a single processor cycle and memory delays for cache hits cause no processor wait states,[5] *i.e.*, a "real" instruction, not a "NO-OP", is executed each cycle. We further assume a Harvard architecture, *i.e.*, two separate buses (and correspondingly two caches), one for data and one for instructions, so that the processor can generate one instruction and one data reference every cycle. If both of these references are cached then an instruction can effectively be executed in one cycle. In addition, when TLBs are located at the processors, we assume that a fixed TLB access time is hidden in the instruction cycle time. When TLBs are located at the memory modules, we assume that the same fixed TLB access time is hidden in the memory access time. The MIPS rating for a pipelined RISC processor is used to define the processor instruction cycle time or clock cycle. In our simulations we assume a 10 MIPS rating (since 10-20 MIPS microprocessors of modest cost are available today). For example, the Am29000 is in this range [*cf.*, AMD; Gimarc and Milutinovic, 1987].

As described above, references to cached instructions and data are effectively satisfied during the instruction cycle. In contrast, instructions that require data accessed from memory cannot be executed in one cycle since they are delayed by network transit time plus memory access time. Memory accesses are generated by cache and TLB misses and references to noncacheable data. A noncacheable reference is initially charged one cycle for processing in addition to the network request service time (described below in Section 7.3) required to satisfy the reference.

7.3. Network Request Service Time

In order to access memory a processor transmits a request through the network; after memory is accessed, a response is transmitted via the network from memory to the requesting processor. Thus, we refer to memory access requests as network requests. As stated in Section 7.1, our simulation model assumes that an Omega network interconnects processors and memory and that each processor is equidistant from each memory module. Thus, service times for network requests increase with system size, since as the system size increases, so does the path from a processor to a memory module. The service time for a network request also depends on the operation being executed (*i.e.*, whether it is a load or a store), the number of packets that comprise the network request, the memory access time, and the network waiting time which, in turn, depends on the intensity of the network traffic.

In this section we address each of these components of the network request service time, with the exception of network waiting time, which is discussed in Section 7.8. After discussing the service times associated with cache and TLB misses, page faults, and page-ins, and the way in which an interval is reconstituted, we discuss the computation of the network waiting time. We choose this order of discussion because the average network waiting time depends on the average intensity of the network traffic, called the *offered load*. In turn, the average offered load depends on the average event size of the traced program and the characteristics of such an event, *i.e.* the ratio of instruction references to data references in the event's interval, the ratio of loads to stores, the instruction cache miss rate, the data cache miss rate, and the TLB miss rate.

---

5 In an attempt to achieve one-cycle-per-instruction performance, RISC optimizing compilers rearrange instructions in order to use the cycles after a delayed branch and a delayed load [*cf.*, Patterson, 1985]. More than 90% of the time these compilers are successful in filling the instruction slot following a load or a branch. For example, the IBM 801, Am29000, Berkeley RISC, and Stanford MIPS all use *delayed branches*, one or more instructions following the branch are always executed whether the branch is taken or not. A similar technique is employed to schedule load instruction delays on the Am29000 and MIPS.

7.3.1. Load vs. Store

Load and stores are treated differently by our simulation model. Processors are delayed by network requests generated by loads of noncacheable data and cache read misses, but in most cases they are not delayed by network requests generated by stores or cache write misses. (Although the fetch-and-add operation was utilized in the programs, the trace did not distinguish between data associated with a load and data associated with a fetch-and-add operation.) A load delays a processor for a time interval equal to the service time associated with the generated network request. Some of this time may be overlapped with other operations, especially on a processor like the Am29000. Accordingly, we assume that a store of cacheable data is immediately acknowledged by the network [Kenner, Dickey, and Teller, 1989] and, thus, a processor is not delayed since the store can be executed in one processor cycle. Likewise, when Read-locked TLBs is adopted, stores generated by references to noncacheable data do not delay the processor. In these cases sequential consistency may be violated. Lamport [1979] deems a system to be *sequential consistent* if "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program". As described by Dubois, Scheurich, and Briggs [1988], if memory accesses are atomic and are performed in program order then memory accesses are said to be *strongly ordered* and sequential consistency is maintained. In the cases described above only accesses to the same memory module by the same processor are strongly ordered. We believe that the comparative performance of the consistency algorithms presented is not affected by our overly aggressive memory policy. In contrast, if Validation or Memory-based TLBs are adopted, stores generated by references to noncacheable data do delay the processor. In the case of Validation, this is because a store will not be successful if stale translation information was used and in the case of Memory-based TLBs, a page fault may ensue. Our assumptions are reasonable for the programs simulated since these programs are assumed to be free of data races, *i.e.*, situations in which concurrent processes may read and write the same memory location in any conceivable order. (See [Adve and Hill, 1990] for a formal definition of data-race-free synchronization models.)

As explained in Section 7.8, even when a store does not delay the processor, it still contributes to the offered load, which influences the service time of other network requests.

7.3.2. Network Architecture and Memory Access Time

The network architecture and the switch speed determine the net transit time of a network request, *i.e.*, the time it takes for a request to be transmitted through the network when there is no contention at the switches. Adding the memory access time and the network waiting time to this, we arrive at the network request service time.

An Omega network has $\log_2 N$ stages, where N is the number of processors. Since we assume that the switch cycle time is equal to the processor cycle time (for an idealized 10 MIPS RISC processor), the time it takes to traverse one stage of the network is equal to one processor cycle. Therefore, the net network transit time is equal to $\log_2 N$ cycles. Since the access times for DRAMs range from 80 ns to 120 ns, we assume that the time required for memory latency (including the time needed for the control logic) is equal to twice the processor cycle time. Thus, the network request service time (in cycles), hereinafter referred to as T_m, is

$$\omega + (m - 1) + \text{network waiting time} + 2,$$

where $\omega = 2 \log_2 N$, the time it takes for a round-trip through the network with no contention at the switches, m is the number of packets that comprise the request, and the network waiting time is the sum of the queueing delays at the switches. We do not model contention at the memory.

In general, $\omega = 2u \log_2 N$, where a switch is composed of u pipelined stages, each with an execution time of one cycle. For faster processors it may be the case that $u \geq 2$; the network latency is thus increased while the network bandwidth and processor execution rate remain unchanged. Although it is hoped that multiple outstanding requests to the network will help hide the high latency of the network [Dickey, Gottlieb, Kenner, and Liu, 1987], the simulator does not overlap requests nor does it support prefetching.

Table 3 shows the types of network requests supported by our simulator and the number of packets associated with each request and its corresponding network response. Note that the number of packets that comprise a request is not always equal to the number of packets that comprise its related response. In the cases where they are unequal, the value of m is equal to the number of packets that comprise the response.

Network Request Type	Packets Per Request	Packets Per Response
Single-word Load	2	2
Double-word Load	2	4
Single-word Store	2	2
Double-word Store	4	2

Table 3. Number of Packets for Network Requests and Responses

7.4. Cache Miss Service Time

The processors in the modeled multiprocessor systems are assumed to have separate instruction and write-through data caches each with a four-word (16-byte) line size. Since memory is assumed to be interleaved on a two-word basis, a cache read miss generates two double-word load network requests, which are not overlapped, and a cache write miss generates two double-word store network requests.[6] Thus, the service time for a cache read miss is

$$2 \times T_4.$$

As mentioned above, although the network requests generated by cache write misses influence the value of the offered load to the network (as described in Section 7.8), the associated service times do not delay the processors.

7.5. TLB Miss Service Time

A TLB miss results in a TLB reload (or a page fault followed by a subsequent TLB reload). TLB reload code involves accessing a system page table to determine the required translation

[6] A cache write miss should generate only one double-word store network request, rather than two. However, this overestimate is not significant since the simulator does not delay a processor on a store and the network load is low, despite our overestimate.

and setting the appropriate TLB entry. When employing any one of our TLB consistency methods, a TLB reload requires read, and possibly write, locks (sometimes implicit rather than explicit, as in Memory-based TLBs). The service time for a TLB reload depends on the code used to effect the reload and the service times associated with cached and noncached references (discussed above).

When referencing cacheable data, a TLB miss results in the execution of a TLB-reload routine that generates a reference to a page table entry that is cacheable and, therefore, may result in a cache miss. In contrast, when referencing noncacheable data, the TLB-reload routine executed to service the TLB miss generates a reference to a shared read/write page table entry that is, itself, not cacheable. As described in Chapter 6, only private and shared read-only data are cacheable, and only noncacheable references can generate page faults, page-ins, and context switches. Both cacheable and noncacheable references generate TLB misses, but the corresponding TLB-reload routines differ because we assume that special measures for TLB consistency are required for only noncacheable data. In fact, if processes are allowed to migrate and TLBs are not flushed on context switches, if TLB consistency is not guaranteed for private and shared read-only pages as well, it is possible that a process when migrating back to a processor on which it was previously resident may find stale translation information for these pages in the processor's TLB.

7.5.1. TLB Hit Rate for User-cacheable Partitions

The user-noncacheable partitions of the TLBs are explicitly modeled. A TLB is assumed to be fully associative and, as discussed in Section 7.1, in our simulations the size of these partitions is assumed to be either 16, 32, or 64 entries. The user-cacheable partitions are not explicitly modeled. Instead, an input TLB hit rate is used to generate TLB misses on cacheable data. A TLB hit rate of 99% is assumed for the user-cacheable partitions of the TLBs, *i.e.*, for user-cacheable instructions and data.

A 98% hit rate was measured for National Semiconductor's 32082, an MMU that is fully associative, has 32 entries, 512-byte pages, and a clock-like replacement algorithm, [National Semiconductor Co., 1982]. And, the VAX-11/780's TLB, which is two-way set-associative, split supervisor/user, and has 128 entries, 512-byte pages, and random replacement, was measured as having a hit rate of 97.5% [Clark and Emer, 1985]. The experiments of Clark and Emer, using hardware to monitor the VAX 11/780, calculated TLB miss ratios separately for instruction stream versus data stream and for process space versus system space. The results show that the TLB miss ratios for the process space versus the system space are strikingly and consistently different across the four categories. In particular, system space miss ratios are much higher than process space miss ratios. The average process space hit rate is approximately 99%. Since we are only studying the user or process space, we use this 99% estimate. As will be seen in Section 7.8, we also use this same hit rate for the user-noncacheable partitions of TLBs when computing the offered load to the network.

7.5.2. TLB Miss Service Times in Real Machines

In order to have points of comparison, before we present the TLB-reload routines used in our simulations and compute the associated TLB miss service times, we first discuss the cost of TLB misses in the IBM System/370 Model 168 and VAX 11/780. Baer [1980] points out that in the IBM System/370 Model 168, if the page table entry is in the TLB, only one processor cycle is necessary for the mapping, while if it is not there from 8 to 26 cycles are spent in the table look-up (assuming no page faults). To measure the cost of TLB misses on the VAX 11/780, [Clark and Emer, 1985] employed a hardware monitor to count the number of cycles between entry and exit of the applicable microcode routines. The routine in its simplest form is roughly 17 microinstructions long; there is some minor additional overhead (one or two cycles) on entry and exit. If the page table entry is not in the cache, reading it may provoke a cache miss and, on a VAX, a subsequent processor wait, or stall, of about 6 cycles. According to the experiments of Clark and Emer, TLB misses take about 22 cycles each on the VAX 11/780, including cache stall and routine entry and exit overhead. The average length of the cache stall is 3.5 cycles. (In order to compare these costs with the data below, the number of cycles to service a TLB miss must be divided by the average number of cycles per instruction for these processors.)

7.5.3. TLB-Reload Routines and Service Times

Our simulation model uses the TLB-reload routines presented in Figure 19 and Figure 21 – Figure 23 to compute service times for TLB misses. The routines of Figure 21 – Figure 23 are meant to be used for TLB misses on noncacheable data as each one ensures TLB consistency. These routines correspond to the three TLB consistency methods described in Chapter 5: Validation, Read-locked TLBs, and Memory-based TLBs, respectively. Based on these routines, and without taking into account network transit or waiting times and assuming that 100% of the TLB reload code is cached, the associated service times for TLB misses on noncacheable data range from 7 to 34 cycles. Since it may not be necessary to guarantee the consistency of TLB entries for private and shared read-only pages, when Read-locked TLBs or Validation is simulated, a simplified, generic TLB-reload routine, which does not ensure TLB consistency, is used to satisfy TLB misses on cacheable references. This generic routine, which also forms the basis of the consistency-ensuring reload routines, is described in Section 7.5.3.3 and presented in Figure 19.

The AMD Am29000 instruction set is used in these routines. We believe the codes we use give a reasonable estimate of the number of instructions required to execute a basic TLB reload on a RISC processor.

7.5.3.1. TLB Entry

On a 32-bit RISC processor, like the Am29000, code similar to that illustrated in Figure 19 might be used to accomplish a TLB reload resulting from a TLB miss on a data access. This code is more easily understood with some knowledge of the structure of the Am29000 TLB and the assumed page table. Therefore, we first depict in Figure 18 the formats of words 0 and 1 of a TLB entry. The Task Identifier (TID) field of word 0 is not of interest to us since we assume a global (*i.e.*, system-wide) segmented virtual address space. (We include it, however, for completeness.) The first two words of a page table entry mirror the format of a TLB entry. (Again, we have a time vs. space tradeoff, in that to decrease the time required to reload a TLB entry, we may have to increase the space needed to store the page table.) Additional fields required to implement any one of our TLB consistency methods are considered to be in successive page table entry words. In these figures and the TLB reload codes that follow we assume a 1-Kbyte page size.

The Am29000's TLB is two-way set-associative with a total of 64 two-word entries, the format of which is pictured in Figure 18. Word 0 of TLB entry i ($i = 0, \ldots, 63$) is stored in TLB register number $2i$. Word 1 of TLB entry i is stored in TLB register number $2i + 1$. Line j ($j = 0, \ldots, 31$) consists of two TLB entries, j and $j + 32$. Virtual addresses are partitioned into three fields for the address-translation process. This partitioning depends on the page size. For a 1-Kbyte page size, bits 15 – 31 of the virtual address comprise the VTAG field of word 0 of a TLB entry; the VTAG is used for virtual address comparison between members of the same TLB line. Bits 10 – 14 select the appropriate TLB line and bits 0 – 9 are the page offset. The bit fields of words 0 and 1 of an Am29000 TLB entry are described below and are pictured in Figure 18.

Bit Fields of Am29000 TLB Entry Word 0:

Bits 31 – 15: Virtual Tag (VTAG) – When the TLB is searched for an address translation, the VTAG field of the TLB entry must match the most-significant 17, 16, 15, or 14 bits of the address being translated – for page sizes of 1, 2, 4, and 8 Kbytes respectively – for the search to be successful. The page size is stored in the MMU Configuration Register.

Bit 14: Valid Entry (V) – 1/0 indicates that the TLB entry is valid/invalid.

Bits 13 – 8: Protection/Access Information (PROT) – For protection and restriction of access.

Bits 7 – 0: Task Identifier (TID) – When the TLB is searched for an address translation, the TID must match the Process Identifier (PID) in the MMU Configuration Register for the translation to be successful. This field allows the TLB entry to be associated with a particular process. (Supervisor-mode accesses are made with PID = 0, no matter what value is in the MMU Configuration Register.) (Again, note that this field is not of interest to us since we assume a global (*i.e.*, system-wide) segmented virtual address space.)

Bit Fields of Am29000 TLB Entry Word 1:

Bits 31 – 10: Real Page Number (RPN) – Gives the most-significant 22, 21, 20, or 19 bits of the physical address of the page, for page sizes of 1, 2, 4, and 8 Kbytes, respectively. It is concatenated to bits 9 – 0, 10 – 0, 11 – 0, or 12 – 0 of the address being translated to form the physical address for the access.

Bits 7 – 6: User Programmable (PGM) – These bits are placed on chip outputs when the address is transmitted for an access. They have no predefined effect on the access: any effect is defined by logic external to the processor. They are typically used to control cacheability.

Bit 1: Usage (U) – Indicates which TLB entry in a given line was least recently used (LRU). 0/1 indicates that the TLB entry in Set 0/1 is the LRU TLB entry.

Bit 0: Input/Output (IO) – 1 indicates a reference to I/O space; 0 indicates a reference to other than the I/O space.

res: indicates that the field is reserved (must be zero).

On the Am29000 a TLB miss causes a trap, which in turn causes a program, called the TLB-reload routine, to execute. The TLB-reload routine is written to support the structure and access method of the page table contained in an external device or memory. When a TLB miss trap occurs, **sr**14, the LRU Recommendation Register (where **sr***i*, *i* = 1, ..., 23, denotes a special-purpose register) is written with the TLB register number of the TLB entry to be used by the TLB-reload routine. For instruction accesses, the Program Counter 1 Register, **sr**11, contains the instruction address which was not successfully translated. For

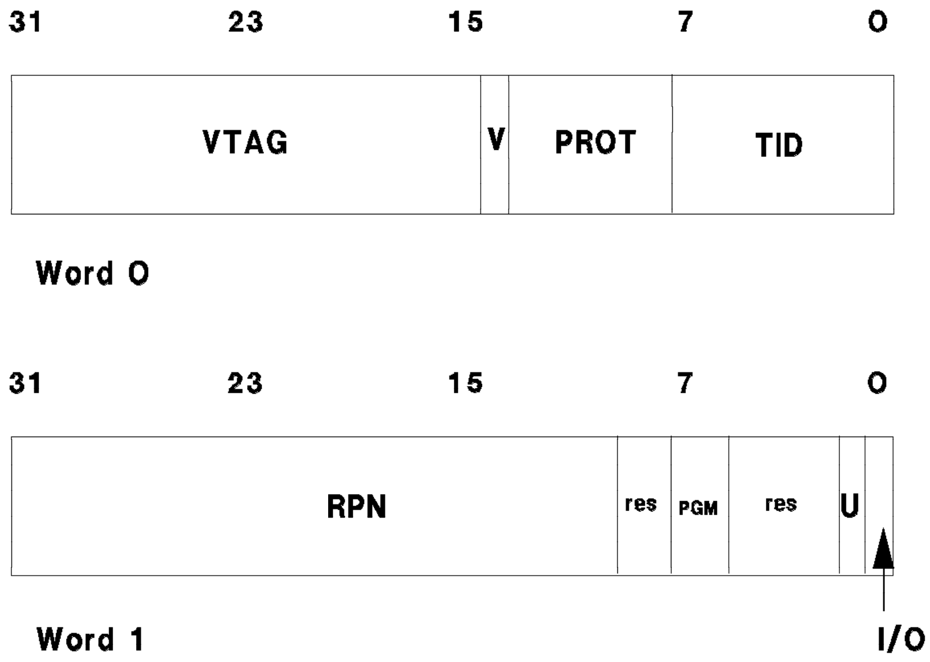


Figure 18. AM29000 TLB Entry Words 0 and 1

data accesses, the Channel Address Register, **sr**4, contains the data address which was not successfully translated. (See [AMD, 1987] for more details.)

7.5.3.2. Assumptions

For TLB reload codes of Figure 19 and Figure 21 – Figure 23 assume the following:

- the page size is 1 Kbytes;
- the first two words of TLB and page table entries are as pictured in Figure 18 and described above (except when reserved bits are used and extra words assumed);
- the code is stored in the cache of each processor (when TLBs are located at the memory modules (see Section 7.5.3.6) the code is stored at each memory module);
- a least-recently used (LRU) TLB replacement policy is adopted;
- the starting address of the page table is stored in global register Z, (where *i*, *i* = A, B, ..., Z, denotes a global register);

- translation is turned off during the execution of this code;
- the code is not interruptible;
- global registers used in this code are reserved for this purpose;[7]
- independent instructions (that do not generate network requests) can be executed during the delay imposed by a previous network request;
- the instruction at which the TLB miss occurred is restarted once the TLB reload is completed; and
- the instruction cache has a 100% hit rate.

Page management requires recording which TLB entries correspond to pages that have been modified. If hardware sets TLB-entry dirty bits on writes and the TLB reload code copies (sets) dirty bits to page table entries, we are dealing with dirty bits more often than necessary (as described below). Therefore, we assume software implementation of a TLB-entry dirty bit as done by AMD and MIPS, as well as others. Each of an Am29000's TLB entries has six bits that specify the permitted accesses for the corresponding page (two sets of read/write/execute bits, one each for supervisor and user modes); these bits are contained in the first word of the page table and TLB entries. We assume that a page table entry has three words, the third word containing, among other things, a dirty bit and the actual write permission. When a page is faulted into memory the write-access bits of the first (but not the third) word are cleared, along with the dirty bit. When a TLB entry is loaded, the TLB-reload routine copies the first two words of the page table entry into the TLB entry (initially the write-access bits are both zeroes). Thus, when the first legal write (after the page was faulted into memory) is attempted, a TLB protection violation trap occurs. The trap handler sets the dirty bit of the page table entry, the appropriate write-protection bit in the first words of the page table and TLB entries (to allow the access), and then returns to the trap routine. When the page is replaced, if the dirty bit is set, it is written to disk. The cost of the trap is only encountered on the initial writes to a page during each memory residency and is incurred at most once by each processor for each residency of a page. To reflect this cost in the service time for a TLB reload is inappropriate since the number of TLB reloads for a page (during each residency) are likely greater than N, especially for shared pages in a shared-memory multiprocessor. (Hardware dirty bits do, in fact, impose this cost on each TLB reload.)

Using this software implementation of a TLB-entry dirty bit, Validation may cause a page to be written to disk unnecessarily. Since the trap is executed before the translation is validated at the memory, the referenced page may be resident when the trap occurs (and will be marked dirty) but may be nonresident when the reference reaches the memory (and thus the write will not take place).

With these assumptions in hand, we present the TLB reload codes that are used as the basis estimating the service times for TLB misses on cacheable and noncacheable data. We make no assumptions as to the placement of page tables in memory and, thus, separately access each word of a page table. If the words of a page table entry were known to be located in consecutive words of the same memory module, then the codes could be improved.

[7] The Am29000 provides a large number of general-purpose registers with the intent that a subset of them will be reserved to store frequently-used data on-chip, where access time is fastest.

7.5.3.3. Generic TLB Reload Code and Miss Service Time

Since it may not be necessary to ensure the consistency of TLB entries for private and shared read-only pages, the following generic TLB reload code is used to estimate the service time for a TLB miss on cacheable data. This code also forms the basis for the TLB reload codes presented in Figure 21 – Figure 23, which are used to estimate the service times for TLB misses on noncacheable data when Validation, Read-locked TLBs, and Memory-based TLBs are used. Note that in Memory-based TLBs, the TLBs for noncacheable data are assumed to be located at the memory, while the TLBs for cacheable data are assumed to be at the processors. (We do this in order to compare just the overheads associated with maintaining the consistency of shared writeable pages.) In Figure 19 and Figure 21 – Figure 23, TLBE stands for TLB entry and PTE stands for page table entry. The code segment depicted in Figure 19 below assumes that Z stores the address of the page table and a page table entry consists of two words.

Using this code segment, we estimate 15 instructions for a TLB reload; this assumes that a page fault is not encountered and that the IRET instruction includes the cost of instruction restart and, thus, counts as two instructions. Only instructions (5) and (9) generate references that may cause a delay in instruction execution, *i.e.*, if the page table entry is not cache resident then a cache miss is generated. We assume that the page table entry is cached with a 97% probability. Thus, since three independent instructions (that do not generate memory requests) can be executed while either LOAD is in progress, in this case, the service time for a TLB miss is

$$15 + .03\times(2\times(2\times T_4)) - 6$$

or

$$9 + .03\times(2\times(2\times T_4)),$$

where the cache miss service time is $2\times T_4$. Remember that a cache line is 16 bytes and, thus, a cache read miss requires two double-word load network requests. If the page table entries were not cacheable, then the service time would be

$$9 + 2\times T_2.$$

However, to estimate the service times for TLB misses on noncacheable references, we must use TLB-reload routines that ensure TLB consistency. To produce consistency-ensuring code, a variable number of instructions and noncacheable data references, that depend on the employed TLB consistency method, are added to the generic code segment. In this way, we generate the TLB reload codes for Validation, Read-locked TLBs, and Memory-based TLBs presented in the next three subsections.

7.5.3.4. Validation TLB Reload Code and Miss Service Time

When Validation is adopted to ensure TLB consistency, each network request is augmented with the generation count (GC) of the related TLB entry. The NYU Ultracomputer project [Gottlieb, *et al.*, February 1983] proposes combining requests targeted to the same memory location when they meet at a switch in the network. Combining is triggered by the inspection of the first packet of a message. Therefore, if the address and GC are both in the first packet, then one may want to combine only requests with the same real address if the GCs are the same. If the real address and the GC are not in the same packet, requests can still be combined – the minimum GC would then be transmitted to memory in the combined request. Even in this case, the path between each processor/memory module pair may need

(1)	MFSR	A,**sr**4	get untranslated data address
(2)	SRL	A,A,10	A←virtual page no.
(3)	SLL	A,A,3	get displacement into page table
(4)	ADD	A,A,Z	A←address of PTE word 0
(5)	LOAD	0,0,B,A	B←PTE word 0
(6)	MFSR	C,**sr**14	C←LRU TLBE word 0 register no.
(7)	ADD	A,A,4	A←address of PTE word 1
(8)	ADD	D,C,1	D←LRU TLBE word 1 register no.
(9)	LOAD	0,0,A,A	A←PTE word 1
(10)	MTTLB	B,C	set LRU TLBE word 0
(11)	SLL	B,B,17	page fault?
(12)	JMPF	B,Fault	if resident bit off, service page fault
(13)	MTTLB	A,D	set LRU TLBE word 1
(14)	IRET		restart access that caused TLB miss trap

Figure 19. Generic TLB-Reload Routine to Satisfy a TLB Miss on a Data Access

to be widened. This is an engineering cost for Validation that we do not reflect in our simulation model.

To implement Validation a read-write lock can be associated with each page or with each frame (as described in Section 5.4.2). The simulator assumes that a read-write lock is associated with each frame and that the two-counter algorithm of Section 5.1.2 is used to protect the integrity of translation information. The TLB entry consists of three words but the third word, which stores the GC, is accessed from the page-frame table rather than the page table. A page-frame table entry is assumed to be two words, the GC field being in the first word. To allow the TLB reload code to be as succinct as possible, the format of the first two words of the TLB entry, and thus the page table entry, are modified so that the frame number (RPN), protection/access information (PROT), and valid bit (V) are all in TLB entry word 1. The new TLB entry format is pictured below in Figure 20.

To generate the TLB-reload routine for Validation, presented in Figure 21, the generic TLB reload code of Figure 19 is augmented by nine instructions. For comparison purposes, if there is a conflict in numbering between the codes of Figure 21 – Figure 23 and the generic code, we indicate in brackets next to the number of an instruction the number of the equivalent instruction of the generic code. In addition, when the same number is used for a different instruction, we place a "†" next to the number of that instruction.

The first step in implementing the readers portion of the two-counter algorithm (implemented by instructions 8 – 13.5) is to access the trial frame number and protection/access information from the page table entry (PTE), the GC (MyC1 = C_1) from the page frame table entry (PFTE), and the actual frame number and protection/access information from the page table entry. If an unsafe modification is in progress, the trial and actual frame numbers or the trial and actual protection/access information will disagree; this step is repeated until agreement is reached. We assume no conflicts of this type, thus, this type of contention is not modeled. The reader portion of the algorithm is completed when the memory request is validated by comparing the accessed GC (MyC1) to that of the GC table entry (MyC2 = C_2). The code segment of Figure 21 assumes that Z stores the address of the page table, Y stores the address of the page frame table, an entry in either table consists of two words.

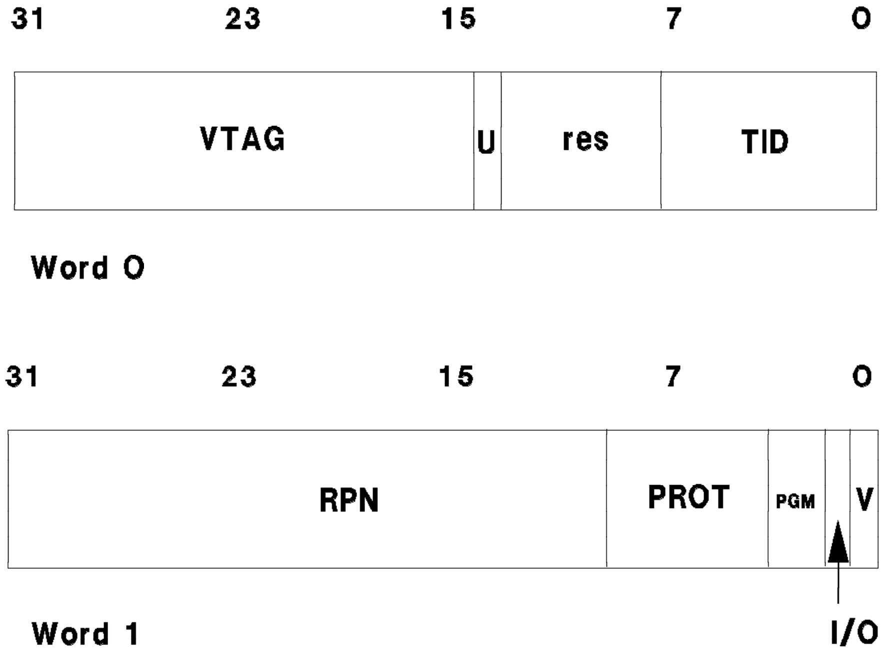


Figure 20. AM29000 TLB Entry Words 0 and 1 Revised

The nine additional instructions of this code generate at least two additional network requests. The delays introduced by the network requests generated by the LOAD instructions in this code can be diminished by executing independent instructions (that do not generate network requests) during the load delay slots. While each of the first three LOADs is in progress two independent instructions can be executed. Thus, adopting Validation, the service time for a TLB miss on a noncacheable reference is

$$24 + (4 \times T_2) - 6$$

or

$$18 + (4 \times T_2),$$

For Validation, a memory request is either satisfied or it fails because the translation is determined to be invalid. Failure is detected by the memory module's comparison of the transmitted GC and the GC of the corresponding entry in the GC table stored at the memory. When it fails, a signal is sent to the requesting processor to initiate the refreshing of the stale TLB information. In this case, since one network request is required to recognize a stale TLB entry, the TLB miss service time is

$$18 + (5 \times T_2),$$

To implement Validation, the memory-network interface must perform additional operations while processing each memory request, potentially increasing the memory access

(1)	MFSR	A,**sr**4	get untranslated data address
(2)	SRL	A,A,10	A←virtual page no.
(3)	SLL	A,A,3	get displacement into page table
(4)	ADD	A,A,Z	A←address of PTE word 0
(5)	LOAD	0,0,B,A	B←PTE word 0
(6)[7]	ADD	A,A,4	A←address of PTE word 1
(7)[6] Again:	MFSR	D,**sr**14	D←LRU TLBE word 0 register no.
(8)[9]	LOAD	0,0,C,A	C←PTE word 1 (trial frame no., PROT)
(9)[10]	MTTLB	D,B	set LRU TLBE word 0
(10)[8]	ADD	D,D,1	D←LRU TLBE word 1 register no.
(11)	SLL	B,C,30	page fault?
(12)	JMPF	B,Fault	if resident bit off, service page fault
(12.1)	SRL	B,C,9	get frame no.
(12.2)	SLL	B,B,3	get displacement into page frame table
(12.3)	ADD	B,B,Y	B←PFTE address
(12.4)	LOAD	0,0,B,B	B←GC (MyC1 = C_1) (PFTE)
(13)	MTTLB	D,C	set LRU TLBE word 1
(13.1)	ADD	D,D,1	D←LRU TLBE word 2 register no.
(13.2)	LOAD	0,0,E,A	E←PTE word 1 again
(13.3)	CPEQ	C,C,E	compare trial and actual PTE word 1
(13.4)	JMPF	C,Again	if read lock on frame not obtained, goto Again
(13.5)	MTTLB	D,B	LRU TLBE word 2 set with GC
(14)	IRET		restart access that caused TLB miss trap

Figure 21. TLB-Reload Routine for Validation

time. We assume that these operations are pipelined and, therefore, do not increase the memory access time. The simulator does not take into account that GCs may need to be refreshed (an infrequent event). As described in Section 6.2.3, the updating of GCs can be accomplished by a message broadcast to all memory modules in the referenced cluster. Either the network can issue multiple (k, the switch size) copies of the message at each level so that the message reaches a selected set of memory modules or the GC tables of a cluster can be connected via a bus so that a GC modification at one GC table results in a broadcast of the change to the other GC tables. (This bus, of course, limits scalability.)

7.5.3.5. Read-locked TLBs' TLB Reload Code and TLB Miss Service Time

As described in Section 5.2.2, if TLB consistency is ensured by Read-locked TLBs, a TLB reload is performed by obtaining a read lock on a page table entry and copying the translation information from the page table entry into a TLB entry. To implement the required locks without unnecessary serialization, we use the one-counter algorithm presented in Section 5.2.1.1 to protect each page table entry. The counter used by this algorithm is the replication count (RC), which represents the number of TLBs containing a copy of the page table entry. Only the readers portion of the algorithm is used in the TLB reload code; the writers portion is used when a process desires to modify a page table entry. To facilitate understanding the TLB reload code, below we list a high-level version of that portion of the TLB reload code used to obtain a read lock on a page table entry (which is based on the one-counter algorithm).

```
begin
   OK := false;
   repeat
      if RC ≥ 0 then
         if fetch-and-add(RC, 1) ≥ 0 then
            OK := true;
         else fetch-and-add(RC, −1);
   until OK;
```

A read lock is released by using a fetch-and-add operation to decrement the RC of the page table entry.

To generate the TLB-reload routine for Read-locked TLBs the generic TLB reload code of Figure 19 is augmented by instructions to release the read lock on the page table entry associated with the LRU TLB entry (if it is valid) and secure a read lock on the page table entry being loaded into the TLB. The code, presented in Figure 22, consists of two parts. Note that, for efficiency, these two parts are interleaved. Instructions 6.1, 6.2, 6.4, 6.6, 6.7, 6.9, 6.11 and 6.17 construct the address of word 2 of the page table entry associated with the LRU TLB entry and decrement the RC of this page table entry, completing the readers portion of the algorithm for the page table entry that is being replaced. The initial part of the readers portion of the one-counter algorithm for the page table entry being loaded into the TLB is implemented by instructions 4, 5, 6.3, 6.5, 6.8, 6.10, and 6.12 – 6.14. (The lock on the page table entry is released when this TLB entry is replaced.)

The fetch-and-add operation is used in the one-counter algorithm, as shown above, and, therefore, in the following TLB-reload routines. In the related codes, it has the form F&A(X,Y,Z), where X is the destination register, Y is a register containing the address of a memory location, and Z is a register containing the increment. The operation increments the contents of the referenced memory location by the contents of Z and loads the old contents of that memory location into X. Fetch-and-ϕ operations are implemented by macros that take two processor cycles and generate one network request. Since they require two machine instructions they cannot be placed in a delay slot.

To implement Read-locked TLBs, the page table entry is augmented by an RC field, the high-order bits of page table entry word 2. Words 0 and 1 of the TLB entry and page table entry are as pictured in Figure 19. The service time of a TLB reload assumes that the replacement of a TLB entry is required and that the LRU TLB entry is valid.
The code segment of Figure 22 assumes that Z stores the page table address + 8. In addition, it assumes that Y (X) stores a value that when added to a PTE word 2 causes an increment (decrement) of the RC.

In comparison to the generic TLB reload code of Figure 19, the static instruction count for this code segment is larger by 22 instructions that generate four network requests to modify two page table entries, the one being replaced and the one being loaded. (We count two instructions and one network request for each of the three fetch-and-add operations.) Since we assume no contention in accessing the read lock, instructions 6.13 – 6.15, counted as four instructions with one network request, are never executed. Delays introduced by the network requests can be diminished by executing nine independent instructions (that do not generate network requests) during the delay slots. Thus, the cost of a TLB reload is

$$37 - 4 + ((2 + 4 - 1) \times T_2) - 9$$

or

(1)	MFSR	A,**sr**4	get untranslated data address
(2)	SRL	A,A,10	A←virtual page no.
(3)	SLL	A,A,3	get displacement into page table
(4)†	ADD	A,A,Z	A←address of PTE word 2
(5)† Again:	LOAD	0,0,B,A	B←PTE word 2 (RC)
(6) Again1:	MFSR	D,**sr**14	D←LRU TLBE word 0 register no.
(6.1)	MFTLB	F,D	F←LRU TLBE word 0
(6.2)	SLL	G,F,17	G←valid LRU TLBE?
(6.3)	JMPF	B,Again	if RC<0, goto Again–deadlock avoidance
(6.4)	SRL	F,F,15	F←LRU TLBE VTAG (virtual addr bits 15–31)
(6.5)	F&A	E,A,Y	else (if RC ≥ 0) increment RC
(6.6)	SLL	F,F,6	F←prepare to construct LRU TLBE VPN
(6.7)	AND	H,D,62	H←TLB line select (bits 10–14 of VA)
(6.8)	OR	F,F,H	F←LRU TLBE VPN by ANDing TLB line select
(6.9)	JMPF	E,Dec	if RC < 0, decrement RC and goto Again1
(6.10)	SLL	F,F,2	F←offset into page table (to PTE word 2)
(6.11)	JMP	Cont	RC ≥ 0 therefore read lock obtained
(6.12)	ADD	F,F,Z	F←address of LRU TLBE PTE word 2
(6.13) Dec:	F&A	E,B,X	decrement PTE RC being loaded
(6.14)	JMP	Again1	read lock not obtained so goto Again1
(6.15)	LOAD	0,0,B,A	copy of (5)
(6.16) Cont:	JMPT	G,Inval	
(7)[4]	SUB	A,A,8	A←address of PTE word 0
(7.1)	F&A	F,F,X	decrement RC, release read lock on LRU TLBE PTE
(8)[5] Inval:	LOAD	0,0,B,A	B←PTE word 0
(9)[7]	ADD	A,A,4	A←address of PTE word 1
(10)[11]	SLL	C,B,17	page fault?
(11)[12]	JMPF	C,Fault	if resident bit off, service page fault
(12)[9]	LOAD	0,0,C,A	C←PTE word 1
(13)[10]	MTTLB	D,B	set LRU TLBE word 0
(14)[8]	ADD	D,D,1	D←LRU TLBE word 1 register no.
(15)[13]	MTTLB	D,C	set LRU TLBE word 1
(16)[14]	IRET		restart access that caused TLB miss trap

Figure 22. TLB-Reload Routine for Read-locked TLBs

$$24 + (5 \times T_2).$$

While the first LOAD (instruction 5) is in progress, instructions 6 – 6.2 can be executed; instructions 6.6, 6.7 and 6.8 can be executed while the instruction 6.5 (F&A) is being processed; instruction 9 in the delay slot of instruction 8 and lastly instructions 13 and 14 can be executed while instruction 12 is in progress.

Adopting Read-locked TLBs, the TLB entry does not require an algorithm-specific field so the number of instructions needed to access the TLB entry information is also unchanged, in comparison to the generic code, at two. Since the RC must be examined before increasing it, to prevent deadlock, additional instructions and an extra network request are required. In Section 5.3, we mention the fact that care must be taken to ensure that the TLBs of idle processors get flushed periodically. This is not enforced in our simulations.

7.5.3.6. Memory-based TLBs' Reload Code and Miss Service Time

In order to implement the TLB consistency method of Section 5.2.5, a TLB is located at each memory module, the general-purpose cache(s) of the processors are virtual-address caches, and a memory request contains the virtual, rather than physical, address. As mentioned in Chapter 5, since this method transmits the virtual address and combining messages is more difficult if the address is not contained in the first packet of the message, this method may require that the packet size be larger. Consequently, a system cost, *i.e.*, a wider path between the the processors and memory, may be inherent in the implementation of this method; we do not reflect this cost in the simulations.

To implement this method we assume that the memory-network interface of each memory module and the memory storing the cluster page table are connected via a cluster bus. We further assume that the memory storing the cluster page table is the same technology DRAM used for main memory, thus, two cycles are required to satisfy a request for one word of a page table entry. (High-speed static RAM could be used instead.) Each memory-network interface includes a TLB and control logic to process TLB misses and to service cluster bus requests. Since the TLBs are located at memory, we assume that the DRAM technique due to Sun Microsystems, Inc. (described in Section 5.2.4) is used in order to overlap translation with memory access so that memory access time is not increased on a TLB hit. In this case the TLB and page table entries are as pictured in Figure 19.

The cluster bus is used to broadcast TLB invalidations (resulting from page outs) and to transmit page table entry read and write requests (due to dirty bit handling and page-in/out synchronization). In addition, the atomicity of address translation and memory access is guaranteed, *i.e.*, a TLB will not respond to a bus request while a memory access is in progress.

A consequence of the location of the TLBs is that a TLB reload does not involve any network requests; it does, however, require bus communication among the TLBs of a cluster, access to the cluster page table, and coordination among the components comprising the network interface of each memory module. Note that if the TLB miss and page fault rates are low and the cost of a TLB reload is small, it is advantageous to permit the memory-network interface to block during TLB activity. If, instead, the interface were nonblocking then the memory service time (for all requests) would have to be increased by at least one cycle since in this case each interface would require a first-in, first-out queue fed by a two-input multiplexer that receives new memory requests from the network and recirculated memory requests from the interface's internal TLB-miss control logic. A TLB reload can block the memory-network interface for as long as seven cycles (without bus contention), as compared to two cycles for a TLB hit/memory access. With a large enough TLB, the frequency of a TLB reload is low, therefore, the infrequent penalty of a maximum of seven cycles is much more attractive than the constant penalty of one cycle.

When a new memory request is received at memory, the TLB is queried while the memory access is started. On a TLB hit, the memory request is satisfied in two cycles, the minimum allowed by the DRAM access time. On a TLB miss (recognized at the end of the first cycle), the processing of a TLB reload proceeds as follows: First, the referenced TLB attempts to become master of the cluster bus. Upon becoming bus master, a read request for word 0 of the needed page table entry is placed on the bus. If another TLB in the cluster contains an entry for the referenced page, the request is satisfied by that TLB. If no other

TLB in the cluster contains the entry, the page table entry is accessed. If the page is not resident (a page fault), a message is sent to the requesting processor and the bus is dropped. If the page is resident, a second bus request is generated for word 1 of the TLB/page table entry. Then a TLB entry is chosen for eviction and the relevant fields of the page table entry are stored in that TLB entry. At this point the bus is dropped. The bus, in effect the lock ensuring the consistency of the TLBs of a cluster, replaces the semaphore associated with each page table entry used, as in Validation and Read-locked TLBs.

Using an Am29000 for the control logic at the memory-network interface would be overkill, therefore, to estimate the cost for a TLB reload using this TLB consistency scheme, we count the number of cycles required to execute a TLB reload instead of counting instructions. Since the number of different kinds of requests that are transmitted on the cluster bus is very small, we assume that the formatting of the request is done during a cycle with other processing. Figure 23 describes the processing associated with each cycle. Note that the number of cycles required to execute a TLB reload (five or seven) depends upon whether or not another TLB already has an entry for the referenced page.

In accordance with the cycle count/description of Figure 23, the simulator estimates a service time of five cycles for a consistency-ensuring TLB reload if another TLB in the cluster has an entry for the referenced page; otherwise the service time is estimated at seven cycles. Our simulation model assumes that translation information for cacheable data is stored in separate TLBs at the processors and, thus, the offered load includes network requests generated by TLB misses on cacheable data. This is done to make the comparison between TLB consistency methods as fair as possible. Since we are comparing consistency-ensuring TLB service time, we are trying to keep all other parameters that may affect our results as similar as possible. For a cluster of constant size (*i.e.*, a constant number of memory modules) the cost of a TLB reload is independent of the system size (*i.e.*, the total number of processors and memory modules) provided that the increase in system size does not raise the TLB miss rate and/or page in/out rate to a point at which the cluster bus becomes overloaded. In contrast, as mentioned above, for Validation and Read-locked TLBs the cost of a TLB reload grows as the number of processors in the system increases.

The dirty bit handling described in in Section 7.1.1.3 is also adopted here. Thus, when a protection violation is generated by a TLB lookup, the memory-network interface sends the requesting processor a message that instructs the processor to execute the trap-handling code.

For Memory-based TLBs, the service time for a TLB reload greatly differs from that for either Validation or Read-locked TLBs. This is so because lock accesses are now replaced by the bus management described above, and noncacheable references to the cluster page table are now cluster bus requests that do not require any network transit.

Network requests are only generated by cache misses and noncacheable references; TLB misses on shared data references occur at memory and a service time is levied on each one as described above. Since TLB misses on shared data references do not result in additional network requests, the network load for Memory-based TLBs is smaller than for Validation or Read-locked TLBs. As a result, the service time for noncacheable references and cache misses decreases.

7.6. Service Times Associated with Page Faults and Page-Ins

As discussed in Chapter 6, we assume that only noncacheable references generate page faults and that a processor context switches only on page faults. The first processor to fault on a page has the responsibility of starting the I/O necessary to cause the page to become memory resident. The service time for starting I/O is assumed to be 1000 cycles. After starting I/O,

Cycle	Referenced TLB	Other TLB	Page Table Memory	Page Fault
1	recognize TLB miss			
2	read request PTE word 0			
3	load TLBE word 0	PTE word 0 transmitted		
4	read request PTE word 1 or load TLBE word 0		PTE word 0 transmitted	
5	load TLBE word 1 or read request PTE word 1	PTE word 1 transmitted		send page fault message
6			wait	
7	load TLBE word 1		PTE word 1 transmitted	

Figure 23. TLB Reload Cycles for Memory-based TLBs

the processor may context switch. Processors encountering page faults after a handler has been assigned to the page may also context switch. Context switching is considered to be done in user mode with a service time of 100 cycles.

The time required for a page to become memory resident depends on the number of required page transfers and the amount of I/O tasks queued for processing. As explained in Section 6.3.3.2, if the frame allocated to the faulted page stores a dirty page then, before reading in the faulted page, the dirty page must be written out. More I/O is required if the dirty page is referenced while it is being written out since, in this case, we allocate another frame to the faulted page and do not evict the referenced page.

Although we assume essentially an unbounded number of I/O devices, we limit the performance of the I/O subsystem by bounding the total I/O transfer rate. The page-in rate generated by the simulated system determines the effectiveness of the defined I/O subsystem. Less time is required to execute a program if a faster processor or a smaller network delay is assumed. A speedup may also result if the time required to reload a TLB, perform a context switch, or service a page-in is decreased. The critical factor in decreasing page-in service time is to decrease the seek time of the disk and/or increase the transfer rate. As the execution time of a program decreases, the same number of page-ins occur in a shorter period of time, increasing the demand for I/O processing power. Depending upon the locality of the program, this can result in a dramatic increase in I/O waiting time.

A typical secondary memory device, a 3600 rpm movable-arm disk, is associated with each paging arena (memory cluster). The time to transfer a page from secondary memory to main memory, T, is given by:

$$T = t_s + t_r + t_t,$$

where t_s is the seek time (the time required to reach the correct cylinder), t_r is the rotation time (the average time to reach the position being accessed on the cylinder), and t_t is the transfer time. On the average t_r will be one-half of the rotational latency, assuming a first-in first-out queue of random requests. A 3600 rpm movable-arm disk has a rotational latency (time for one rotation) of approximately 16.67 milliseconds, thus its average rotation time is approximately 8.33 ms.

Various techniques can be used to optimize paging I/O. For example, as described by Kuck [1978], systems may have multiple drives, each of which may seek simultaneously. (This overlap lowers the effective seek times.) Priorities can be established among a set of requests for I/O service. Software can be used to schedule disk accesses and a disk cache can be utilized. Assuming some optimization of paging I/O, we assume an average seek time of 15 ms which we believe is a reasonable estimate[8] but is considered high for IBM 3380 disks (for which a mean seek time of 7.2 ms is used by Kim [1986]); and we assume a transfer rate of 2 Mbytes/second or 2 Kbytes/ms. (For comparison, the transfer rate for the IBM 3380 disks is 3 Mbytes/second according to Kim [1986]). Thus, the time to read and transfer a page of 2 Kbytes from secondary memory to main memory, T, is approximately 24 ms and hence in one second the I/O for approximately 41.5 page faults can be serviced. In terms of simulation time, the time to read and transfer a page of 2 Kbytes depends on the processor cycle time. Assuming an idealized 10 MIPS RISC processor, the time to read and transfer the page would be 240,000 cycles.

7.7. Reconstituting a S/370 Interval and Mapping It to a RISC Interval

Since the only information in our traces regarding cacheable references is the number that occur in each event, to simulate a multiprocessor system we must reconstitute a IBM System 370 (S/370) interval before it can be processed. To do this, we need to estimate the number of references per instruction for the S/370 traces and the ratio of loads to stores. Since we wish to model a system with RISC processing elements using S/370 traces, we also need to map each S/370 interval to an equivalent interval on such a processor. To do this, we need to know how the number of instruction and data references made by a RISC machine compares with that of an IBM System/370. Given this information and the instruction and data cache hit rates and the TLB hit rate for the user-cacheable partition, the cache and TLB misses associated with the event can be generated and the service time for the interval can be computed.

7.7.1. Ratio of Instruction References to Data References

For the applications we have traced using an IBM System/370 (a CISC, complex instruction-set computer) the average number of references per instruction is about 1.7, as shown below in Table 4. These figures represent statistics for a minimal run of each program, *i.e.*, the initialization and termination code plus one iteration of the main computational loop. Section 7.9 describes the programs used to generate these traces.

The S/370 fetches double words and, thus, more than one instruction or operand may be fetched per memory reference. However, we have assumed that each instruction reference generates an instruction fetch and each data reference generates a data fetch. Since the applications studied are scientific/mathematical applications that make extensive use of double-precision (*i.e.*, 64-bit) floating-point variables, we effectively treat all data fetches as being double-word fetches.

---

[8] Smith [Aug. 1985] notes that current disks, without disk caches, are accessible on average in 20-30 ms. A 10-inch Fujitsu Eagle has an average seek time of 18 ms.

Application	Peskin	Simple
S/370 Instruction Reference Count	232964686	128448222
S/370 Data Reference Count	133287229	98180896
S/370 Total Reference Count	366251915	226629118
Average Number of References Per Instruction	≈ 1.57	≈ 1.76

Table 4. S/370 Reference Counts: Peskin and Simple Traces

Since we are using S/370 traces to drive a simulation model of a multiprocessor comprised of RISC processing elements, we must map an S/370 event to an equivalent event on an idealized RISC processor. In other words, we must map the S/370 ratio of instruction references to data references, 1:0.7, to an equivalent ratio on an idealized RISC processor, and estimate the ratio of data loads to data stores.

Davidson and Vaughan [1987] showed that even with large caches, a machine with a simple instruction set can expect to generate twice as many main memory read requests as a machine with a complex instruction set. Since a RISC machine requires more instructions to implement a given function, to achieve good performance RISC architectures have fast clock cycles, instruction caches (to reduce processor-memory instruction traffic), and often have multiple register sets (to reduce processor-memory data traffic). Having multiple register sets allows registers to be allocated to subroutines by means of a concept known as *register windows*. As explained by Stone [1987], each procedure sees a region of registers, and that region overlaps partially with the window seen by the calling procedure and by procedures that it calls. The windows dynamically change as procedures are entered and exited and provide a means for passing parameters and receiving results in the registers without having to save and reload registers.

The effect of the increase in memory traffic due to more instruction references and the offsetting effects of the decrease in processor-memory data traffic and the number of loads and stores due to multiple register sets is to be reflected in our mapping of a S/370 event to an event of a hypothetical RISC processor. But, how many more instructions are executed by a modern RISC machine and how does the amount of data referencing memory traffic compare?

Several papers have compared the storage requirements for comparable programs on a CISC processor versus a RISC processor [Hill, *et al.*, 1986; Patterson and Sequin, 1981; Chow and Horowitz, 1987; Davidson and Vaughan, 1987; and Colwell, *et al.*, 1985]. Given the program storage requirement or, better yet, the instruction traffic requirement for a processor [*cf.* Davidson and Vaughan, 1987], does not directly allow us to compare the number of instructions executed. A requirement of 50% more instruction bytes, does not equate to 50% more instructions executed. In general, RISC architectures have a fixed 32-bit instruction size and format, while CISC machines have an instruction set of multiple sizes, formats, and addressing modes.

Flynn, Mitchell, and Mulder [1987] measured instruction set tradeoffs using a computer architecture simulation platform developed at Stanford. In this paper they base their tradeoff measurements on memory traffic. Using Pascal programs of medium size as benchmarks, several architectures were compared, two of which are of particular interest to us, the Fix32 and the OBI360 (each with 32-bit data paths). The Fix32 is a RISC-type architecture and the OBI360 is generally similar to the IBM System/360 with the storage-to-storage instruction format excluded. Without a cache 50% more instruction bytes were required to execute a program by the Fix32 than were required by the OBI360. This figure rises to 100% for caches of size 8 Kbytes and 16 Kbytes, indicating that caches of this size allow the OBI360 to capture its working set while the Fix32 has not yet done so. RISC instructions are 32 bits wide while the IBM System/360 has variable size instructions (from 16 bits to 48 bits, although the OBI360 excludes the 48-bit storage-to-storage instructions). Flynn, Mitchell, and Mulder [1987] show that the Fix32-RX, the Fix32 with the RX format (a 32-bit wide instruction that allows one instruction operand to reside in memory) added, requires 10 percent fewer instructions than the Fix32. Flynn [1989] also reported that their benchmarks of optimized Fortran code yield 11.4% more instructions on a RISC processor (HP Precision, similar to the IBM 801) than on the IBM System/370. (Given this information, we see that on average a RISC instruction is approximately 1.33 the size of a S/370 instruction.)

Eickemeyer and Patel [1987] evaluated the performance of multiple register sets by measuring the performance of the VAX 11/780 with multiple register sets of various sizes. Measurements were made by tracing the execution of ten programs, each run on small input files. (The execution time, however, was sufficiently long to reduce the effects of initialization in the programs.) After the execution trace of a program was made, the trace was mapped into the trace that would have resulted from tracing a VAX with multiple register sets; and then this new trace was evaluated. The observation of interest here is that when using the multiple register-set architecture the change in the number of data references shows a large decrease, with changes ranging from 21% to 43%, mostly due to the reduction in local and parameter references and in register saving. They claim, on average, a decrease in memory references of approximately 35% can be expected. In addition, when using the multiple register set architectures they note a small positive change in the number of instructions for all programs studied. Changes (both negative and positive) in the number of instructions result from overflow and underflow as well as from the mapping to the multiple register sets. Flynn, Mitchell, and Mulder [1987] also studied the effect that large register sets have on processor performance. They found that, ignoring expression evaluation, data traffic can be reduced on either the Fix32 or the OBI360 (described above) by increasing the number of registers and/or using global register allocation. This reduction can be as much as 35%. Considering all the data traffic, a Fix32 with multiple register sets (128 registers in all) compared to the OBI360 with 16 or 32 registers shows a decrease of over 40% in data traffic.

Using the data of Flynn [1989] regarding the comparison of the dynamic instruction counts of a RISC processor vs. the IBM S/370 and reflecting the results of Eickemeyer and Patel [1987] and Flynn, Mitchell, and Mulder [1987], we map an S/370 event to an event on an idealized RISC processor by increasing the number of instruction references by 11.4% and by decreasing the number of cacheable data references by 35%. Recall that we are mapping a CISC machine without multiple register sets to a RISC machine with multiple register sets. (As shown by Eickemeyer and Patel and Flynn, Mitchell, and Mulder, multiple register sets on a CISC machine would also result in a performance advantage.)

The comparisons cited above are neither based on RISC processors with floating-point units nor floating-point intensive codes. The relative size of such a program run a RISC processor without a floating-point unit will be larger than that run on a RISC processor with a floating-point unit. It is likely that floating-point instructions will be included in RISC instruction sets. But, running such a program on a RISC processor with a floating-point unit most likely compromises our one-instruction-per-cycle assumption.

Since estimating such numbers for a hypothetical RISC processor is like shooting at a moving target, we feel that these approximations are as good as we can get and believe they are close enough to allow us to compare the performance of methods for ensuring TLB consistency on multiprocessor systems. As an example, a S/370 event of length 27, having 1.7 references per instruction (and hence $\approx$ 15.3 instructions and $\approx$ 11.7 data references), becomes a RISC event of length 25, where ≈ 17.1 of the references are for instructions, ≈ 6.9 for cacheable data, and 1 for noncacheable data. This gives us a 2.16:1 ratio of instruction references to data references, *i.e.*, ≈ 1.46 references per instruction.

This instruction to data reference ratio seems reasonable. In comparing the IBM S/370 traces (55% instructions, excluding the Cobol traces) and those of the CDC 6400 (77% instructions), which has a RISC-like instruction set, Alan Jay Smith [June 1985] states that the ratio of instructions to data loads and stores will range from about 1:1 for relatively complex (32-bit) architectures up to about 3:1 for extremely simplified architectures, assuming a standard (single) register set. It should be evident that with simplified processor architectures instruction sequences will be longer and, thus, the ratio of instructions to data higher. For example, Dhrystones 1.1 run on an Am29000 simulator shows 79% of all references are instruction fetches, a ratio of instructions to data of about 4 to 1.

Since the performance of the instruction cache is not necessarily the same as that of the data cache, information about the ratio of instruction references to data references is necessary in order to assign a service time to an interval. In addition, since the service times for loads and stores are not necessarily the same, we must also estimate the number of data loads and stores in an interval.

7.7.2. *Ratio of Data Loads to Data Stores*

Smith's [June 1985] trace-driven simulations are used to measure cache performance with 49 traces from six machine architectures (the IBM 370e, the IBM 360/91, the DEC VAX, the Zilog Z8000, the CDC 6400, and the Motorola 68000), derived from seven programming languages (FORTRAN, 370 Assembler, APL, C, LISP, AlgolW, COBOL). For all of the architectures studied, Smith contends that it seems satisfactory to say that data reads (on the average) outnumber data writes by about 2 to 1. Both the number of registers (and the number of sets of registers) and the degree of register use optimization in the compiler affect the frequencies of loads and stores, therefore, Smith feels his data are insufficient to make a definitive statement about the ratio of reads to writes in a RISC machine. Eggers and Katz [1988] and Agarwal and Gupta [1988] show an overall read to write ratio of about 3:1 for the parallel applications studied; they include both private and shared data. Dhrystones 1.1 run on an Am29000 simulator shows that approximately 43% of data accesses are stores. Gross, *et al.* [1988] measure and evaluate the MIPS architecture; for the unoptimized programs that they studied, loads are about three times as frequent as stores; for the optimized codes studied loads are only 1.7 times as frequent. This decrease in frequency occurs both because register allocation is slightly more successful at removing loads than stores, and because of other global optimizations that remove loads but do not affect stores (*e.g.*, common subexpression elimination of an expression). Considering these findings, we estimate the ratio of loads to stores to be about 3:1;thus, 75% of the cacheable references are loads; 25% stores.

Given the ratio of instruction references to data references, the mapping of a S/370 interval to a RISC interval, and the ratio of data loads to data stores, all we need are the hit rates of the instruction cache, data cache, and user-cacheable partition of the TLB in order to assign a service time to an interval.

7.7.3. *Cache Hit Rates*

Due to the high-bandwidth, high-latency nature of the interconnection network, high cache hit rates are extremely important in order to achieve acceptable performance; therefore, we assume the caches are large enough to deliver hit rates (on cacheable references) experienced on traditional machines with large caches. We use the performance statistics reported by

Smith [June 1985] for large-size caches with 16-byte lines to arrive at our assumed hit rates. For a cache size of 65 Kbytes with a 16-byte line size Smith suggests design target miss ratios of 2% and 3% for instruction and data caches, respectively. Since high cache hit rates are imperative for good performance on a multiprocessor system, we assume larger caches, a 1% miss rate for the instruction cache, and a 3% miss rate for the data cache.

The mapping of the interval of the sample S/370 event described above in Section 7.7.1, resulted in a RISC event of length 25 with 17.1 instruction references, 6.9 cacheable data references, and 1 noncacheable data reference. Since we assume a 3 to 1 ratio of loads to stores for cacheable data, the 6.9 cacheable data references are comprised of 5.175 loads and 1.725 stores. Thus, the sample event generates .171 instruction cache read misses and ≈ .155 data cache read misses and ≈ .052 data cache write misses. In addition, the event generates .24 TLB misses on user-cacheable data.

As was explained in Section 7.3.1, cache read misses and TLB misses delay the processor, while cache write misses do not. But, all cache and TLB misses contribute to the offered load and, thus, the average network waiting time, which is the last bit of information we need before we can compute the service times of cache and TLB misses and references to noncacheable data.

7.8. Network Waiting Time

As mentioned above, the only information we are lacking in order to assign a service time to an event is the network waiting time experienced by network requests generated by cache and TLB misses and references to noncacheable data. The network waiting time is the sum of the queueing delays at the switches in the network. Except in the case of Memory-based TLBs, we do not model contention at the memory.

Formulas derived by Kruskal and Snir [1983] have been used to study the performance of both the NYU Ultracomputer and the IBM RP3. We use a formula derived in their later work with Weiss [Kruskal, Snir, and Weiss, 1986], which extends the work of Kruskal and Snir [1983], to estimate the network waiting time. In this section the terms network request and message are synonymous.

Kruskal, Snir, and Weiss [1986] analyze the delay experienced by a network request in a buffered, multistage, packet-switched banyan network. This analysis leads to good approximations for the total delay of a message through the entire network. In particular, for messages of different sizes, the total waiting time for an n stage network is approximated by:

$$\frac{m\rho \sum_{i=1}^{n} m_i (m_i - \frac{1}{k}) g_i}{2(1 - m\rho) \sum_{i=1}^{n} m_i g_i} + (n-1)(1 + \frac{4m\rho}{5k}) \frac{(1 - \frac{1}{k}) m\rho \sum_{i=1}^{n} (m_i - \frac{1}{k}) g_i}{2(1 - m\rho)(m - \frac{1}{k})}, \quad (7-1)$$

where m is the average number of packets per message, m_i, $i = 1, \ldots, n$, are the number of packets in each type of message each of which occur with probability g_i, k is the switch size, *e.g.*, for a $2x2$ switch, $k = 2$, and ρ $(0 \le \rho \le \frac{1}{m})$ is the traffic intensity or the number of messages per processor cycle, also called the request rate. This expression approximates the average total waiting time as the average waiting time from the first stage plus $(n-1)$ times the waiting time at the later stages. The delay time is given in switch cycles.

For the simulations, we use

$$k = 2$$

$$n = \log_2 N,$$

The average number of packets per message, m, is calculated at the beginning of a simulation. Its value depends on the average interval size of the traced program and the mix of instructions and data, all of which are input parameters (defined in Section 7.5).

The formula of Kruskal, Snir, and Weiss [1986] was validated via simulations only for the forward (processor-to-memory) direction. As can be surmised from the above formula, the network waiting time grows with ρ.

Liu [1990] indicates that the waiting time in the return direction is not as large as the waiting time in the forward direction. Liu uses an independent reference model to generate network requests. His early simulations indicate that for a network depth of greater than two and a network load of less than .8 the following expression approximates the average total waiting time in the memory-to-processor direction.

$$.45\lambda + .6\lambda + (n-2)\times.7\lambda, \qquad (7-2)$$

where λ is the waiting time at stages other than the first in the processor-to-memory direction. The greater the load, the greater λ. The first term, .45 λ, is a close approximation to the network queueing delay of the stage of the network nearest to the memory and to the waiting time experienced in a one-stage network in the memory-to-processor direction. The second term, .6 λ approximates the delay of the second nearest stage to the memory. The last term represents the waiting time at all other stages in the return path, each stage having a queueing delay of .7 λ.

We use Equations (7-1) and (7-2) above in our simulations to estimate network transit time. Shown more explicitly, the waiting time in the memory-to-processor direction is approximated by:

$$(.7n - .35\lambda) + (1 + \frac{4m\rho}{5k})\,\frac{(1-\frac{1}{k})m\rho\sum_{i=1}^{n} m_i(m_i - \frac{1}{k})g_i}{2(1-m\rho)(m-\frac{1}{k})}$$

Since the average length of a network response can be different from the average length of its corresponding network request, m, m_i, and g_i, $i = 1, \ldots, n$ are not necessarily the same as in the expression shown above for the waiting time in the processor-to-memory direction. Thus, the network waiting time is the sum of the waiting time in the forward direction plus the waiting time in the backward direction.

Liu's later simulations [1990] show that the average total waiting time in the memory-to-processor direction is actually

$$.25\lambda + .5\lambda + .65\lambda + (n-3)\times.7\lambda, \qquad (7-3)$$

where λ is the waiting time at the last stage in the processor-to-memory direction. The first term, .25 λ, is a close approximation to the network queueing delay of the stage of the network nearest to the memory and to the waiting time experienced in a one-stage network in the memory-to-processor direction. The second term, .5 λ approximates the delay of the second nearest stage to the memory. The third term, .65 λ approximates the delay of the third nearest stage to the memory. The last term represents the waiting time at all other stages in the return path, each stage having a queueing delay of .7 λ. As can be seen, for larger networks both Equations (7-2) and (7-3) are $O(.7\lambda N)$.

In order to compute an average waiting time, we must estimate ρ, the offered network load, which describes the rate at which network traffic is generated.

7.8.1. Offered Network Load

Given N and k, the average event length, the estimated mix of instructions and data, the assumed percentages of loads and stores, a specific TLB consistency scheme, the instruction cache, data cache, and TLB hit rates, we estimate the number of network requests generated by an average size event and, thus, approximate an average ρ. This value of ρ remains constant during an entire simulation and is used to compute the service times for cache and TLB misses and references to noncacheable data.

All the data needed to estimate ρ has been presented in this chapter, with the exception of the average event lengths exhibited by the programs used to generate the input traces. For the main computational section of each of the applications that we traced, we computed the average interval length for the IBM System/370, shown below in Table 5.

Application	Peskin	Simple
Noncacheable Reference Count	14530323	17269861
Cacheable Reference Count	365603943	128392502
Average Interval Length	≈ 26	≈ 8

Table 5. Event Characteristics for Peskin and Simple Traces

As an example, we compute ρ assuming that an average event is of length 27 (an average interval is of length 26), TLBs are located at the processors, Read-locked TLBs is used to ensure the consistency of the user-noncacheable partitions of the TLBs, and the generic code of Figure 19 is used to satisfy TLB misses on cacheable data.

The simulator computes the contribution to ρ of user-noncacheable TLB misses according to the TLB consistency scheme being modeled. When TLBs are at the memory-side of the network, the cost of a generic TLB reload is the same as that of a consistency-ensuring TLB reload. Also, an Am29000 processor is not used to implement Memory-based TLBs; TLB reload code is replaced by control logic. Here, as in the proposed implementations of Validation and Read-locked TLBs, we assume that TLB reloads are executed by an Am29000 and the generic and consistency-ensuring TLB reloads take different amounts of time to execute.

In accordance with the previous sections of this chapter, we assume that:

- an average of 1.7 references are made for each S/370 instruction,
- the RISC to CISC instruction count ratio is 1.114:1,
- the reduction in data references due to multiple register sets is 35%,
- 75% of data references are loads,
- the instruction cache hit rate is 99%,
- the data cache hit rate is 97%, and
- the hit rate of the user-cacheable partitions of TLBs is 99%.

In spite of the fact that one of the results of a simulation is the average hit rate for the user-noncacheable partitions of the TLBs, in order to compute the offered load, we must assume a hypothetical hit rate. We use the same hit rate as is assumed for the user-cacheable partitions, 99%. Using these assumed values, the analysis given above shows that the 27 references comprising an average size S/370 event are composed of 15.3 instruction references, 10.7 cacheable data references, and 1 noncacheable data reference. The equivalent RISC event has 25 references comprised of 17.1 instruction references, 6.9 cacheable data references, and 1 reference to noncacheable data. This RISC interval generates .171 instruction cache misses and .207 data cache misses. Of the 6.9 cacheable data references, 5.2 are loads and 1.7 are stores; of the .207 data cache misses, .155 are read misses. The (instruction and data) cache read misses $(.171 + .155)$ generate $.326 \times 2 = .652$ double-word load requests (assuming that one message is required to access two words and that the line size is two double words, as are the cases in the Ultra III prototype) to be submitted to the network, in addition to the message required to access the noncacheable datum and the 1.7 double-word store requests. We assume that memory is interleaved on a double-word basis. A request for two words is comprised of two packets and the response is comprised of four packets.

Assuming 100% cache and TLB hit rates for TLB reload code segments, the sample RISC event generates .25 TLB misses, .24 for cacheable page table entries and .01 for noncacheable page table entries. Since the generic TLB reload code consists of 9 instructions that generate two memory requests, the .24 TLB misses increase the event length by $.24 \times 9 = 2.16$ instructions which generate $.24 \times 2 = .48$ memory requests. These memory requests (generated by the TLB reload code for page table entries that are cacheable) result in $(.24 \times 2) \times .03 \approx .014$ data cache misses. These cache read misses generate $.014 \times 2 = .028$ double-word load network requests. Assuming that consistency-ensuring TLB reloads are performed by following the Read-locked TLBs algorithm, the .01 TLB misses on noncacheable references generate $.01 \times 24 = .24$ instruction requests and $.01 \times 5 = .05$ single-word network requests for data (see Section 7.1.3.2). Thus, our sample event is comprised of $17.1 + 2.16 + .24 = 19.5$ instructions that generate ≈ 3.43 network requests $(.652 + .028$ network requests to satisfy cache read misses, 1 to access the noncacheable datum, 1.7 stores generated by the event, and .05 network requests generated by the .01 TLB misses on noncacheable data).

To calculate the request rate, ρ, we must compute the number of cycles in which these 3.43 messages are generated. As mentioned above, references to cached instructions and data and noncacheable data stores are effectively satisfied during the instruction cycle but the processor is delayed by instruction cache misses, TLB misses, and data cache read misses, as well as the noncacheable reference if it is a load. In calculating ρ, we assume the noncacheable reference is a load 75% of the time. Thus, the processor is delayed by only 1.48 messages $(.652 + .028 + .75 + .05)$, 1.43 double-word load requests and .05 single-word load requests, of the 3.43 messages. (The offered load is determined by the total number of messages generated during the event, in this case, 3.43).

Waiting for responses to memory requests transmitted through the network causes the processor to lower the offered load to the network. Lowering the load decreases the latency and allows the load to increase again. Assuming that the offered load and latency reach a steady state, a "fixed point" (*cf.*, [Norton and Pfister, 1985)]) can be reached where the expected latency yields a processor utilization consistent with the request rate. Using this concept, we compute the number of cycles required to execute these 19.5 instructions in the following way.

First we compute the average round-trip delay for a network request using the formulas of Kruskal, Snir, and Weiss [1986] and Liu [1990] with

$$\rho = \frac{\text{number of network requests}}{\text{time}} \quad ,$$

where time represents the number of cycles in which these requests were generated. For our application, the numerator is 3.43 and the denominator is 19.5, which is the the number of instructions executed in the interval plus the cost associated with the network requests due to double-word and single-word loads. Initially, the network waiting time is considered to be zero. We assume that data loads and stores are executed on double words, a load of a noncacheable page table entry (as a result of a TLB reload) is a single-word load and, as mentioned above, a cache fill generates two double-word loads. Assuming no network queueing delays and $N = 64$ (the network depth is, therefore, 6), Each of the cache read misses $(.326 + .014)$ generate two network double-word load requests and the .75 loads of noncacheable data generate .75 double-word load requests. The response to each double-word load request is four packets long. Thus, two more cycles are added to the network transit time $(15 + 2)$. $\rho = 3.43/(19.5 + (1.43\times17) + (.05\times15)) \approx .078$. We iterate this process, using a new ρ based on the computed latency until a fixed point is reached, where the expected latency produces a new value of ρ equal to the old value of ρ. The initial ρ is .078 (as shown above) and a fixed point is reached with $\rho \approx .071$. A network load of .071 results in a round-trip network waiting time of 3 cycles; the average service time for a double-word load network request is, therefore, approximately 20 cycles. Thus, the 3.43 messages are generated in $19.5 + (1.43\times20) + (.05\times18) = 49$ cycles, giving a load of $\frac{3.43}{49} \approx .071$.

The IBM RP3 project estimates a 1:10:15 ratio for delays in their 512-processor architecture, *i.e.*, one processor cycle time delay for a cache hit, 10 for a local or home memory reference, and 15 for a remote memory reference (see [Pfister, *et al.*,1985] for further details). The major part of these delays are due to interfacing, the network itself is relatively fast; thus, their estimate is expected to grow minimally for larger configurations. For 64 processors, the value we use (20) is higher than the RP3 estimate, however, we assume that most of the memory references generate double-word accesses.

Our estimate of ρ, the offered load, does not take into account the requests generated as a result of page faults or other operating system functions, such as clock interrupts.

7.8.2. Comparison of Offered Loads and TLB Miss Service Times

Adopting Read-locked TLBs and given $N = 64$, when a fixed point is reached $\rho = .071$ and the average network waiting time is 3 cycles. Thus, the average service time for a double-word load is 20 cycles; for a single-word load or store, it is 18 cycles. In this case, the generic TLB reload code has an average service time of

$$9 + .03\times(2\times(2\times20)) = 11.4 \text{ cycles.}$$

This is because the page table entries are cached with a probability of 97% and we assume that the two references cause two separate cache misses. Recall that the cache miss service time is $2\times T_2$ which in this case is equal to 2×20. When TLB consistency is to be ensured, the words of the referenced page table entry (or entries) are not cacheable and are accessed by generating single-word loads. Thus, the TLB miss service time assuming the Read-locked TLBs reload code is

$$24 + 5\times18 = 114 \text{ cycles.}$$

In comparison, using Validation instead of Read-locked TLBs but otherwise making the same assumptions and assuming the same input parameters, the corresponding fixed point occurs when $\rho \approx .065$ and the average network waiting time is 2.7 cycles. In this case the generic TLB miss service time is

$$9 + .03\times(2\times(2\times19.7)) = 11.364 \text{ cycles}$$

and the consistency-ensuring TLB miss service time is either

$$18 + 5\times17.7 = 106.5 \text{ cycles.}$$

or

$$18 + 4\times17.7 = 88.8 \text{ cycles}$$

depending on whether or not a stale TLB entry was used to access memory.

Similarly if we adopt Memory-based TLBs instead, when a fixed point is reached, $\rho\approx.066$ and the average network waiting time is $\approx$ 2.7 cycles. In this case, the generic TLB miss service time is the same as that for Validation since TLBs for cacheable data are assumed to be located at the processors. In contrast, since TLBs for noncacheable data are located at the memory and, thus, no network requests are generated by the TLB reload code, the consistency-ensuring TLB miss service time is either 5 or 7 cycles depending on whether or not the referenced words are resident in other TLBs.

To indicate the effects of adopting one TLB consistency method instead of another, we compare the respective service times for the sample event used to compute the offered load in Section 7.8.1. As shown in Section 7.8.1, assuming $N = 64$ and adopting Read-locked TLBs, $\rho = .071$ and the average network waiting time is 3 cycles. Thus, the service time for the sample event of 17.1 instructions, 6.9 cacheable data references, and 1 noncacheable reference is

$$19.5 + (1.43\times20) + (.05\times18) = 49 \text{ cycles.}$$

The instruction count of the sample event is increased from 17.1 by 2.16 instructions for TLB misses on cacheable data and .24 instructions for TLB misses on noncacheable data. The 1.43 double-word load requests are comprised of .652 requests generated by references in the sample interval, itself, to satisfy cache read misses, .028 requests to satisfy TLB misses, and .75 requests to satisfy the noncacheable reference (which is estimated to be a load 75% of the time). The .05 single-word load requests are generated by the Read-locked TLBs reload code which generates 5 network requests and is executed 1% of the time. In this case, the 19.5 instructions generate 1.43 double-word load requests and .05 single-word load requests.

If Validation is adopted and $N = 64$, $\rho\approx.065$ and the average network waiting time is $\approx$ 2.7 cycles. Then, the service time for the sample event is

$$19.44 + (1.43\times19.7) + (.3\times17.7)\approx52.92 \text{ cycles.}$$

In this case, the instruction count of the sample event is increased from 17.1 by the 2.16 instructions for TLB misses on cacheable data and .18 instructions for TLB misses on noncacheable data. In comparison to Read-locked TLBs, the number of single-word network requests increases by .25 since a store of noncacheable data delays the processor. If a stale TLB entry is not used to access memory then the number of single-word network requests drops to .29.

If Memory-based TLBs is adopted and $N = 64$, $\rho\approx.066$ and the average network waiting time is $\approx$ 2.7 cycles. Then, the service time for the sample event is

$$19.32 + (1.43\times19.7) + (.25\times17.7)\approx51.92 \text{ cycles.}$$

In this case, TLB misses on cacheable data generate network requests but TLB misses on noncacheable data do not. Since the service time for TLB misses on noncacheable data is at best 5 cycles and at worst 7 cycles, we use the average TLB reload cost of 6 cycles. In comparison to Read-locked TLBs, the difference in the instruction count (19.5 vs. 19.32) is that TLB misses on noncacheable data now generate only .06 cycles rather than .24. As with Validation, the event's .25 stores of noncacheable data delay the processor. But, since network requests are not generated by TLB misses, the number of single-word network requests is .25 rather than .30.

Table 6, presented below, summarizes the comparisons made in this chapter with respect to our three TLB consistency methods. These comparisons assume that $N = 64$. TLB miss services times and sample event service times are given in cycles. The worst case TLB miss service times are given for Validation and Memory-based TLBs.

TLB Consistency Method	Network Load (ρ)	TLB Miss Service Time	Sample Event Service Time Instruction Count + Double-Word Load Requests + Single-Word Load Requests
Validation	.065	$18 + 5\times18 =$ 108	$19.44 + (1.43\times19.7) + (.30\times17.7)\approx$ 52.92
Read-locked TLBs	.071	$24 + 5\times18 =$ 114	$19.50 + (1.43\times20.0) + (.05\times18)\approx$ 49.00
Memory-based TLBs	.066	5 or 7	$19.32 + (1.43\times19.7) + (.25\times17.7)\approx$ 51.92

Table 6. Comparison of TLB Consistency Methods

7.9. Summary of Input Parameters

Before describing the programs used to generate our input address traces, we summarize the assumptions made in our model.

1. A TLB has four partitions: user cacheable, user noncacheable, system cacheable, and system noncacheable. Using a fixed TLB miss rate for the user cacheable partition we delay the processor for TLB misses on cacheable references. The user-noncacheable partition of the TLB and the pages associated with this partition are specifically managed so that the TLB miss rate, the TLB management time, and the paging behavior associated with noncacheable memory can be studied.
2. The time required to service a page fault is independent of the TLB consistency scheme employed.
3. TLBs are not purged on task switches and are assumed to be fully associative with strict LRU replacement.
4. An idealized pipelined RISC processor is assumed such that one real instruction (not a “NO-OP”) is executed every cycle.
5. The processor is a Harvard architecture, allowing one instruction and one data reference to be generated at each cycle.

6. The delay time for an event is approximated by the time required to access and process cached data, satisfy cache misses, and access and process noncacheable data. Further delay may be experienced if TLB misses and/or a page fault are encountered.
7. The average event length is a parameter to the simulator; the average number of references per instruction for each trace is computed using this data.
8. A S/370 event is mapped to an event on a hypothetical RISC processor by increasing the number of instructions by 11.4% and decreasing the number of data references by 35%.
9. The separate data and instruction caches of the processor (each having a four-word line size) exhibit a 97% and 99% hit ratio (for cacheable entries), respectively.
10. 75% of data references are considered loads; 25% stores.
11. References generated by cache read misses and loads and fetch-and-ϕs to shared data (network requests) are additionally charged for network transit time plus memory access time since they can not be completed until the generated reference is satisfied and processed.
12. A noncacheable reference is initially charged one cycle for processing.
13. The network transit time plus memory access time which is equal to ω + (m - 1) + network waiting time + 2, where $\omega = 2 \log_2 N$, the time it takes for a round-trip through the network with no contention at the switches, and the waiting time is the sum of the queueing delays at the switches.
14. No additional delay is charged to a memory store except in the case of stores of noncacheable data when either Validation or Memory-based TLBs is employed to ensure TLB consistency.
15. A formula of Kruskal, Snir, and Weiss [1986] is used to approximate network waiting time in the processor-to-memory direction.
16. The network waiting time in the memory-to-processor direction is approximated using a formula of Liu [1990].
17. The switch cycle time equals the processor cycle time.
18. To compute the load offered to the network we iterate to a fixed point, where the expected latency yields a processor utilization consistent with the request rate.
19. To compute the offered load we use a TLB hit rate of 99%.
20. The memory access delay is two switch cycles (for a 10 MIPS processor).
21. The number of packets in a memory request submitted to the network or a related response is two except in the cases of a double-word load response and a double-word store request, when the number of packets is four.
22. Except for the case of Memory-based TLBs, the service time for a TLB miss on noncacheable data depends on T_2, the network transit time (including waiting) and memory access time for one message.
23. A data reference generates either a double-word load or a double-word store. A cache miss generates two double-word loads and a reference to a noncacheable page table entry (as a result of a TLB reload) generates a single-word load.
24. A generic TLB reload (where no TLB consistency technique is used) requires 15 instructions with 2 noncacheable references. The service time (in cycles) for a TLB miss on cacheable data is $9 + 2 \times T_4$.

25. For Validation the service time for a TLB miss on noncacheable data is $18 + 5\times T_2$ or $18 + 4\times T_2$, depending on whether or not stale translation information is used to access memory.
26. When employing Read-locked TLBs, the service time for a TLB miss on noncacheable data is $24 + 5\times T_2$.
27. Using Memory-based TLBs, no network requests are required to service a TLB miss on noncacheable data. Therefore, the TLB reload cost is only 5 or 7 cycles, dependent upon whether or not the page table entry must be accessed from memory.
28. A 3600 rpm movable-arm disk is associated with each paging arena. This device has an average rotation time of 8.33 ms, an average seek time of 15 ms, and a transfer rate of 2 Mbytes/second. Hence, an nK-byte page can be transferred in $23 + \frac{n}{2}$ ms on average. Assuming 10 MIPS RISC processors, the time to read and transfer a page of 2 Kbytes costs approximately 240,000 cycles.
29. The cost of an start-I/O is 1000 cycles.
30. A context switch, performed in user-mode, requires 100 cycles.

7.10. Address Traces

The address traces used to produce our simulation results were generated from two double-precision, floating-point intensive, scientific applications that employ iterative, grid-based algorithms. The computationally intensive portion of each of these programs consists a block of code executed repeatedly, typically until convergence is achieved. The flow of control within this block of code either does not change or changes in a very simple manner, as is illustrated by Darema, Karp, and Teller [1987]. Each program was successfully parallelized and run under both VM/EPEX (up to 32 virtual processors) and WASHCLOTH [Gottlieb, 1980], an instruction-level simulator of the N.Y.U. Ultracomputer.

Since the employed version of the VM/EPEX system restricted the size of a program's shared data to be a maximum of 15 Mbytes, the problem size of each program was defined to meet this constraint. The problem size, and thus the size of shared data, of each of the programs is smaller than desired. In multiprogramming mode, however, using the collected traces our simulator can model a configuration with a much larger shared memory, the size of which is equal to the sum of the shared data utilized by each program. As of yet we have not simulated in multiprogramming mode.

To date, we have collected traces of three parallel programs. These programs are described in some detail by Darema, Karp, and Teller [1987a]. Here we briefly describe the two programs used to generate the input address traces for our simulations, giving only details that are relevant to this thesis.

7.10.1. Peskin

This Fortran program, called Peskin in this thesis, is a simplified two-dimensional version of the blood flow through heart valves. It simulates the flow of an incompressible fluid coupled to an elastic membrane, *i.e.* the boundary behavior in a periodic square domain filled with a fluid. The most time consuming physics of the simulation was practically untouched; the simplification mostly affects the initialization section of the program where the membrane is represented as a stretched ellipse rather than as a realistic three-dimensional representation of the heart. Based on an algorithm of Peskin [1977], the program iteratively recomputes the position of the membrane and the velocity of the fluid for successive discrete times. The problem is defined in a square grid comprised of NxN mesh nodes. As described by Lubachevsky [1981], each iteration requires solving Navier-Stokes equations for the fluid, using an FFT and linear equation solvers, and performing an iterative Newton's minimization of a tension functional for recomputation of the new position of the membrane. The system of linear equations arising in this minimization is solved using the odd-even reduction method of Peskin [1981a].

While at N.Y.U. Lubachevsky parallelized the serial program that implements Peskin's algorithm and ran it under for WASHCLOTH. This program was modified for EPEX.

The program has an initialization section, a computational section that is repeatedly executed until convergence, an I/O section that writes partial results after every n iterations, and a termination section. The files that encapsulate the process flow graph are defined so that the computational and the interim I/O sections can be simulated a specified number of times.

The problem size used in this study has 4.5 Mbytes of shared data, approximately 4.5 Kbytes of private data per process and approximately 650 Kbytes of code. Although the ratio of noncacheable data to cacheable data is large, as shown below in Table 7, the ratio noncacheable references to cacheable references is small. The next larger size problem has 16 Mbytes of shared data, which exceeds EPEX limits. The program is double-precision floating-point intensive. The amount of I/O in the computation loop is very small, primarily contained in the initialization and termination code sections. For this problem size, the maximum size of a parallel task group is 512.

As shown in Table 8, this program, with all I/O statements removed, executed for five iterations under VM/EPEX with 1, 2, 4, 8, 16, and 32 simulated processors had the following execution times. Note that the VM/EPEX system does not take into account the increased cost of memory access as the system size grows.

7.10.2. Simple

The other program used in this thesis is popularly known as Simple (*cf.*, [Crowley, Hendrickson, and Rudy, 1978]); it is a benchmark physics code that models the hydrodynamic and thermal behavior of fluids in two dimensions. The hydrodynamics is a standard two-dimensional Lagrangian formulation using an artificial viscosity. The equations of inviscid compressible hydrodynamics and simple heat conduction are solved by finite difference methods. A tabular ideal gas state equation determines the relationships between the state variables.

The problem is formulated on an N x N mesh; the physical domain is divided into a set of zones. Each problem also has a logical domain and the two domains are connected by the Jacobian of the transformation from one to the other.

The original serial program from Lawrence Livermore Laboratory was parallelized for WASHCLOTH by Lubachevsky. His code was modified for EPEX.

The flow of computation of the program is serial initialization followed by successive iterations of a main loop and finally termination. As described by Axelrod, Dubois, and Eltgroth [1983], each iteration of the main loop advances the solution by a time increment of Δt. The main loop consists of a hydrodynamics sweep of the zones, a heat conduction sweep, and a housekeeping step.

Again, the problem size is defined by the dimensions of the grid. The modeled problem size uses a 256 x 256 grid and requires 11 Mbytes of shared data. The next larger size problem (512 x 512) would require approximately 40 Mbytes of shared memory. Private data for the problem size consume approximately 1 Kbytes of storage per process; code, approximately 100 Kbytes. As is the case with the Peskin program, although the ratio of

Program Section	Noncacheable Reference Count	% Total Reference Count	Cacheable Reference Count	% Total Reference Count
Initialization --Serial Tasks	148846		77210631	
Initialization --Replicate Tasks	538		1745	
Initialization --Total	149384	.19%	77212376	99.81%
Computation --Serial Tasks	6141251		160579258	
Computation --Replicate Tasks	200		1524	
Computation --Parallel Tasks	8388872		205023161	
Computation --Total	14530323	3.8%	365603943	96.2%
Termination --Serial Tasks	30720		15776834	
Termination --Total	30720	.19%	15776834	99.81%

Table 7. Peskin Trace: Noncacheable/Cacheable References

noncacheable data to cacheable data is large, as shown in Table 9, the ratio of noncacheable references to cacheable references is small.

Number of Processors	Execution Time (seconds)
1	92.88
2	69.50
4	40.35
8	34.17
16	22.57
32	20.96

Table 8. Peskin: VM/EPEX Times (five iterations)

Program Section	Noncacheable Reference Count	% Total Reference Count	Cacheable Reference Count	% Total Reference Count
Initialization --Serial Tasks	3251566		43117825	
Initialization --Replicate Tasks	2		15	
Initialization --Total	3251568	7%	43117840	93%
Computation --Serial Tasks	48		3646	
Computation --Replicate Tasks	19		404	
Computation --Parallel Tasks	17269794		128392502	
Computation --Total	17269861	11.86%	128396552	88.14%
Termination --Serial Tasks	467761		21592672	
Termination --Total	467761	2.12%	21592672	97.88%

Table 9. Simple Trace: Noncacheable/Cacheable References

As shown in Table 10, this program (with I/O) run under EPEX with 1, 2, 4, 8, and 16 simulated processors had the following execution times. Again, note that EPEX does not take into account the increased cost of memory access as the system size grows.

Number of Processors	Execution Time (seconds)
1	65.77
2	50.26
4	26.16
8	17.33
16	9.01

Table 10. Simple: VM/EPEX Times (one iteration)

Chapter 8. Simulation Results

In this chapter we evaluate and compare the performance of solutions to the translation-lookaside buffer consistency problem in scalable multiprocessor architectures. Performance was measured via a trace-driven simulation study, for which the three solutions we proposed in Teller, Kenner, and Snir [1988] are the primary subjects. A *translation-lookaside buffer (TLB)* is a special-purpose, virtual-address cache that is used to implement virtual memory efficiently. The *TLB consistency problem* arises in multiprocessors with multiple TLBs when a page-table modification causes TLB entries to become outdated. Out-of-date, virtual-to-physical address translation information that are cached in TLBs can lead to erroneous memory accesses. Therefore, a TLB consistency solution must be employed to prevent the existence of such information or its use to access memory. (Refer to Chapter 2 for background information and Chapter 5 for a description of our three TLB consistency solutions.)

Results of our simulation study that are presented in this chapter are summarized below in numbered paragraphs. A question not answered by prior related research, but answered by the results of our simulation study, is: In a scalable multiprocessor architecture with a multi-stage network interconnecting processors and memory, how does the computational cost of a TLB consistency solution grow as N, the number of processors in the system, increases? In such a multiprocessor, the cost of a solution depends on N. For example, in such a system with processor-based TLBs, the cost of a solution depends on the cost of network transit, which is $O(\log N)$.

Related Research: Black, *et al.* [1989] and Rosenburg [1989] have studied the performance of their TLB consistency solutions in real multiprocessor systems. Their TLB Shootdown solutions interrupt all processors executing processes that are permitted to access a page table when a modification of the page table may cause TLB inconsistencies. Upon being interrupted, these processors participate in the underlying algorithm, which ensures TLB consistency. Black, *et al.* study the performance of TLB Shootdown by executing four parallel programs, which afford a small degree of parallelism, in a 16-processor NS32332 Encore Multimax. The Encore Multimax interconnects processors and memory via a shared bus. Rosenburg compares the performance of his modified version of TLB Shootdown with that of the original algorithm by executing a prototypical parallel program in IBM's 64-processor RP3, which interconnects processors and memory via a multi-stage network. The test program spawns 63 threads within a single address space, binds each thread to a processor, allocates a region of local memory for each thread, executes a parallel computation, and then terminates. (A more comprehensive description of the TLB Shootdown solutions is given in Chapter 4.)

To our knowledge, trace-driven simulations have not been used to study the performance of TLBs in multiprocessor systems. However, trace-driven simulations have been used to study the performance of general-purpose caches in multiprocessors. These simulations conventionally have been driven by either synthetic reference streams [Archibald and Baer, 1986] or a set of address traces, one trace for each of P processors, generated by real multiprocessor systems with $N \geq P$ [Eggers and Katz, 1987].

Simulation Study: Our simulation study of the performance of TLB consistency solutions in scalable multiprocessor architectures includes the effects of varying:

- N, the number of processors,
- network transit time, which is $O(\log N)$,
- TLB size,

- page size,
- workload, and
- frequency with which the TLB consistency problem would arise if a TLB consistency solution was not in place.

In contrast, prior related performance studies were conducted in real systems with 16 or 64 processors, where the average cost of network transit remains relatively constant and the sizes of TLBs, pages, and frames are fixed.

Our simulation methodology allows us to drive the simulations of a range of multiprocessor systems of different sizes and architectures with just one uniprocessor address trace of a parallel program. This contrasts with the conventional practices of either generating N synthetic reference streams, for each N, or employing a set of multiprocessor address traces, one set for each N. In order to study the performance of a multiprocessor with N processors via a trace-driven simulator, a set of N reference streams is needed, one stream for each processor. In order to study the performance of a scalable multiprocessor architecture, one such set is needed for each size system, *i.e.*, for each N.

Our traces and our methodology have been employed by Agarwal and Cherian [1989] to study the cost of synchronization in multiprocessor systems. Our traces and an independent but similar methodology were employed by So, *et al.* [1987] to analyze the speedup of parallel programs, and by Kumar and So [1989] to study nonuniformities in memory access patterns, processor utilization, and occurrence of hot spots in multiprocessor systems.

For processor-based TLBs, our model of TLB-reload overhead confirms our related simulation results. Unlike the other TLB consistency solutions [Cheriton, *et al.*, 1986; Wood, *et al.*, 1986; Goodman, 1987; Thompson, *et al.*, 1988; Black, *et al.*, 1989; and Rosenburg, 1989], which are described in Chapters 3 and 4, the cost of our solutions is embodied primarily in the cost of TLB reloads. (A TLB reload, which is triggered by a TLB miss, loads into a TLB the address translation information for a page.) The computational overhead of our solutions essentially is equal to $R \times T_{reload}$, where R is the number of generated TLB reloads and T_{reload} is the average cost of a reload. Our model of TLB-reload overhead estimates the computational overhead by $R \times C_{reload}$, where C_{reload} is the contention-free cost of a TLB reload.

Results: The results of our study are summarized below:

1. ***O(N) TLB Reloads***: We show how certain parallel programming constructs can cause R to grow linearly with N. Such constructs are contained in the traced programs used in this study. Our simulations of systems with processor-based TLBs that were driven by the address trace of one of these programs validate our model of R's growth with N: R grows linearly with N.

2. ***Nonevidence of O(N) TLB Reloads***: Even though the programs used in our study contain constructs that can cause R to grow linearly with N, our other simulations do not indicate that R grows linearly with N.

3. ***Comparison of Our Solutions***: When we compare the performance of our three solutions we refer to them as two methods of ensuring TLB consistency: processor-based TLBs and memory-based TLBs. This is because our two solutions for systems with processor-based TLBs have similar performance characteristics: T_{reload} grows as $\log N$. Thus, for both processor-based TLB consistency solutions, if R grows linearly with N, TLB-reload overhead grows asymptotically as $N \log N$.

Since the organization of main memory strongly determines the performance of memory-based TLBs, we briefly digress to define the concept of a paging arena: Memory can be organized into clusters, where the words of a page are interleaved across the memory modules of a cluster. In a system with memory-based TLBs, the mapping of pages to a cluster is fixed and, thus, a memory cluster is called a *paging arena*.

For the studied workloads and within the range of *N* for our experiments, our results show that:

- The performance of memory-based TLBs depends on the number of paging arenas.
- Processor-based TLBs outperform memory-based TLBs in systems with one paging arena; memory-based TLBs outperform memory-based TLBs in systems with two paging arenas.

4. ***Our Solutions vs. TLB Shootdown***: Although we did not simulate systems employing TLB Shootdown, for the workloads studied that illustrate *R* growing linearly with *N*, we give evidence that for large *N*, TLB Shootdown would have a significantly higher computational overhead than our solutions. This comparison is based on simulation data and an estimate of TLB Shootdown's overhead. For a system that employs TLB Shootdown, the simulation data defines the TLB-reload overhead and the number of invocations of the TLB Shootdown algorithm.

Chapter Organization: This chapter is divided into seven sections: Section 8.1 describes the experiments that comprise our study and explains the notation used in this chapter and the accompanying graphs. Section 8.2 shows how reference patterns produced by certain parallel programming constructs can result in *R* growing linearly with *N*, and discusses other factors which effect *R*'s growth. The following sections present the results of our simulation study: Sections 8.3, 8.4, and 8.5 present results regarding the performance of our TLB consistency solutions. Section 8.6 compares the performance of TLB Shootdown to that of our solutions. Finally, in Section 8.7, we present general simulation results that relate to TLB-miss and TLB-reload rates, page-in activity, and execution time.

8.1. Experiments and Notation

This section describes the experiments that comprise our study and explains the notation used in this chapter and the accompanying graphs. Our study was conducted using a simulation system that was designed and implemented according to the simulation methodology described in Chapter 6. As mentioned above, the primary subjects of our study are the solutions we proposed in Teller, Kenner, and Snir [1988]: Validation and Read-locked TLBs, which assume a TLB located at each processor, and Memory-based TLBs, which assumes a TLB located at each memory module.

The simulated, scalable multiprocessor resembles the NYU Ultracomputer [Gottlieb, 1987], particularly in the sense that the processors and memory modules are interconnected via a multi-stage Omega network and all memory modules are equidistant from each processor. Architectural characteristics of this multiprocessor, *e.g.*, the network switch size and the speed of the I/O devices, are simulator input parameters and are discussed in Chapter 7.

Our simulations were driven by the address traces of two parallel programs:

1. Simple: a benchmark physics code that models the hydrodynamic and thermal behavior of fluids in two dimensions, and
2. Peskin: a simplified two-dimensional model of the blood flow through heart valves.

These programs are described briefly in Chapter 7 and in detail in Darema, Karp, and Teller [1987]. For each program, one parameter defines the size of the problem and the maximum, available parallelism. Due to limitations inherent in IBM's VM/EPEX system [Darema-Rogers, *et al.*, 1985; Stone, *et al.*, 1985] with respect to the size of a program's shared data, Simple's parameter and, thus, the maximum, available parallelism was limited to 256. In the case of Peskin, the maximum, available parallelism was limited to 512.

8.1.1. Experiments

An *experiment* consists of a set of simulations that are driven by the same program trace and model systems differing only in terms of N. The underlying architecture of the simulated systems is fixed, as defined in Chapter 7, and the TLB and physical memory sizes are fixed.

We define six different experiments and, thus, six *parallel workloads* for each program trace by using three different TLB sizes for each of two physical memory sizes. A TLB contains 16, 32, or 64 entries. In the context of this chapter, the size of *physical memory* refers to the number of frames reserved for either 33% or 50% of the number of user, shared pages in virtual memory. The size of *virtual memory*, which refers to the number of pages reserved for user, shared pages, is determined by the size of a traced program's shared data. For a fixed page size, the physical memory size determines the number of page-ins and, thus, the number of page faults that arise during a simulation. A page fault may cause a processor to context switch to another task.

A parallel workload is likely to differ somewhat among the simulations that comprise an experiment. This is because the scheduling of tasks to processors and, thus, the sequence of instructions that each processor executes is likely to differ for different N. (In this chapter, the terms process and task are used interchangeably.)

For each trace, a set of these six different experiments was conducted three times, once for each of our TLB consistency solutions. Thus, we often refer to such a set of experiments with respect to the solution they employ and the trace that was used to drive the related simulations. For example, we refer to the Validation/Simple experiments or, in the context of a discussion of Validation, we refer to the same set of experiments as the Simple experiments. Analogously, we refer to systems that employ Validation as Validation systems. In general, an experiment includes simulations with $N = 2^i$, $i = 0, \ldots, 6$. Memory-based TLBs experiments also include simulations with $N = 2^7$. In this way, we are able to evaluate the performance of our TLB consistency solutions in scalable multiprocessor systems.

In our simulations the physical memory size determines the number of page-table modifications that would result in TLB inconsistencies if a TLB consistency solution was not in place. This is because, unlike a real system, page evictions are the only events that require page-table modifications. Such page-table modifications cause systems employing Validation or Memory-based TLBs to incur extra computational cost: In a Validation system, stale TLB entries may be generated and then corrected. In a Memory-based TLBs system, unnecessary network traffic may be generated.

A *simulation* encompasses the execution of one iteration of the initialization, computational, and termination sections of a program. Unless otherwise noted, our graphs are produced from data collected from entire simulations. We also keep track of data in other ways, *e.g.*, with respect to each different section of a simulation, which corresponds to each different section of a program. Thus, when it makes sense we present a subset of the data collected during an entire simulation. For example, at times we present data concerning an event that was collected during only a simulation's computational section.

In this study, the performance of our TLB consistency solutions can be evaluated by examining data collected during either an entire simulation or one iteration of the simulation's computational section. This is because, for each of the programs used in our study, R's growth with N is characterized by the program's computational section. The initialization sections of the Simple and Peskin programs primarily consist of serial sections that initialize shared variables. The initialization sections also include small replicate sections that initialize private variables and copy shared, read-only data to private pages. The number of shared pages referenced in these replicate sections is insignificant. The termination sections of these programs consist of only one serial section. Thus, the number of TLB reloads generated during the initialization and termination sections of the simulations essentially does not change as N increases. The computational sections of these programs are executed repetitively, and each iteration is composed of essentially the same code (*i.e.*, the flow of control of each iteration is essentially the same). Thus, for a specific simulation, regardless of the number of iterations executed, the number of reloads generated by an iteration of the computational section essentially does not change from one iteration to the next. For example, assume that c reloads are generated by the initialization and termination sections of a simulation, and $aN + b$ reloads are generated by the computational section of the simulation. Then, if the computational section is simulated n times, $R \approx c + naN + nb$.

8.1.2. Notation

Before beginning our discussion of the results of our simulation study, we discuss the notation used in this chapter and the accompanying graphs. Referring to Graph 1, note the caption in the upper right corner of the graph. This caption, *Simple/4k page/33%*, identifies the experiments that generated the associated data. The three fields that comprise the caption denote, in order, the:

1. program trace used to drive the experiments,
2. defined page size, and
3. defined physical memory size, in terms of the percentage of virtual pages that can be memory resident.

Since all of our experiments assume a 4-Kbyte page size, we drop the page-size field in the text that follows, even though we include it in the captions of our graphs. The virtual memory size for Simple is 2497 pages and for Peskin, 1083. Thus, in the case of the Simple/33% experiments, the data was generated by simulations of systems with a physical memory comprised of 830, 4-Kbyte frames, that were driven by Simple's address trace.

Normally, one graph presents the data generated by a set of three experiments for a specific TLB consistency solution and address trace. These experiments are for a fixed physical memory size and varying TLB size. Accordingly, the data of these experiments are identified, as in Graph 1, by the labels: *16 TLBEs*, *32 TLBEs*, and *64 TLBEs*. If a graph or table depicts data of only one of these experiments, then the TLB size is included in the caption. For example, refer to Graph 9 with the caption *Simple/4k page/33%/64 TLBEs*, which will be referred to in the text as *Simple/33%/64 TLBEs*. This caption denotes that Graph 9 depicts data collected during the experiment that is comprised of simulations of systems with a physical memory of 830, 4-Kbyte frames and 64-entry TLBs, that were driven by Simple's trace.

In our graphs, actual data points are shown and joined by lines, the style of which indicate the experiment that generated them. For example, in Graph 4, the data points provided by the experiment for systems with 16-entry TLBs are joined by a solid line, while the data points provided by the experiment for systems with 32-entry TLBs are joined by a dashed line. In contrast, a regression line fitted to the data points generated by an experiment is denoted by a dotted line without data points. A regression line is labeled as *Fit*. As an example, consider the dotted lines in Graph 4, which denote the lines generated by regression analysis using as input the sets of data points generated by the related Validation experiments.

In some graphs we use the following abbreviations:

- *Val* for Validation,
- *Lock* for Read-locked TLBs,
- *Mem* for Memory-based TLBs, and
- *Est OH* for estimated, TLB-reload overhead.

The estimated, TLB-reload overhead represents our modeled, TLB-reload overhead for systems with processor-based TLBs. As mentioned in the introduction to this chapter, this is computed by multiplying *R*, as reported by a simulation, times the contention-free TLB-reload cost (C_{reload}).

In most cases, the y axis of our graphs has a standard, linear scale. However, in a few instances the scale of the y axis is logarithmic. Often the scale of the x axis is logarithmic. A logarithmic scale is based on $\log_{10}$.

8.2. R's Growth As N Increases

The model that we present in this section is used to illustrate how certain parallel programming constructs generate reference patterns that can cause *R* to grow linearly with *N*, and why some of our experiments indicate this growth. This model makes several assumptions that may not be true in real systems. For example, assumptions are made regarding TLB size and task scheduling that generally are not realistic. Such constructs are contained in the traced programs used to drive the simulations that comprise our experiments. Each of these programs solve a fixed-size problem, where the maximum, available parallelism is either 256 or 512.

A linear growth of *R* with *N* may become apparent only beyond a certain *N*, TLB size, page size, or in the case of memory-based TLBs, the number of paging arenas, or may be evidenced only by certain schedulings of tasks to processors. For example, as we illustrate in this section, if some of our model's assumptions are dropped, the number of related TLB reloads may not grow linearly with *N*. Thus, even though a program contains such constructs, experiments driven by such a program's trace may not evidence *R* growing linearly with *N*. Accordingly, the remainder of our experiments do not show this growth: Instead, *R* grows sublinearly with *N*. None of our experiments show *R* growing superlinearly with *N*. Observations for systems with processor-based TLBs are mixed, whereas all of our experiments for memory-based TLBs indicate *R* growing sublinearly with *N*.

In Section 8.2.1 we show why *R* may grow linearly with *N*. We give examples of parallel programming constructs contained in the programs used in this study that can generate a number of related TLB reloads that grows linearly with *N* and, thus, can cause *R* to grow linearly with *N*. In addition, we show how changes in TLB size and task scheduling can effect *R*'s growth with *N*. Finally, we show how these parallel programming constructs impact the effective size of processor-based versus memory-based TLBs. Section 8.2.2 discusses other factors that effect *R*'s growth.

8.2.1. Why *R* May Grow Linearly With *N*

The SPMD (single-program, multiple-data) programming model [Darema-Rogers, *et al.*, 1985], which is adopted by VM/EPEX, defines three different types of code sections:

1. serial,
2. replicate, and
3. parallel.

Although the programs used in our study follow this model, unlike the VM/EPEX system, our simulator does not bind tasks to processors. Instead, we employ a self-service paradigm, where processors are allocated tasks from a shared task queue. The tasks that comprise a program are placed on the shared task queue. Precedent relationships among tasks are maintained by locking the task queue when necessary.

In this section, we show that:

- Only in systems with memory-based TLBs can a serial code section generate a number of TLB reloads that grows linearly with *N*.
- In a system with processor- or memory-based TLBs, a concurrent task group, *i.e.* a group of tasks that can be executed concurrently, can cause *R* to grow linearly *N*. The linear growth in systems with processor-based TLBs generally results from page sharing. In systems with memory-based TLBs, it generally results from the replication of translation information among all the TLBs in a paging arena.

We divide this section into four subsections, three of which are dedicated to the three types of code sections mentioned above, and one which discusses the effective size of processor-based TLBs versus memory-based TLBs.

Although we study the performance of only the partitions of TLBs reserved for user-shared pages, in the next three subsections we illustrate how references in the three types of code sections, to both private and user-shared pages, can cause *R* to grow linearly with *N*.

The model used in this section is defined by 11 assumptions, only seven of which are presented below. The remaining assumptions are applicable to concurrent task groups of size greater than one. Therefore, they are presented in the following section.

1. A shared page is referenced using a unique, global, virtual address.
2. A processor-based TLB is large enough to store the translation information for all the pages referenced by a given task.
3. We do not neglect valid TLB-resident translation information loaded on behalf of tasks in the same concurrent task group as the task in question.
4. All other valid TLB-resident translation information is neglected.
5. A memory-based TLB is large enough to store the translation information for all pages in its cluster that are referenced by a set of at most *N* tasks in a concurrent task group.
6. *M*, the number of tasks in a concurrent task group, is greater than or equal to *N*.
7. A task, once scheduled to a processor, remains processor resident until its execution is complete.

In essence, assumptions 2 through 5 imply that TLBs are considered to be cold and infinite at the beginning of the execution of a concurrent task group. In addition, TLBs are considered to be warm with respect to pages actively shared among tasks in a concurrent task group while the group is executing.

Relaxing assumption 1, 2, 3, 5, or 7 is likely to increase the number of generated TLB reloads. Relaxing assumption 1 means that tasks may not be able to access the same shared page with the same translation information. Accordingly, tasks may not be able to use valid, TLB entries that were loaded on behalf of other tasks. Assumptions 2 and 5 guarantee that at most one TLB reload is required to access a referenced page. Dropping either of these assumptions, TLB entries can be replaced before they are reused. If we drop assumption 3, then a processor-based TLB is always considered to be cold. Therefore, the translation information for each page referenced by a task must be loaded at least once. Dropping assumption 7 may increase the number of reloads, and dropping assumption 4 may decrease the number of reloads.

8.2.1.1. Serial Code Sections

A serial code section defines a serial task to be executed by only one processor; a serial task is a concurrent task group of size one. Therefore, pages referenced by a serial task are referenced by only one processor during a specific time interval.

A serial task executed in a system with processor-based TLBs generates one reload per referenced page, while in a system with memory-based TLB it generates an average of c ($1 \leq c \leq C$) reloads per referenced page. In a system with memory-based TLB, one reload is generated for each of the c memory modules accessed in the referenced paging arena, in order to access the referenced words of the page. Thus, a serial task generates c times as many memory-based TLB reloads as processor-based TLB reloads. (Note, however, that the cost of a memory-based TLB reload can be much smaller than that of a processor-based TLB reload.)

The number of processor-based TLB reloads generated by a serial task remains constant as N increases, in comparison to the $O(c)$ memory-based TLB reloads generated. If c is $O(N)$, a serial task generates $O(N)$ memory-based TLB reloads.

8.2.1.2. Replicate Code Sections

A replicate code section causes N replicated tasks to be created. These tasks are then executed redundantly by the N processors cooperating in the execution of the program. Before executing a serial or parallel task, a processor must execute all replicate tasks that are its ancestors in the program's task precedence graph.

Since this and the next subsections discuss concurrent tasks groups, we first present the four other assumptions that define our model. These are in addition to the seven presented at the beginning of Section 8.2.1:

8. The replicate tasks created from a replicate code section constitute a concurrent task group of their own. In other words, although in some cases a replicate task can be executed concurrently with other types of tasks, we assume that the tasks created from a replicate code section are executed concurrently and no other tasks can be executed concurrently with any one of these tasks.
9. C is less than or equal to the maximum available parallelism.
10. All N processors in a system are cooperating in the execution of the related program.
11. When tasks can be executed concurrently and M, the maximum available parallelism, is greater than or equal to N, then all N processors concurrently execute a task.

As we show below, in a replicate code section:

- References to a private page generate N processor-based TLB reloads or $N \times c$ ($1 \leq c \leq C$) memory-based TLB reloads.

- References to a shared page generate N processor-based TLB reloads or c memory-based TLB reloads.

As the TLB or page size increases, so does the probability that processors will share translation information. As the page size increases, the replication of translation information may be increased.

8.2.1.2.1. Private Pages: Each private page referenced in a replicate code section causes N distinct private pages to be referenced by N processors, respectively. Accordingly, N processor-based TLB reloads are generated.

In a system with memory-based TLBs, references to a private page in a replicate code section generate $N \times c$ TLB reloads, where c $(1 \le c \le C)$ is the average number of memory modules accessed per referenced private page. Recall that our model considers memory-based TLBs to have an infinite number of entries while a replicate task group (which is a concurrent task group of size $M = N$) is executing. As a result, TLB reloads generated for the use of one process do not replace translation information loaded on behalf of another, concurrently executing process.

8.2.1.2.2. Shared Pages: A shared page referenced in a replicate code section is accessed by all N processors and, therefore, generates N processor-based TLB reloads.

Under our assumptions, an average of c $(1 \le c \le C)$ memory-based TLB reloads are generated per shared page referenced in a replicate code section. This is because we assume that the translation information for a shared page referenced by a concurrent task group remains resident during the execution of all the tasks in the group.

If we drop either assumption 5 or 11 (*i.e.*, if we assume finite-size memory-based TLBs or assume that tasks of a replicate task group are not executed concurrently), instead of c reloads per referenced shared page, $N \times c$ reloads can be generated. In this case, each time the translation information for a shared page is loaded, it remains TLB resident only during the execution of one of the replicate tasks.

8.2.1.3. Parallel Code Sections

For similar reasons to those outlined in Section 8.2.1.2, the number of TLB reloads generated by references to private pages in a parallel code section can grow linearly with N. This is because both replicate and parallel code sections define concurrent task groups, and in the SPMD programming model, private pages are associated with processors rather than processes.

Unlike the tasks of a replicate task group, the parallel tasks created from a parallel code section usually are different, at least in terms of referenced shared data. That is, shared pages referenced in a parallel code section need not be referenced by all of the tasks in the associated parallel task group. In addition, parallel task groups can be of size M, where M is not necessarily equal to N. For the traced programs used in this study, tasks in two different parallel task groups cannot be executed concurrently. Thus, in this section we assume that each group defines its own concurrent task group.

In contrast to a replicate task group, even if a shared page is referenced by all the tasks in a parallel task group, it is not necessarily the case that the number of related TLB reloads grows linearly with N. Under our assumptions, in the case of processor-based TLBs, this growth is evidenced because translation information loaded on behalf of one task in the group is available for the use of another task in the group. This is not the case if we drop assumption 2 or 3 (*i.e.*, if we consider processor-based TLBs to be of a finite size or we neglect valid TLB entries loaded on behalf of previously executed tasks in the group). Then, the number of reloads may not grow linearly with N. For example, if TLBs are always cold when a task begins to execute, then in order for a task to access any page, the page's translation information must be loaded into the processor's TLB, at least once. Consequently, references to a page that is actively shared by all the tasks in a parallel task group generate at least M TLB reloads, independent of N.

References to the the same shared datum by all the tasks in a concurrent task group can cause the number of related processor-based TLB reloads to grow linearly with N, but do not change the number of related memory-based TLB reloads. As previously discussed, the number of memory-based TLB reloads can grow linearly with N even when a page is referenced by only one task: If references to the page require that all C memory modules in a paging arena be accessed and C is $O(N)$, then the number of related reloads grows as $O(N)$. Below we list examples of shared data, which is stored in shared pages, that may be accessed by all the tasks in a concurrent task group:

1. shared arrays,
2. iteration variables of parallel loops,
3. accumulators, and
4. synchronization variables.

In the next two subsections, we present examples of parallel code sections, from the programs Simple and Peskin, that illustrate how the number of related TLB reloads are caused to grow linearly with N.

8.2.1.3.1. Example One: First we consider a parallel code section in the subroutine ESUBK of the program Simple. This code section defines a parallel task group of size 256 and has the following general structure.

```
PARDO 200 J = 1, 256
   DO 100 K = L, M
      SQ = U(J,K)**2 + V(J,K)**2
      ADD = 0.
      ADD = ADD + AMASS(J,K+1)
      Fetch-and-Add(ACCUM, SQ*ADD)
   100 CONTINUE
200 CONTINUE
```

The execution of a task defined by this code section requires the translation information for the shared pages that store J, ACCUM, and one row of each of the arrays U, V, and AMASS. In addition, the translation information for the shared pages that store referenced synchronization and operating systems variables and the referenced private pages, such as the ones that store the private variables L and M, are needed.

Suppose that J, a shared iteration variable, and ACCUM, a shared accumulator, are stored in the same page. Then, under our assumptions, the related references generate N processor-based TLB reloads or one memory-based TLB reload. If we drop assumption 3 and, hence, neglect TLB-resident translation information loaded on behalf of a previously executed task in the defined parallel task group, then $M = 256$ processor-based TLB reloads are generated, one for each task in the group.

Now let us consider the number of TLB reloads generated by the references to the shared arrays U, V, and AMASS. Each task references essentially all the elements of a unique row of each of the three shared arrays. Or, in other words, each task references one element in each column of the three shared arrays. Array elements are stored in double words in column-major order, and each array is dimensioned as 256×256. Thus, for the problem size simulated, the maximum value of J is 256, and K ranges from 1 to 255.

Given a 4-Kbyte page, which stores two columns of an array, a task's references to the shared arrays U, V, and AMASS generate 3×128 processor-based TLB reloads, one for each pair of columns of each array. Therefore, $N \times 3 \times 128$ related processor-based TLB reloads are generated. If we drop assumption 3, then $M \times 3 \times 128$ processor-based TLB reloads are generated.

In a system with memory-based TLBs, the number of related memory-based TLB reloads is $C \times 3 \times 128$. This is because, even though all processors share the translation information for the referenced pages, the elements of each array are distributed among all C memory modules of a cluster. If C is $O(N)$, the number of related memory-based TLB reloads grows linearly with N.

8.2.1.3.2. Example Two: Next we consider the general structure of an example taken from the program Peskin, where f and g are (inline) arithmetic expressions.

```
PARDO 700 K = 1, 512
    Call KERN(K)
700 END PARDO

Subroutine KERN(K)
DO 10 I = 1, 2
    IX(I,K) = f(X(I,K))
10 CONTINUE
DO 20 I = 1, 4
DO 20 J = 1, 4
    D(I,J,K) = g(IX(1,K), X(1,K), IX(2,K), X(2,K))
20 CONTINUE
RETURN
```

The execution of a task defined by this code section requires the translation information for the shared pages that store the arrays IX, X, and D, and for the pages that store the shared iteration variable, K, and referenced synchronization and operating system variables.

As in Example One, the references to the shared iteration variable, K, generate one memory-based TLB reload, or $M = 512$ or N processor-based TLB reloads, depending on whether or not we drop assumption 3.

This code section was written so that each task processes one column of IX, X, and D. IX and X are dimensioned as 2×512, and D is dimensioned as 4×4×512. A column of D is defined as D(I, J, K), with I and J varied from 1 to 4, and K fixed. Array elements are stored in double words, and the elements of an array are stored in column-major order. Assuming a 4-Kbyte page size, the elements of 256 columns of X or IX are stored in one page and the elements of 32 columns of D are stored in one page, for a total of 20 pages. The execution of a task requires that three pages be accessed.

The number of related processor-based TLB reloads depends on how the $M = 512$ tasks are scheduled to processors. To minimize the number of reloads in a system with N processors numbered as $i = 1, \ldots, N$, processor i should be assigned tasks with $K = (i-1)P + 1, \ldots, iP$, where $P = M/N$. This scheduling of tasks to processors maximizes the page sharing among tasks executing on the same processor and, thus, minimizes the number of related TLB reloads.

For $N = 1$, this task scheduling generates 20 processor-based TLB reloads, one for each of the two shared pages storing IX and X, and one for each of the 16 pages storing D. For $N > 1$, this task scheduling generates $N \times (2 + \lfloor 16/N \rfloor + (\lfloor N/16 \rfloor \times 16/N))$ processor-based TLB reloads. That is, each of the N processors must access one each of the pages that store IX and X. For $N = 2^i$, $1 \leq i \leq 4$, D's 16 pages are distributed evenly among the N processors and, thus, generate $16/N$ reloads per processor. For $N = 2^i$, $5 \leq i \leq 9$, each of D's 16 pages are shared by $N/16$ unique processors and, thus, generate one reload per processor.

In a system with memory-based TLBs, the number of generated TLB reloads also depends on the scheduling of tasks to processors. The minimum number of reloads is generated when the ith set of N tasks executed has $K = (i-1)N + 1, \ldots, iN$. In this way, $20 \times C$ reloads are generated, C for each page that stores the referenced arrays. If C is $O(N)$, then the number of memory-based reloads grow linearly with N.

8.2.1.4. Effective TLB Size

So far, in this section we have shown how certain parallel programming constructs generate memory reference patterns that can cause R to grow linearly with N. Now we compare the impact that these reference patterns have on the effective size of processor-based TLBs versus memory-based TLBs of fixed and equal sizes. Thus, in this section we drop assumptions 2 and 5 and, hence, consider both processor- and memory-based TLB to be of finite size.

The memory-based TLBs effectively constitute one, large, shared TLB, with variable-size sets of entries used by individual processors and sets of processors. Accordingly, the size of a processor's memory-based TLB changes dynamically: At a specific time the effective size of a processor's memory-based TLB equals the number of unique TLB entries that are used by the processor. Thus, if a memory-based TLB entry is shared among P processors, then that entry may account for one entry in the calculation of the effective size of each of the P processors' memory-based TLBs. Replications of an entry that are used by a processor are not counted in the calculation of the effective size of its memory-based TLB.

The next two subsections illustrate how:

- replication of translation information can cause the effective size of memory-based TLBs to be smaller than that of processor-based TLBs and
- sharing of translation information among processors can cause the effective size of memory-based TLBs to be larger than that of processor-based TLBs.

Again, note that as the TLB or page size increases, so does the probability that processors will share translation information. As the page size increases, the replication of translation information may be increased.

8.2.1.4.1. Pages Referenced by One Processor: If only one processor references a (private or shared) page during a specific time interval, at any time during the interval, at most one processor-based TLB entry is used to store the page's translation information. In contrast, at any time during the same time interval, at most c
($1 \leq c \leq C$) memory-based TLB entries are used to store the page's translation information. Thus, the effective size of a processor's memory-based TLB may be smaller than that of a processor-based TLB.

If $N > 1$ and a serial task is being executed, the effective size of the memory-based TLB associated with its executing processor is equal to or greater than that of a processor-based TLB. The effective size of the memory-based TLB depends on how many TLBs contain the same entries. The sizes are equal when each of the memory-based TLBs contain the same entries. Since no other processor is executing, the maximum-size TLB is comprised of all the entries in all the memory-based TLBs.

If $N > 1$, N concurrent tasks are being executed, and only one processor references a page during a specific time interval, then the memory-based TLB entries used to store the page's translation information can cause the effective size of a processor's TLB to be smaller than that of a processor-based TLB. For example, suppose that memory-based TLB entries are distributed equally among all N processors. (Thus, the effective size of each processor's memory-based TLB is equal to that of a processor-based TLB.) Then, the replication at all C TLBs in a paging arena of the translation information for a page that is referenced by only one processor can reduce the effective size of each of the other processors' TLBs by one. Accordingly, the effective size of a memory-based TLB can be smaller than that of a processor-based TLB.

8.2.1.4.2. Pages Referenced by Multiple Processors: If $P > 1$ processors reference a shared page during the same time interval, at any time during the interval, at most one processor-based TLB entry per processor is allocated to the page. Therefore, at any time during the interval, as many as P processor-based TLB entries are allocated to the page. The corresponding number of memory-based TLB entries per processor depends on the generated reference pattern.

If all P processors are referencing words of the page that are stored in the same memory module, then only one memory-based TLB-entry is needed. In this case, the effective size of a memory-based TLB can be larger than that of a processor-based TLB, since the one TLB entry effectively counts as N TLB entries.

If $C = P$ and all P processors are accessing all C memory modules in a paging arena in order to reference words of the page, then the size of processor- and memory-based TLBs, can be equal. This may also be the case if each processor references a unique memory module.

8.2.2. Other Factors that Effect R's Growth

As was shown in the previous section, R's growth with N can be effected by the TLB size, page size, or the scheduling of tasks to processors. In this section we discuss another way in which R's growth can be effected by task scheduling and how it can be effected by TLB-entry thrashing.

8.2.2.1. TLB Contexts

Task scheduling effects R's growth in another way: As N increases, the scheduling of tasks and, thus, the TLB context in which a task is executed may change. A change in TLB context may result in an increase or a decrease in the number of generated TLB reloads. Consider a program that is executed by one processor so that the tasks comprising the program are executed in a specific order. Dropping assumption 4 (*i.e.*, considering valid TLB entries loaded on behalf of previously executed tasks) and assuming that a TLB is not flushed on a context switch (as is assumed by our simulator), at a specific time during the execution of the program the context or state of the processor's TLB is defined by the tasks already executed. If the same program is executed in a multiprocessor and the tasks comprising the program are distributed among all N processors, then a processor executes a random subset of the tasks it would have executed if it alone had executed the entire program. Thus, the TLB context in which a specific task is executed may change with N. The TLB context in which a task executes determines which TLB entries are replaced and, thus, the number of TLB reloads generated during the task's execution.

8.2.2.2. TLB-entry Thrashing

In a system with processor-based TLBs, the TLB context in which a task executes is determined by the tasks that were previously executed on the same processor. In a system with memory-based TLBs, a task's TLB context may be determined by tasks that were previously executed on its processor, as well as other processors. This is because memory-based TLBs essentially constitute one, large, shared TLB, where interprocessor TLB interference causes TLB-entry thrashing. *TLB-entry thrashing* refers to the replacement of translation information before it can be reused. TLB-entry thrashing in systems with memory-based TLBs is caused by multiple processors concurrently accessing the same TLB. An example of this is when the translation information for page *a* is loaded for the use of one processor and then, before the processor can reuse the translation information, it is replaced with the translation information for another page, on the behalf of another processor. Thus, TLB-entry thrashing occurs in realistic systems where assumption 5 is not met (*i.e.*, in systems where memory-based TLBs have a finite size).

A processor-based TLB is conventionally accessed by only one process during the same time interval that a memory-based TLB is accessed by multiple processes. TLB-entry thrashing in systems with processor-based TLBs results from interprocess TLB interference, which may occur when a processor context switches from one process to a second process, and then context switches back to the first process. In the remainder of this thesis, when we refer to TLB-entry thrashing, it is in the context of systems with memory-based TLBs and refers to interprocessor TLB interference.

8.3. Simulation Results: Validation

In this section we present the results of our simulations of systems employing Validation. First, we briefly describe the Validation method of ensuring TLB consistency and summarize our simulation results. Then, in the three subsections that follow, we present supporting data produced by our Validation experiments.

Validation solves the most general TLB consistency problem by detecting stale translation information before it is used to access memory. This is accomplished by enhancing a generic TLB-reload routine and, thus, increasing the cost of a TLB reload. The generic TLB-reload routine does not ensure TLB consistency. As described in Chapter 5, in a Validation system, TLBs are located at the processors. Each TLB entry has a field that stores a generation count that is loaded from a Validation table, one of which is located at each memory module of a cluster. The TLB-reload routine uses a two-counter algorithm, discussed in Chapter 5, which ensures the integrity of the generation count associated with a frame. The generation count stored in a TLB entry accompanies a memory request and acts as a time stamp that allows detection of stale translation information. Before the execution of a page-table modification that could otherwise cause TLB inconsistencies is executed, the related frame's generation count is updated in the Validation tables of the frame's memory cluster. A memory request generated by using stale translation information can be detected by comparing the transmitted generation count to the one stored in a Validation table. If they are not the same, the memory access is aborted, and the referencing processor is notified to update the related translation information. Otherwise, the memory request is satisfied.

For a scalable multiprocessor architecture, the performance of Validation depends on N and the growth of R (the number of TLB reloads) as N increases. The cost of a TLB reload grows as $\log N$ and, as shown in Section 8.2, R may increase linearly with N. If R grows linearly with N, then Validation's TLB-reload overhead (OH_{Val}) grows asymptotically as $N \log N$.

Our simulation study indicates the following:

1. As discussed in Section 8.3.1.1, for our Validation/Simple experiments R grows linearly with N.
2. Both the Simple and Peskin programs contain programming constructs that can cause R to grow linearly with N. However, the data provided by Validation/Peskin simulations do not illustrate this growth for the related workloads and the range of N of our experiments.
3. As discussed in Section 8.3.2, our model of processor-based TLB-reload overhead confirms our simulation results for processor-based TLB consistency solutions.
4. Our simulations indicate that OH_{Val} grows as $\log N$. This is the case even for our simulations that report Validation's R (R_{Val}) growing linearly with N, for which we expect OH_{Val} to grow as $N \log N$. As discussed in Section 8.3.3, this apparent inconsistency between our model and our simulation results is explained by the coefficients present in the equations that describe R_{Val}'s growth. The coefficients delay the onset of the asymptotic $N \log N$ growth rate of OH_{Val} beyond our experiments' range of N. (We use the term asymptotic somewhat loosely in this chapter since a program's maximum available parallelism clearly limits the value of N to which we refer.)
5. In general, R_{Val} and OH_{Val} decrease as the TLB size increases. This is illustrated in Graphs 1–8.

8.3.1. Validation's R

A linear growth of R with N is exhibited by the Validation/Simple experiments. These results are illustrated in Graphs 1 and 2, and the associated entries of Table 11. Graphs 3 and 4, and the associated entries of Table 11, show that our Validation/Peskin experiments (for $N = 2^i$, $i = 0, \ldots, 6$), do not evidence R's linear growth with N.

We used linear regression to characterize R's growth with N. The lines defined by the resulting equations, which are presented in Table 11, are depicted along with our experimental results in Graphs 1–4. Table 11 gives, in parentheses under the caption that describes a set of experiments, the number of the graph that illustrates the related simulation results and regression lines. The regression equations are of the form $y = ax + b$, where $y = R$ and $x = N$.

Graphs 1–4 show that the growth of R with N is traceable to the computational section of the associated program. In these graphs, data provided by the computational sections of the simulations of systems with 64-entry TLBs are depicted along with data collected during entire simulations of systems with 16-, 32-, and 64-entry TLBs.

8.3.1.1. Simple Experiments

The Simple experiments provide data that illustrate a definite linear relationship between R_{Val} and N. In Table 11, this is indicated by the values of r^2, the coefficient of determination of the related linear equations, being greater than .99. This means that the straight lines defined by the equations describe the observed values of R_{Val} very well or, in other words, the regression lines fit the data points generated by our experiments very well. The linear relationship between R_{Val} and N is illustrated in Graphs 1 and 2, where the regression lines fit the data so well that they can hardly be seen.

Since the Simple experiments indicate that R_{Val} grows linearly with N, it is clear that the corresponding executions of the Simple program are examples of the workloads that produce $O(N)$ processor-based TLB reloads. Such workloads were discussed in Section 8.2.

Other data generated by the Simple experiments may partially explain this behavior. This data indicates that the majority of page-ins (in this context a page-in refers to a particular main-memory residency of a page) are referenced by one processor and the majority of the remaining page-ins are referenced by N processors. In a system with processor-based TLBs, a *fully-productive page-in*, *i.e.*, one referenced by all N processors, generates a minimum of N TLB reloads. For the Simple/33%/64 TLBEs experiment, this data is presented in Table 12 and Graph 9. For the Simple/50%/64 TLBEs experiment, it is presented in Graph 10. Table 12 and Table 15, which is discussed below with respect to the Peskin experiments, indicate the percentage of page-ins referenced by each number of processors in systems with 2^i, $i = 0, \ldots, 6$, processors during an entire simulation. In parentheses below each of these percentages, we indicate the percentage of page-ins referenced by more than one processor. Very similar percentages apply if we examine the results gathered during only the computational section of the simulations, because the majority of page-ins occur during the computational section.

In the case of the Simple experiments, fully-productive page-ins may exclusively explain the linear growth of R_{Val} with N: There is a high correlation between the number of additional TLB reloads generated during the computational section of the Simple simulations in an N-processor system, as compared to a one-processor system, to the respective minimum number of reloads due to fully-productive page-ins. Corresponding data provided by the Simple/33%/64 TLBEs and Simple/50%/64 TLBEs experiments is presented in Table 13 and Table 14, respectively. We use this data, instead of data collected during entire simulations, because fully-productive page-ins are rare or nonexistent in the initialization and termination sections of the traced programs. In these tables, the column labeled $R_N - R_1$ indicates the additional number of TLB reloads generated in an N-processor system, as compared to a one-processor system. The column labeled f gives the number of fully-productive page-ins that occur, and the column labeled $(N - 1)\times f$ indicates the minimum number of TLB reloads that are generated by the fully-productive page-ins. If the physical memory is smaller than the virtual memory, the number of fully-productive page-ins differs for different size systems. Accordingly, the value of f for each size system is included in these tables. For the Simple experiments, the correlation between $R_N - R_1$ and $(N - 1)\times f$ was measured by linear regression analysis. The correlation coefficient for the Simple/33%/64 TLBEs experiment is .999902. For the Simple/50%/64 TLBEs experiment, it is .999859.

8.3.1.2. Peskin Experiments

The Peskin program contains parallel programming constructs that could cause the number of related processor-based TLB reloads to grow linearly with N. But, as mentioned in the introduction to this section, the results of the Peskin experiments do not illustrate a linear growth of R_{Val} with N. This growth may be delayed beyond the range of N or the TLB sizes of our experiments.

Although data provided by our Peskin experiments indicates that R may grow linearly with N for larger N, the data are not sufficient to conclude that R grows linearly with N. This can be seen by examining the corresponding entries in Table 11 and Graphs 3 and 4.

Experiment Set (Graph No.)	TLBEs	Equation for R	r^2
Simple/33% (1)	16	$6130.260N + 905757.9$	.999621
	32	$6108.721N + 904831.1$	.999629
	64	$6093.506N + 899240.1$	.999648
Simple/50% (2)	16	$5013.867N + 1180031.0$	.999546
	32	$5304.050N + 1102218.0$	.999615
	64	$5269.707N + 1102032.0$	.999625
Peskin/33% (3)	16	$1629.723N + 838583.2$	.884066
	32	$2307.628N + 814469.7$	.885555
	64	$1935.908N + 775704.2$	.896055
Peskin/50% (4)	16	$1386.900N + 843622.7$	.682807
	32	$1151.763N + 825393.7$	.509499
	64	$516.6518N + 793133.6$	.129367

Table 11. Regression Equations for Validation's R

As shown, the values of r^2 are about .89 for the Peskin/33% experiment and between .13 and .68 for the Peskin/50% experiment.

In the case of the Peskin experiments, fully-productive page-ins possibly explain only a small percentage of the growth of R_{Val}. Supporting data for the computational sections of the Peskin/33%/64 TLBEs and Peskin/50%/64 TLBEs simulations are presented in Table 16 and Table 17. For these experiments the correlation between $R_N - R_1$ and $(N-1)\times f$, measured by linear regression analysis, is low. For the Peskin/33%/64 TLBEs experiment, the correlation coefficient is .775199. For the Peskin/50%/64 TLBEs experiment, it is .179803. These low correlations are partially due to the majority of the page-ins referenced by more than one processor being referenced by two or three processors, instead of N. (Of course, even the page-ins referenced by more than one processor can result in R growing linearly with N.) Supporting data is presented in Table 15 and Graphs 11 and 12, which shows that, like the Simple experiments, the majority of page-ins are referenced by only one processor.

8.3.2. Validation's TLB-reload Overhead

As shown in Graph 13, in a system with processor-based TLBs, C_{reload} (the contention-free cost of a TLB reload) grows as $\log N$. Thus, if R remains constant as N increases, the TLB-reload overhead increases as $\log N$. If R increases linearly with N, the TLB-reload overhead increases asymptotically as $N \log N$. This is because as N increases, its cost is dominated by the cost of network transit, which is $O(\log N)$.

As explained in Chapter 5, Validation's TLB-reload overhead is defined by the following equation:

$$OH = S \times t_{stale} + R \times T_{reload} ,$$

Number of Referencing Processors	Percentage of Page-Ins Referenced						
	$N=1$	$N=2$	$N=4$	$N=8$	$N=16$	$N=32$	$N=64$
1	100%	98%	98%	98%	98%	98%	98%
2 through $N-1$ for $N>2$			.4% (15%)	.4% (20%)	.4% (20%	.5% (21%)	.5% (20%)
N for $N>1$		2% (100%)	1.6% (85%)	1.6% (80%)	1.6% (80%)	1.5% (79%)	1.5% (80%)

Table 12. Analysis of Page-Ins Referenced by Processors—Simple/33%/64 TLBEs

where S is the number of stale TLB entries used by the N processors during program execution, t_{stale} is the average time for one round trip through the interconnection network (which is required when stale translation information is transmitted with a memory request), R is the number of TLB reloads, and T_{reload} is the average processor time expended in an N-processor system to execute the TLB-reload routine associated with Validation. T_{reload} and t_{stale} grow as $\log N$. Our simulations confirm that R_{Val} sometimes grows as $O(N)$ and as indicated below, our simulations show that S is inconsequential.

As described in Chapter 7, in a system that adopts Validation, a TLB reload costs 18 cycles plus four network requests for single-word loads. If a miss is triggered by the use of stale translation information, the cost is increased by one network request. Our simulator effectively makes T_{reload} a little more expensive (by the cost of one network request). In our simulations of Validation systems a TLB reload is considered to satisfy a TLB miss triggered by the use of stale translation information. Instead, since our simulator can detect whether an entry is stale before using it, we could have varied T_{reload} depending upon whether a reload was caused by the use of a stale TLB entry or not.

A network request for a single-word load costs $2\log_2 N + 2 + 1$ cycles, plus the waiting time incurred as a result of contention at the switches or at memory. The $2\log_2 N$ cycles represent network transit. The network is packet switched, network requests are pipelined, and both single-word load requests and responses each consist of two packets. Accordingly, one cycle represents the pipelining of the second packet of a single-word load response. The other two cycles represent memory access time.

Our simulator models network waiting time but models a composite of TLB and memory contention only in the case of Memory-based TLBs. As described in Chapter 7, to model the network waiting time, we use the formula of Kruskal, Snir, and Weiss [1986] for the processor-to-memory direction and a formula of Liu [1990] for the memory-to-processor direction. We assume a 2×2 switch size and a variable number of packets per message.

Data presented in Table 18 and Graphs 5–8 show that Validation's TLB-reload overhead, OH_{Val}, grows logarithmically with N (for $N=2^i$, $i=0, \ldots, 6$). In addition, they show that the lines defined by the regression equations presented in Table 18, which characterize this logarithmic relationship, fit the experimental data very well. In Table 18 the number of the graph that illustrates the simulation results and the related regression lines associated with a set of experiments is given in parentheses under the caption that describes the set of experiments. The regression equations are of the form $y = a \log x + b$, where $y = OH_{Val}$ and $x = N$. The logarithmic growth of OH_{Val} with N is verified by the values of r^2 all being greater than .97. Graphs 5 and 6 depict the results of the Simple simulations, and Graphs 7 and 8 depict the results of the Peskin simulations. For the Simple/33% experiments, the

N	*R*	$R_N - R_1$	*f*	$(N-1)\times f$
1	421961		386528	
2	432238	10277	6925	6925
4	447730	25769	6374	19122
8	472610	50649	6026	42182
16	521073	99112	6006	90090
32	617672	195711	5985	185535
64	810206	388245	5977	376551

Table 13. Analysis of Increase of R—Simple/33%/64 TLBEs—Computational Section Only: In this table *f* represents the number of fully-productive page-ins during the simulation of the computational section of the program trace. Correlation coefficient for $R_N - R_1$ vs. $(N-1)\times f$ is .999902.

N	*R*	$R_N - R_1$	*f*	$(N-1)\times f$
1	624374		382199	
2	633616	9242	4651	4651
4	647481	23107	3866	11598
8	668355	43981	3751	26257
16	710174	85800	3751	56265
32	793864	169490	3742	116002
64	960549	336175	3733	235179

Table 14. Analysis of Increase of R—Simple/50%/64 TLBEs—Computational Section Only: In this table *f* represents the number of fully-productive page-ins during the simulation of the computational section of the program trace. Correlation coefficient of $R_N - R_1$ vs. $(N-1)\times f$ is .999859.

data points for the experiments with different size TLBs vary so little that they can hardly be differentiated from one another. This is also true for the regression lines associated with these experiments. For the Simple/50% experiments with 32- and 64-entry TLBs, the data points, as well as the regression lines, are essentially coincident and, thus, can not be differentiated.

As illustrated in Graphs 5–8, $S\times t_{stale}$ contributes little or nothing to OH_{Val}. Here $S\times t_{stale}$, the time expended as a result of using stale translation information to generate memory requests, is represented by *Stale Time*. The Simple experiments indicate that no stale entries are used and the use of stale entries in the Peskin experiments is rare.

Since network contention is small and stale translation information is used rarely, OH_{Val} is well approximated by our model of processor-based TLB-reload overhead, which estimates the overhead by $R\times C_{reload}$, where *R* is as reported by our simulations. The agreement between the results of our Simple experiments and our model is illustrated in Graphs 14 and 16. Graphs 14 and 16 indicate that, within the range of *N* for our experiments, this is true whether C_{reload} is defined as the cost of a TLB reload triggered by the use of stale translation information or a miss triggered by the absence of an entry for the referenced page.

Number of Referencing Processors	Percentage of Page-Ins Referenced						
	$N = 1$	**$N = 2$**	**$N = 4$**	**$N = 8$**	**$N = 16$**	**$N = 32$**	**$N = 64$**
1	100%	71%	69%	67%	66%	66%	65%
2		29% (100%)	22% (72%)	22% (66%)	22% (64%)	22% (65%)	22% (65%)
3			6% (19%)	7% (21%)	7% (20%)	7% (21%)	7% (20%)
4			3% (9%)	2% (6%)	2% (7%)	2% (7%)	3% (7%)
5 through $N - 1$ for $N > 4$				.5% (1%)	1% (3%)	1% (4%)	1% (4%)
N for $N > 4$				2% (6%)	2% (6%)	1% (3%)	1% (4%)

Table 15. Analysis of Page-Ins Referenced by Processors—Peskin/33%/64 TLBEs

(Graph 13 compares these two definitions of C_{reload}.) The lines labeled *Simulated Validation-All Stale* and *Validation-No Stale* illustrate the difference in the agreement between our experimental results (*Simulation Trend*) and our model.

8.3.3. Simulation Results vs. Simulation Model

As discussed in Section 8.3.1, the Validation/Simple experiments evidence R growing linearly with N. However, as discussed above and as illustrated in Graphs 14 and 16, instead of OH_{Val} growing as $N \log N$, it grows as $\log N$. This apparent inconsistency between our simulation results and our model is due to the coefficients present in the equations that describe the growth of R_{Val} delaying the onset of the asymptotic $N \log N$ growth rate of OH_{Val} beyond $N = 64$. For the Simple experiments, $R_{Val} = aN + b$, where $b \neg a$. As a result, OH_{Val}, which is well approximated by $R_{Val} \times C_{reload}$, is approximately equal to $R \log N \approx b \log N$ for the modest values of N simulated.

We illustrate this phenomenon by examining the results of the Simple experiments for systems with 64-entry TLBs. Analogous results hold for the related 16- and 32-entry TLB experiments. We organize this section into two subsections, each one devoted to one experiment. In Section 8.3.3.1 we carefully examine the results of the Simple/33% experiment and show why the asymptotic $N \log N$ growth rate of OH_{Val} does not become apparent until $N = 128$. In Section 8.3.3.2, we leave out the detailed explanation since it is similar to the one given in Section 8.3.3.1 and, instead, present only the facts and figures.

Notice that speculative information is included in Graphs 14–17 for $N > 256$. Our Simple and Peskin experiments are driven by traces of programs that solve fixed-size problems. Therefore, a maximum available parallelism is defined for each program. In these graphs, we extrapolate beyond the maximum available parallelism, which is 256.

N	R	$R_N - R_1$	f	$(N-1)\times f$
1	690382		71261	
2	699862	9480	18951	18951
4	729170	38788	2308	6924
8	728796	38414	1568	10976
16	738096	47714	1442	21630
32	791739	101357	1385	42935
64	819050	128688	1342	84546

Table 16. Analysis of Increase of R—Peskin/33%/64 TLBEs—Computational Section Only: In this table f represents the number of fully-productive page-ins during the simulation of the computational section of the program trace. Correlation coefficient of $R_N - R_1$ vs. $(N-1)\times f$ is .775199.

N	R	$R_N - R_1$	f	$(N-1)\times f$
1	676077		2589	
2	704050	27973	2087	2087
4	746420	70343	1584	4752
8	748817	72740	1497	10479
16	759099	83022	1416	21240
32	769455	93378	1383	42873
64	734937	58860	1333	83979

Table 17. Analysis of Increase of R—Peskin/50%/64 TLBEs—Computational Section Only: In this table f represents the number of fully-productive page-ins during the simulation of the computational section of the program trace. Correlation coefficient of $R_N - R_1$ vs. $(N-1)\times f$ is .179803.

8.3.3.1. Simple/33%

Referring to Table 11, we see that the Simple/33% experiment with 64-entry TLBs shows that R_{Val} is well approximated by the equation: $6093.506\,N + 899240.1$. Assuming this growth, we estimate the growth of OH_{Val} by $R\times C_{reload}$, where

$$C_{reload} = 18 + 5\,(2\log_2 N + 3) = 33 + 10\log_2 N$$

or

$$C_{reload} = 18 + 4\,(2\log_2 N + 3) = 30 + 8\log_2 N\,.$$

The first equation approximates the contention-free cost of a TLB reload that satisfies a TLB miss triggered by the use of stale translation information, and the second equation approximates the reload cost otherwise. Whichever definition of C_{reload} is used, if R grows linearly with N, OH_{Val} grows asymptotically as $N \log N$.

Graph 14, which plots OH_{Val} versus $\log_{10}N$, shows that:

Experiment Set (Graph No.)	TLBEs	Equation for OH_{Val}	r^2
Simple/33% (5)	16	55706414 log N + 23481226	.975254
	32	55599574 log N + 23475710	.975322
	64	55325349 log N + 23305418	.975122
Simple/50% (6)	16	63168834 log N + 34028252	.988913
	32	60962068 log N + 31069553	.986205
	64	60845289 log N + 31103111	.986345
Peskin/33% (7)	16	40177991 log N + 26022986	.991695
	32	41913901 log N + 24122378	.995219
	64	39109473 log N + 23364348	.994715
Peskin/50% (8)	16	39952860 log N + 26434542	.992170
	32	38503194 log N + 26095587	.990336
	64	35374829 log N + 25596236	.983039

Table 18. Regression Equations for Validation's TLB-reload Overhead

- The extrapolation of our experimental data (*Simulation Trend*) indicates OH_{Val}'s growth as being $O(\log N)$.
- Our simulation model indicates that $OH_{Val} = R_{Val} \times C_{reload}$ (*Validation-All Stale* and *Validation-No Stale*) grows asymptotically as $N \log N$.
- The $N \log N$ growth rate of the modeled OH_{Val} becomes apparent at $N = 128$.

Graph 15 plots the same data for OH_{Val} versus N. In these graphs the line labeled *Simulation Trend* is the regression line that describes the trend indicated by our simulation results. The line labeled *Ideal Trend* represents the value of OH_{Val} obtained by assuming that R remains constant and $C_{reload} = 30 + 8 \log_2 N$

Our estimated overheads are of the form:

$$OH_{est} = R\,(c + d \log_2 N)\ ,$$

where, as in our simulations, we assume $c = 33$ and $d = 10$. We use the regression equation

$$R = aN + b\ ,$$

where, in this case, $a = 6093.506$ and $b = 899240.1$. Thus,

$$OH_{est} = (aN + b)\,(c + d \log_2 N)\ ,$$

which equals

$$caN + cb + daN \log_2 N + db \log_2 N.$$

As shown in Table 19, since $b \neg a$, the term in $\log_2 N$ dominates until $N = 128$. At $N = 256$, the term in $N \log_2 N$ begins to dominate.

N	*caN*	*daN* $\log_2 N$	*db* $\log_2 N$
1	2010862	0	0
2	402171	121870	8992401
4	804343	487481	17984802
8	1608686	1462441	26977203
16	3217371	3899844	35969604
32	6434742	9749610	44962005
64	12869485	23399063	53954406
128	25738969	54597814	62946807
256	51477939	1.2 E+08	71939208
512	1.0 E+08	2.8 E+08	80931609
1024	2.1 E+08	6.2 E+08	89924010

Table 19. Analysis of Validation's Estimated TLB-reload Overhead—Simple/33%/64 TLBEs

8.3.3.2. Simple/50%

Referring to Table 11, we see that the Simple/50% experiment with 64-entry TLBs shows that R_{Val} is defined well by the equation: $5269.707\,N + 1102032$. As in Section 8.3.3.1, we estimate the growth of OH_{Val} by $R \times C_{reload}$. Validation's estimated overheads, its simulation trend, and its ideal trend are illustrated in Graphs 16 and 17. These graphs are analogous to Graphs 14 and 15 for the Simple/33%/64 TLBEs experiment.

The reason why these experiments indicate that OH_{Val} grows as $\log N$, instead of $N \log N$, is the same as the one presented in Section 8.2.3.1. Therefore, rather than go through the same argument, below we merely define the coefficients of the terms in the equation

$$OH_{Val} = (aN + b)\ (c + d \log_2 N)\ ,$$

which equals

$$caN + cb + daN \log_2 N + db \log_2 N$$

and present Table 20, which lists the value of each of these terms as N increases. Using the regression equation for the Simple/50%/64 TLBEs experiment's R, $a = 5269.707$ and $b = 1102032$. The coefficient of N, ca, is 173900. The coefficient of $N \log_2 N$, da, is 105394. And, the coefficient of $\log_2 N$, db, is 11020320.

Analyzing Table 20, we see that Validation's estimated TLB-reload overhead for the Simple/50%/64 TLBEs experiment does not begin to grow as $N \log N$ until $N = 256$. Since $b \neg a$, the term in $\log_2 N$ dominates until $N = 128$. At $N = 256$, the term in $N \log_2 N$ begins to dominate.

N	*caN*	*daN* $\log_2 N$	*db* $\log_2 N$
1	173900	0	0
2	347800	105394	11020320
4	695601	421576	22040640
8	1391203	1264730	33060960
16	2782405	3372612	44081280
32	5564811	8431531	55101600
64	11129621	20235674	66121920
128	22259242	47216575	77142240
256	44518485	1.1 E+08	88162560
512	89036969	2.4 E+08	99182880
1024	1.8 E+08	5.4 E+08	1.1 E+08

Table 20. Analysis of Validation's Estimated Implementation Overhead—Simple/50%/64 TLBEs

8.4. Simulation Results: Read-locked TLBs

In this section we present the results of our simulations of systems employing Read-locked TLBs. First, we briefly describe the Read-locked TLBs method of ensuring TLB consistency and summarize our simulation results. Then, in the two subsections that follow, we present supporting data generated by our Read-locked TLBs experiments. Although Read-locked TLBs can be implemented in a system with memory-based TLBs, we assume it is implemented in a system with processor-based TLBs.

In a Read-locked TLBs system, TLB consistency is ensured by avoiding and postponing changes that may cause translation information cached in one or more TLBs to become inconsistent. To implement this idea, a counter associated with each resident page is incremented whenever translation information for the page is loaded into a TLB and decremented whenever a TLB entry for the page is invalidated or replaced. To ensure the integrity of a counter, the one-counter algorithm, described in Chapter 5, is used for its modification. A change to a page's translation information, that could cause TLB inconsistencies, is safe to make only if the page's counter is equal to zero. Thus, the Read-locked TLBs' reload routine is an enhanced version of our generic TLB-reload routine and, thus, has a higher cost. As N increases, T_{reload} is dominated by network transit, which grows as $\log N$. Thus, Read-locked TLBs' T_{reload}, like Validation's, is $O(\log N)$.

Read-locked TLBs does not require hardware support, but unlike Validation, does not solve the most general TLB consistency problem. This is because it cannot support all memory sharing patterns since, without other intervention, some page-table modifications may have to be postponed indefinitely. For example, suppose that a page-table modification is postponed because multiple TLBs contain the translation for a page . If we further suppose that only a subset of the entries in these TLBs are being used, then the entry in question may not be replaced for a long time.

We report the results of fewer simulations in the case of Read-locked TLBs, as compared to Validation, because the simulation of certain Read-locked TLBs systems can result in deadlock. This is because our simulations, like our model, do not account for the overhead due to deadlock recovery for two reasons:

1. Real systems would be designed to prevent deadlock.
2. Including deadlock recovery overhead would bias our performance results because performance would be correlated with deadlock recovery.

As a result of not modeling deadlock recovery for Read-locked TLBs systems, as the TLB size increases, the number of simulations that comprise an experiment decreases. As described in Section 5.2.2, deadlock can occur in systems where the sum of the total number of TLB entries plus the number of frames that can be allocated simultaneously to nonresident pages is not less than the number of page frames.

The performance of Read-locked TLBs, like that of Validation, depends on N and R. If R grows linearly with N, TLB-reload overhead grows asymptotically as $N \log N$.

Our simulation study indicates that:

1. As discussed in Section 8.4.1, in general, no more page-ins are generated in Read-locked TLBs systems than are generated in Validation or Memory-based TLBs systems.
2. The Read-locked TLBs/Simple experiments, discussed in Sections 8.4.2 and 8.4.3, indicate that:
   - Read-locked TLBs' R (R_{Lock}) is almost identical to R_{Val},
   - Read-locked TLBs' TLB-reload overhead (OH_{Lock}) grows as $O(\log N)$, and
   - an apparent inconsistency exists between our simulation results and our model with respect to OH_{Lock}.
3. The Read-locked TLBs/Peskin experiments, like the Validation/Peskin experiments, provide data for our experiments' range of N that does not evidence a linear growth of R with N. OH_{Lock}, like OH_{Val}, grows as $\log N$.
4. In general, $OH_{Val} < OH_{Lock}$.

With the exception of the first result, the explanations of the other results are similar to those with respect to the Validation experiments, which are presented in Section 8.3. This is because T_{reload} for processor-based TLBs is $O(\log N)$ and the growth of R_{Lock} is similar to that of R_{Val}. Accordingly, this section merely presents the data provided by the Read-locked TLBs experiments. To illustrate the similarities between the Read-locked TLBs and Validation experiments, the graphs discussed in this section present data provided by both Read-locked TLBs and Validation experiments.

8.4.1. Read-locked TLBs and Paging Activity

As Graphs 18–21 illustrate, in general, no more page-ins are generated in Read-locked TLBs systems, than are generated in Validation or Memory-based TLBs systems. One might expect otherwise, especially in comparison to Validation systems, since in Read-locked TLBs systems a page cannot be evicted if a copy of its translation information is cached in any other TLB except that of the processor evicting the page. This restriction causes the related page-replacement policy employed by our Read-locked TLBs experiments to be a modified LRU (least recently used) policy, rather than a strict LRU policy, which is what we employ in our Validation experiments. Graphs 18–21 plot the number of generated page-ins vs. N for Validation, Read-locked TLBs, and Memory-based TLBs experiments. In these graphs, the number of page-ins for Validation and Memory-based TLBs systems with 64-entry TLBs are plotted along with the number of page-ins for all the related Read-locked TLBs experiments. As indicated, a comparable or smaller number of page-ins are generated by Read-locked TLBs systems, as compared to Validation or Memory-based TLBs systems. Of course, these results are inconclusive as to whether the page-replacement policy adopted by

Read-locked TLBs results in less paging activity than a strict LRU policy, but it gives cause for further investigation.

8.4.2. *Read-locked TLBs' R*

The Simple experiments for both Read-locked TLBs and Validation report values of R that are almost identical. Thus, in these cases, R_{Lock}, like R_{Val}, grows linearly with N. The same growth is indicated whether we examine data collected during entire simulations or during the computational section of simulations (see Section 8.3.1). Like the Validation/Peskin experiments, the data provided by the Read-locked TLBs/Peskin experiments, for the experiments' range of N, do not evidence a linear growth of R with N.

Graphs 22–25 compare R_{Lock} and R_{Val}, and show that for the Simple experiments (see Graphs 22 and 23) R_{Val} is essentially indistinguishable from R_{Lock}. In Graphs 24 and 25, R_{Lock} is represented by experimental data and R_{Val} is represented by the regression lines that fit analogous experimental data. In contrast, for the Peskin experiments, for which we are not able to get a good linear fit to the data, Graphs 24 and 25 represent R_{Val} by the experimental data provided by the analogous Validation experiments.

8.4.3. *Read-locked TLBs' Reload Overhead*

In the case of the Simple experiments, although R_{Val} and R_{Lock} are almost identical, OH_{Val} and OH_{Lock} differ somewhat. This is because, as shown below, the definition of T_{reload} is different. As explained in Chapter 5, Read-locked TLBs' reload overhead is defined by

$$OH = R \times T_{reload.}$$

Since our Validation experiments indicate that TLB reloads are rarely triggered by the use of stale translation information, both OH_{Lock} and OH_{Val} are defined by the same equation.

Validation's TLB reload cost is lower than that of Read-locked TLBs. As defined in Chapter 7, Read-locked TLBs' T_{reload} is 24 cycles plus five network requests for single-word loads. For our simulations, Validation's is 18 cycles plus five network requests for single-word loads. (The different TLB-reload costs are compared in Graph 13.)

Thus, it should not be surprising that:

1. OH_{Val} is noticeably smaller than OH_{Lock} in all but the Peskin/33% experiments.
2. Like the Validation/Simple experiments, the Read-locked TLBs/Simple experiments show that OH_{Lock} grows as $\log N$, although it has an asymptotic growth of $O(N \log N)$.

Graphs 26 and 27 compare OH_{Lock} and OH_{Val} for the Simple experiments, while graphs 28 and 29 compare OH_{Lock} and OH_{Val} for the Peskin experiments. Except in the case of the Peskin/33% experiment, OH_{Lock} and OH_{Val} grow as $\log N$, as was demonstrated for OH_{Val} in Section 8.3.3.

8.5. Simulation Results: Memory-based TLBs

In this section we present the results of our simulations of systems employing Memory-based TLBs. First, we briefly describe the Memory-based TLBs method of ensuring TLB consistency and summarize our simulation results. Then, in the two subsections that follow, we present supporting data generated by our Memory-based TLBs experiments.

As described in Chapter 5, for memory-based TLBs the contention-free cost of a TLB reload, C_{reload}, is very small (5 or 7 cycles) and remains constant as N increases. This is because, unlike C_{reload} for processor-based TLBs, it does not depend on the cost of network transit. A TLB is located at each memory module, rather than each processor. Consequently, address translation and TLB reloads take place at memory. Memory modules are organized into clusters called paging arenas, where the mapping of a page is fixed to a specific arena. A bus interconnects the TLBs associated with a memory cluster and the cluster page table (which has an entry for each page that may map to a frame in the cluster). To ensure the integrity of a cluster's page table and to ensure TLB consistency, a cluster's page table is accessible only via the cluster's bus, and a TLB is designed similar to a snoopy cache so that it monitors all cluster bus transactions. The bus acts as a lock on a cluster's page table. A page-table modification triggers the TLBs of the associated paging arena to update or invalidate TLB entries for the related page. (Recall that a page's translation information can be in as many as C TLBs, where C is the number of memory modules in a paging arena.) A TLB miss generates a bus request for the referenced page's translation information, and the request is satisfied by either another TLB in the cluster or the cluster's page-table memory. Network transit is required only to inform a processor about an exception caused by a memory reference. Contention for TLBs and cluster buses strongly determines the average cost of a TLB reload.

Recall that when we compare the performance of our three solutions we refer to them as two methods of ensuring TLB consistency: processor-based TLBs and memory-based TLBs. This is because, as shown in Section 8.3 and 8.4, Validation and Read-locked TLBs, our two solutions for processor-based TLBs, have similar performance characteristics. Even though memory-based TLBs in systems with multiple paging arenas may outperform processor-based TLBs, note that Validation solves the most general TLB consistency problem, whereas Memory-based TLBs does so only in systems with one paging arena. This is because pages that map to the same frame must map to the same memory cluster.

For the studied workloads and within the range of N for our experiments, our results show that:

1. The performance of memory-based TLBs depends on the number of paging arenas. As shown in Section 8.5.1, for a particular N and fixed physical memory and TLB sizes, as the number of paging arenas increases, Memory-based TLBs R (R_{Mem}) decreases. As a result, as shown in Section 8.5.2, contention for TLBs and cluster buses decrease and, therefore, TLB-reload overhead decreases.
2. Processor-based TLBs outperform memory-based TLBs in systems with one paging arena; memory-based TLBs outperform processor-based TLBs in systems with two paging arenas.

8.5.1. Memory-based TLBs' R

With respect to Memory-based TLBs' R within the range of N for our experiments, our study shows that:

1. For systems with one or two paging arenas, R_{Mem} grows sublinearly with N and grows monotonically with N, at least for $N > 2$.
2. R_{Mem} decreases as the TLB size or the number of paging arenas increases.
3. R_{Mem} for systems with one paging arena and $4N$-entry TLBs is comparable to R_{Val} (for systems with 64-entry TLBs), R_{Mem} for systems with one paging arena is generally larger than R_{Val}, and R_{Mem} for systems with two paging arenas is generally smaller than R_{Val}.

All Validation experiments are for systems with one paging arena.

Graphs 30–33 illustrate the growth of R_{Mem} as N increases, and compare R_{Mem} to R_{Val}. Each of these graph includes: the experimental results for Memory-based TLBs systems with one paging arena and 16-, 32-, and 64-entry TLBs, the experimental results for Memory-based TLBs systems with two paging arenas and 64-entry TLBs, and the regression lines or data points that define R_{Val} for analogous experiments with one paging arena and 16- and 64-entry TLBs. (For the Validation/Simple experiments, we illustrate the regression lines defined by the equations presented in Table 11 of Section 8.2. For the Validation/Peskin experiments, we illustrate the resulting data points, since we did not get a good linear fit for these experiments.) In addition, Graphs 30 and 31, which illustrate results of the Simple experiments (Graphs 32 and 33 illustrate results of the Peskin experiments), include experimental results for Memory-based TLBs systems with one paging arena and $4N$-entry TLBs.

8.5.1.1. Monotonic Increase of R with N

First we consider our Memory-based TLBs experiments with fixed-size TLBs. These include Simple experiments for $N = 2^i$, $i = 0, \ldots, 7$ and Peskin experiments for $N = 2^i$, $i = 0, \ldots, 6$. For these experiments, the essentially monotonic growth of R_{Mem} with N, indicated in Graphs 30–33, is likely due to two factors:

- C increases with N and a page's translation information can be replicated at as many as C TLBs.
- TLB-entry thrashing, which occurs in memory-based TLBs due to multiple processes accessing shared TLBs, may increase with N.

Note that the number of TLBs (of the same size) and, thus, the number of TLB entries increase linearly with N.

R_{Mem} clearly does not grow linearly with N. But, as the regression lines in Graphs 30–33 indicate, for larger N this may be the trend. The data provided by our experiments is not sufficient to allow us to come to this conclusion.

8.5.1.2. R Decreases as TLB Size or Number of Paging Arenas Increase

As shown in Graphs 30–33, for a particular N, R_{Mem} decreases as the TLB size or the number of paging arenas increases. The decrease with an increasing TLB size is likely due to less interprocessor and intraprocess TLB-entry thrashing. The decrease in R_{Mem} as the number of paging arenas increases is due to two factors:

1. As the number of paging arenas increases, the replication of translation information across the TLBs of a cluster decreases. Assuming spatial locality of reference, multiple words of a page are referenced during a specific time interval. Therefore, an average of $O(c)$, $1 \le c \le C$, TLB reloads are generated for each page-in. As the number of paging arenas increases, c decreases and so does the replication factor.
2. As the number of paging arenas increases, the number of frames per arena decreases and the number of pages that map to an arena decreases. Accordingly, a TLB is used to cache translation information for a decreasing number of pages and, thus, TLB-entry thrashing is likely to decrease.

At the extreme, preliminary simulations for $N = 2^i$, $i = 0, \ldots, 4$, indicate that R_{Mem} for systems with N paging arenas decreases with N. In this case the memory-based TLBs constitute one, large, shared TLB with no replication of translation information.

As illustrated in Graphs 18–21, in addition to the fact that R_{Mem} decreases as the number of paging arenas increases, 17 out of 24 simulations indicate that the number of page-ins generated by systems with two paging arenas is less than the number generated by systems with one.

8.5.1.3. Comparison with Validation's R

Although we believe that the replication of translation information at memory-based TLBs is the major reason for R's growth as N increases, we ran an experiment to determine the effect on R's growth of TLB-entry thrashing due to multiple processes accessing the same shared TLBs. Our hypothesis is: If the TLB size grows linearly with N, more of the context of each processor is cached. Accordingly, R_{Mem} would decrease and be made comparable to R_{Val}. We ran experiments using the Simple trace that confirm this hypothesis. Our corresponding Simple experiment, for $N = 2^i$, $i = 2, \ldots, 7$, assumes a $4N$ TLB size. (Of course, we do not suggest such hardware, since it grows quadratically in N.) As illustrated in Graphs 30 and 31, for both the Simple experiments, R_{Mem} is comparable to R_{Val} and, thus, grows linearly with N.

In the case of the Simple experiments, for a fixed TLB size, R_{Mem} for systems with one paging arena is smaller than R_{Val} for $N <$ TLB size/2. In the case of the Peskin experiments, R_{Mem} is smaller than R_{Val} only for $N = 1$. R_{Mem} generally being larger than R_{Val} is due to replication of translation information among multiple TLBs and TLB-entry thrashing due to multiple processes accessing the same shared TLBs.

Since an increase of the number of paging arenas leads to a decrease in the replication of translation information and TLB-entry thrashing, it is not surprising that R_{Mem} for systems with two paging arenas is smaller than R_{Mem} for systems with one paging arena. In addition, R_{Mem} for systems with two paging arenas approaches R_{Val} from below. Unlike memory-based TLBs, there is no reason to believe that multiple paging arenas would decrease the number of processor-based TLB reloads.

8.5.2. Memory-based TLBs' Reload Overhead

Our study shows that, like R_{Mem}, Memory-based TLBs' reload overhead (OH_{Mem}) for a specific N decreases as the TLB size or the number of paging arenas increases. In addition, OH_{Mem} for the studied workloads and within the range of N for our experiments, grows more than linearly with N.

Even though Memory-based TLBs' C_{reload} is small and remains constant as N increases, there are other components of T_{reload} for the modeled hardware that are variable and depend on R, and thus on N, and the number of paging arenas:

- $T_{pgexcpt}$, the network transit time related to informing processors about exceptions caused by the memory references that they generated,
- $T_{\text{busy-TLB}}$, the time that memory references are delayed while waiting for TLBs that are busy servicing TLB misses, and
- $T_{\text{busy-bus}}$, the time that TLB-miss servicing is delayed as a result of a busy cluster bus.

Memory-based TLBs reload overhead is expressed as

$$OH_{Mem} = R_1 C_{reload_1} + R_2 C_{reload_2} + T_{\text{busy-TLB}} + T_{\text{busy-bus}} + T_{pgexcpt},$$

where R_1 and C_{reload_1} are the number of TLB reloads serviced by TLBs in a cluster and the contention-free cost of each, and R_2 and C_{reload_2} are the number of TLB reloads serviced by a cluster's page-table memory and the contention-free cost of each. The total number of reloads is referred to as $R_{Mem} = R_1 + R_2$.

As we show below, essentially all the components of OH_{Mem} decrease as R_{Mem} decreases. (As shown in the previous section, R_{Mem} decreases as the TLB size or the number of paging arenas increases.) Thus, as R_{Mem} decreases, OH_{Mem} decreases.

Our simulator maintains a cumulative record of each component of OH_{Mem} and our results, which are illustrated in Graphs 34–37, indicate that:

1. $T_{pgexcpt}$ grows logarithmically with N. This is because one network round trip is required per exception. In our experiments the number of exceptions, which is equal to the number of page-ins, remains fairly constant as N increases. As shown in Graphs 34–37, $T_{pgexcpt}$ is comparable in systems with one and two paging arenas. This is because, as indicated in Graphs 18–21, the associated experiments, with the exception of the Peskin/33% experiments, report a comparable number of page-ins.
2. In contrast, as illustrated in Graphs 34–37 and Graphs 38–41, $T_{\text{busy-TLB}}$ and $T_{\text{busy-bus}}$ increase with N and decrease as the number of paging arenas increases. This is expected: As N increases, so does R, which means that TLBs are busy more and cluster buses are used more frequently. As the number of paging arenas increases, R_{Mem} decreases, which means that TLBs are busy less and cluster buses are used less frequently.
3. Since R_{Mem} decreases as the number of paging arenas increases, so does $R_1 \times C_{reload_1} + R_2 \times C_{reload_2}$. The contention-free cost of a TLB reload is smaller if the corresponding request for translation information is serviced by another TLB. As indicated in Graphs 42–45, R_2, the number of TLB reloads serviced by page-table memory decreases as N increases. This means that as N increases, the contention-free cost of reloads decreases. The decrease in the number of reloads serviced by the page-table memory is at first dramatic, but at a certain N it levels off. The leveling off occurs because at different points in a program the context changes. As a result, no matter how many TLBs are in a cluster, the translation information for some pages will not be TLB resident. Even if the TLBs are large enough to store the translation information for all pages mapped to their clusters, each page's translation information must be loaded once.

Like R_{Mem}, OH_{Mem} decreases as the TLB size or the number of paging arenas increases. This is confirmed by our simulations, as illustrated in Graphs 46–49. Note that the disparity between OH_{Mem} for systems with two paging arenas and for systems with one paging arena increases with N.

The experiments for systems with multiple paging arenas include preliminary simulations ($N \le 2^i$ with $i = 1, \ldots, 4$) of systems with N paging arenas. As expected, these simulations also indicate that memory-based TLBs perform better as the number of paging arenas increases. Note, however, that the following caveats apply:

1. There is no cluster bus contention in such a system, however, because words of a page are not interleaved among multiple memory modules, network and memory contention may increase with N and I/O performance may be compromised. Our simulation results support the assertion that memory contention increases as N increases.
2. Memory-based TLBs in a system with multiple paging arenas does not solve the most general TLB consistency problem.

Although initially OH_{Mem} appears to grow as $\log N$, it quickly grow more than linearly with N. Unlike our processor-based TLB experiments, for which we have a model for TLB-reload overhead, we are not able to characterize the growth of OH_{Mem} as N increases.

8.5.2.1. Comparison with Processor-based TLBs

As illustrated in Graphs 46–49, processor-based TLBs outperform memory-based TLBs in systems with one paging arena and memory-based TLBs outperform processor-based TLBs in systems with two paging arenas.

Comparing Graphs 46–49 and Graphs 30–33, which illustrate the growth of R_{Mem} and R_{Val} for the same experiments, we see that $R_{Mem} \geq R_{Val}$ does not necessarily mean that $OH_{Mem} \geq OH_{Val}$. Thus, memory-based TLBs can tolerate a higher TLB miss-rate than processor-based TLBs and still have a lower TLB-reload overhead.

Although our results indicate that memory-based TLBs perform better than processor-based TLBs, provided that the number of paging arenas grows as $O(N)$, the caveats mentioned at the end of the previous section apply.

8.5.2.2. How to Improve the Performance of Memory-based TLBs

As illustrated in Graphs 34–37 and Graphs 38–41, our simulations show that as N increases, $T_{\text{busy-TLB}}$ and $T_{\text{busy-bus}}$ become more significant in terms of their effects on OH_{Mem}. To reduce their effects we suggest the following:

1. Memory-based TLBs should be designed as lock-up free caches [Kroft, 1981], instead of as blocking caches, as assumed by our simulator.
2. A memory-based TLB subsystem should include a buffer to store memory requests that generate TLB misses that are awaiting service.
3. Memory-based TLBs should be designed to snatch from their cluster buses needed translation information requested by other TLBs.

As explained in Chapter 5, we assume a blocking TLB and our simulator does not provide buffering for TLB misses that are awaiting service. Without buffering, a TLB is considered busy until an outstanding TLB miss is serviced. Thus, a TLB miss blocks the memory-network interface from receiving a memory request from the network until the miss is serviced.

Assuming our first two suggestions, references that generate TLB hits would be processed and TLB misses would be queued, if necessary, for servicing. Note, however, that if TLB misses are buffered then a TLB subsystem must be sophisticated enough to recognize subsequent misses on a page for which the translation information has already been loaded into the TLB. In order to get a sense of the effect of employing a lock-up free TLB, in Graphs 50–53 we present $OH_{Mem} - T_{\text{busy-TLB}}$, which we label as *OH Mem w/o Busy-TLB Time*. In this case, the estimate of OH_{Mem} is somewhat inaccurate because some of the time recorded in $T_{\text{busy-TLB}}$ should be added to $T_{\text{busy-bus}}$ and, thus, increase *OH Mem w/o Busy-TLB Time*. Unfortunately, we do not have enough data to determine the correct amount to add.

Care must be taken in making such a design change if sequential consistency is to be ensured. Memory references from the same processor to the same memory module could be serviced in a different order than they were generated. Of course, in addition, upon loading the translation information of a page, after processing the reference that initiated the reload, references to that page that generated TLB misses and are queued for service must be processed in the order received.

In our simulator a TLB monitors the cluster bus for changes to and requests for translation information. Suppose we implemented our third suggestion. Then a TLB also monitors the cluster bus for responses to requests for translation information from other TLBs. In this case, if a related response provides translation information that would satisfy queued TLB misses, a TLB could load that translation information and process the associated memory references. As N increases and more pages are shared, this design change may make more sense. For a particular TLB, all the other TLBs in its cluster form a second-level cache of page-table entries. Accordingly, as Graphs 42–45 illustrate, we find that as N increases the number of TLB reloads satisfied by other TLBs increases.

8.6. TLB Shootdown vs. Our Solutions

Although we did not simulate systems employing TLB Shootdown, for the workloads studied that cause R to grow linearly with N, we give evidence that TLB Shootdown would have a significantly higher overhead than our solutions for these workloads when N is large. This comparison is based on estimates of TLB Shootdown's overhead and simulation data that defines TLB-reload overhead and the number of invocations of TLB Shootdown.

In our comparison we assume Rosenburg's modified TLB Shootdown algorithm [1989], which he showed to have better performance than the original (more generally applicable) algorithm of Black, *et al.* [1989]. TLB Shootdown is implemented by invoking an algorithm at each page-table modification that may cause TLB inconsistencies. TLBs are assumed to be located at processors. The algorithm interrupts the execution of each processor that is executing a process that has access to the subject page table. To compare the performance of TLB Shootdown to that of our solutions, it is necessary to include the cost of TLB reloads in TLB Shootdown's overhead. This is because for our solutions the cost of ensuring TLB consistency is embodied in the cost of TLB reloads.

To compare TLB Shootdown's overhead with that of our solutions, we define TLB Shootdown's overhead as

$$OH_{Shoot} = R \times C_{reload} + \beta \times T_{Shoot},$$

where R is the number of TLB reloads generated in a system with processor-based TLBs, C_{reload} is the contention-free cost of a generic TLB-reload routine (which does not ensure TLB consistency), β represents the number of invocations of the TLB Shootdown algorithm, and T_{Shoot} is our estimated cost of an invocation of the algorithm.

Our simulation study provides us with data to define R, C_{reload}, and β:

- Assuming a 64-entry TLB, we define $R = R_{Val}$, where R_{Val} represents data provided by the Validation/Simple experiments. Since in this case R_{Val} grows linearly with N, we use the regression equation for R_{Val}, which fits the experimental data very well. This seems a reasonable choice since a TLB Shootdown system would likely generate the same number of TLB reloads as a Validation system.
- Our model for the TLB-reload overhead of processor-based TLBs, $R \times C_{reload}$, well approximates our experimental results. Thus, $R_{Val} \times C_{reload}$ is a good approximation of the TLB-reload overhead component of OH_{Shoot}.
- In our simulations only page evictions cause the TLB consistency problem to arise. Since the TLB Shootdown algorithm is invoked for each page-table modification that may cause TLB inconsistencies, β is defined to be the number of page evictions, which is equal to the number of page-ins less the number of frames. The number of page-ins are reported by our simulations. Since TLB inconsistencies can arise as a result of other page-table modifications (see Chapter 2), β represents a lower bound.

Recall that since C_{reload} for processor-based TLBs grows as $\log N$, if R grows linearly with N then $R \times C_{reload}$ has an asymptotic growth of $O(N \log N)$. The cost of the generic TLB-reload routine is

$$C_{reload} = 9 + 2 \times (2 \log_2 N + 3),$$

i.e., 9 cycles plus the cost of two single-word loads.

As was shown in Chapter 4, without including the cost of generic TLB reloads, TLB Shootdown's overhead can be expressed as $\beta(T_{change} + P \times T_{participate})$, where T_{change} represents the average time, both busy and idle, expended by a processor making a page-table modification that could cause TLB inconsistencies to arise; $T_{participate}$ represents the average time expended by each processor that also participates in the algorithm, and P is the average number of participating processors.

As described below, T_{Shoot} approximates $T_{change} + P \times T_{participate}$. We assume that $P = N$ because in the SPMD programming model and, thus, in our simulations all shared pages are accessible to all N processors. Our estimate of $T_{participate}$:

$$3 \times (2\ \log_2 N + 3) + 100,$$

where $2\ \log_2 N + 3$ is the cost of a single-word load or store, also grows as $N \log N$. We do not account for the time spent by a processor making a page-table modification in our simulations, therefore, we do not include it in our estimate of T_{Shoot}. Thus, we underestimate the number of required network requests. Our estimate of T_{Shoot} concentrates on those components of the algorithm that increase with N, *i.e.*, the number of processors participating in the algorithm and the cost of network transit, which is $O(\log N)$.

Since we assume that N processors participate in the algorithm, an invocation of the algorithm by one of these processors interrupts $N - 1$ processors. Upon taking the interrupt, each of the $N - 1$ processors accesses its requests counter and the related posted message, which indicates the TLB actions to perform. These accesses are both single-word loads. After performing the stipulated TLB-entry invalidations, each of the $N - 1$ processors sets its update counter (a single-word store), which notifies the modifying processor that it has completed its part of the algorithm. After this, an interrupted processor resumes execution. The three accesses of each of the interrupted processors account for $(N - 1) \times (3 \times (2 \log N + 3))$ cycles. The cost of servicing these interrupts and performing the necessary TLB actions to ensure TLB consistency is estimated at $(N - 1) \times 100$ cycles. Even though network requests are generally required to service an interrupt, we do not include them in this estimate. In both cases, we multiply by N, instead of $N - 1$, to account for a portion of the modifying processor's time spent interrupting processors and waiting for the interrupted processors to do their assigned work. With broadcast capability the cost of interrupting N processors is $O(\log N)$, otherwise it is $O(N \log N)$.

As illustrated in Graphs 54 and 59, our comparison indicates that our solutions outperform TLB Shootdown. OH_{Shoot} clearly depends on β, which in our study depends on the number of page evictions. As we will show in Section 8.5.2, the programs that we use in this study have not proven to be amenable to a demand paging environment. Therefore, we also estimate TLB Shootdown's overhead for smaller β. For the Simple/33%/64 TLBEs experiment, dividing β by 10 produces similar results. Dividing β by 20 produces similar results to those shown above, with the exception that TLB Shootdown outperforms Memory-based TLB systems with one paging arena. However, recall that in our simulations TLB inconsistencies are caused by only page evictions and that in a real system they can result from an increase in page protection, the invalidation of a virtual-to-physical memory mapping, and the remapping of a virtual or physical page. Also recall that the performance of Memory-based TLBs improves as the number of paging arenas increases.

Since $R \times C_{reload}$ asymptotically grows as $N \log N$ and our estimate of T_{Shoot} asymptotically grows as $N \log N$, it is not surprising that OH_{Shoot}, like OH_{Val} asymptotically grows as $N \log N$.

8.7. Simulation Results: Miscellany

In this section we discuss some general results that may be of interest to the reader:

- TLB-miss and TLB-reload rates,
- execution time, and
- *critical* pages, *i.e.*, pages that can cause program execution to pause.

8.7.1. TLB-Miss and TLB-Reload Rates

For our Memory-based TLBs and Validation experiments for systems with 64-entry TLBs, Graphs 60–63 present TLB-miss rates and Graphs 64–67 present TLB-reloads rates. The rates are computed by dividing the number of TLB misses or TLB reloads by the number of references to shared pages. A TLB miss results either in a TLB reload or a page fault, therefore, the difference between the number of TLB misses and the number of TLB reloads is the number of page faults.

For entire simulations, the TLB-miss rates range from a little over 10% to a little over 15%, and the TLB-reload rates range from approximately 3% to 8%. For the computational section of simulations, the TLB-miss rates range from a little over 7% to a little over 12%, and the TLB-reload rates range from approximately 1% to 6%.

8.7.2. Execution Time and Hot Pages

The execution times for an experiment reported by our simulations are fairly constant as N increases. (Notice that the variance between two points on the y-axis is very small.) In fact, as illustrated in Graphs 68–71, where the execution time of an experiment is denoted as *Simulation Time*, they slightly increase for larger N. Thus, no speedup is observed and we are are not able to measure the effect of TLB management overhead on speedup. This behavior occurred whether we simulated one iteration or multiple iterations. Our data shows that this is because the programs we used in this study are not conducive to demand paging. This was also found to be true by Peskin, McQueen, and Greenberg [1987] with respect to the three-dimensional version of the Peskin program.

Critical pages are pages that can cause program execution to pause. Thus, critical pages can severely degrade performance. Examples of critical pages are:

- a page on which a serial task faults,
- a page or set of pages on which N concurrently executing tasks fault.

In these cases, such pages can cause program execution to pause if the related task or tasks cannot be executed concurrently with any other tasks defined by the program. N tasks all can fault concurrently on the same shared page or on N different pages.

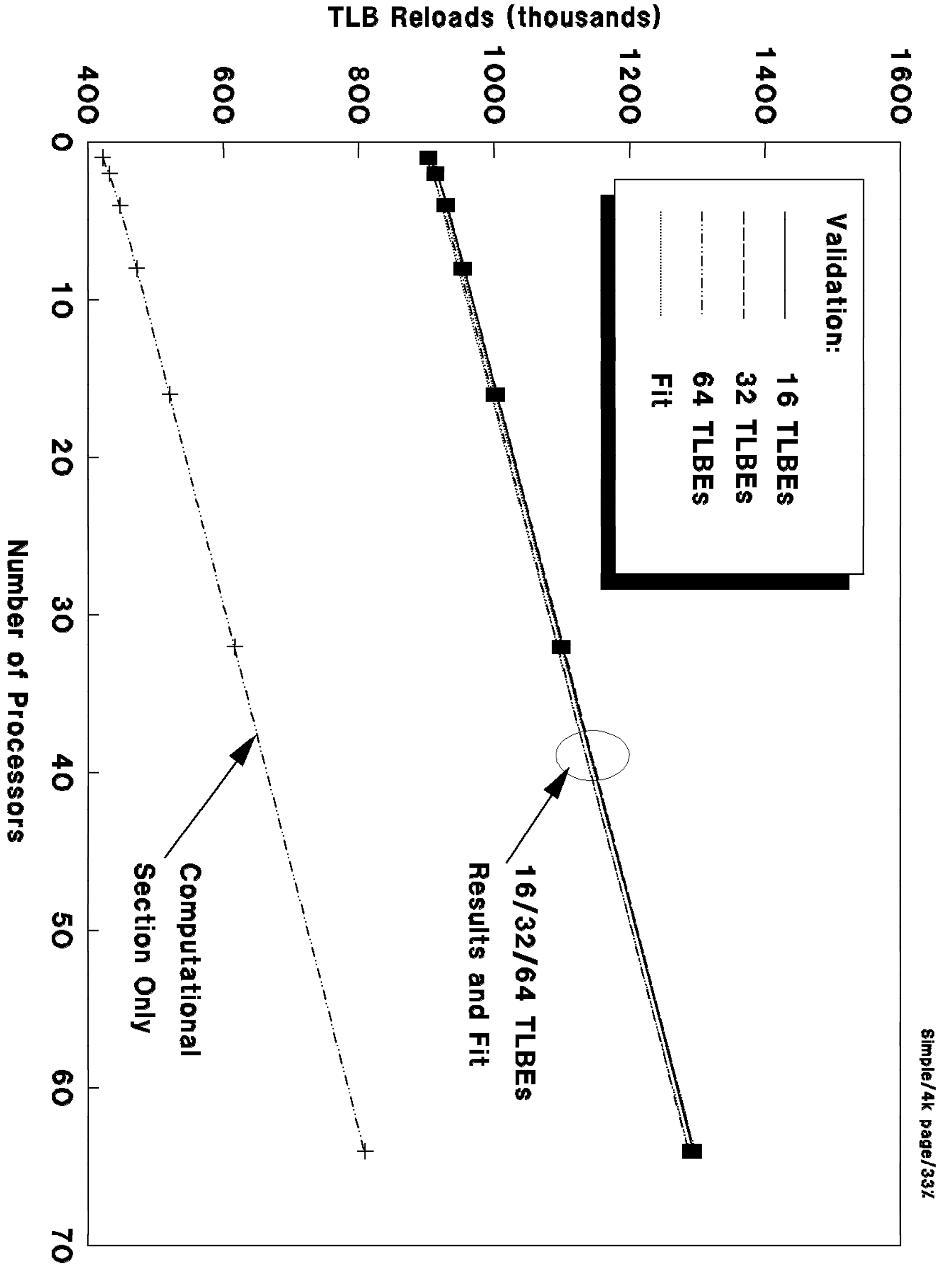
For example, consider a program with a parallel task group that, because of precedence constraints, cannot be executed with any other tasks in the program, *i.e.*, only the tasks in the group can be executed concurrently. Assume that there are $M > N$ tasks in the group. If each task faults on a critical page, program execution may pauses until a page or pages become resident. Assuming that processors may context switch on a page fault, as N increases, the probability of there being a runnable task to execute decreases. This increases the probability that program execution is paused and increases the potential waiting time.

This problem associated with critical pages is one reason why the execution time of our experiments remains fairly constant as N increases. In our simulations a processor that encounters a page fault context switches to another task, if a runnable task is available.

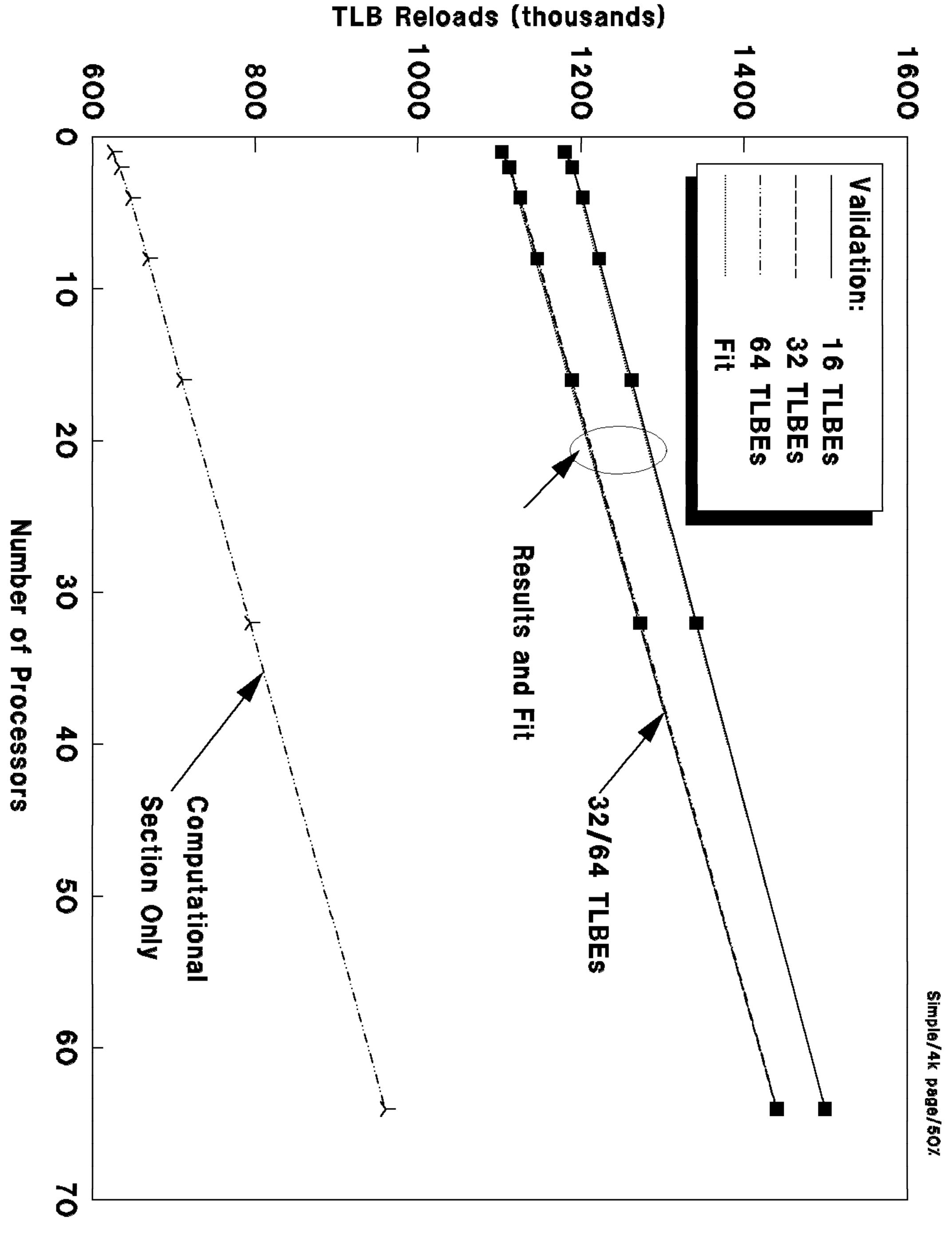
Otherwise, the processor waits until the referenced page becomes memory resident. Our simulation data indicates that hot pages do exist in our workloads. As indicated in Graphs 9–12, our experiments show that there are a set of pages that are referenced by all N processors and, accordingly, there are a set of pages that are faulted on by all N processors. Simulation data also shows that there are pages on which all the tasks of a parallel task group fault: For the Simple/33% and Simple/50% experiments, respectively, over 4300 and 2300 times, all 256 tasks faulted on a page. Even though this is not alot in comparison to the approximately 380,000 times that one task faulted on a page, as a result of such behavior, execution pauses if there are no other tasks to execute. For the Peskin/33% and Peskin/50%, respectively, experiments over 740 and 540 times, all 256 tasks faulted on a page. Approximately 80,000 and 1,800 times only one task faulted on a page.

Experiments where context switching is minimized show that critical pages do, indeed, degrade the execution time of our experiments. Context switching is minimized when the sizes of physical memory and virtual memory are equal. In this case, context switching is caused only by initial page-ins. As illustrated in Graphs 72 and 73, the execution time of such experiments decreases as N increases up to approximately $N = 16$ for the Simple experiments and $N = 8$ for the Peskin experiments. Graphs 72 and 73 show the simulation time for the initialization, computational, and termination sections of the program. Provided there is no serial bottleneck, we believe that if we had simulated many iterations of the computational section, execution time would decrease as N increases. This is because the initial effect of the page-ins would not dominate the simulation time.

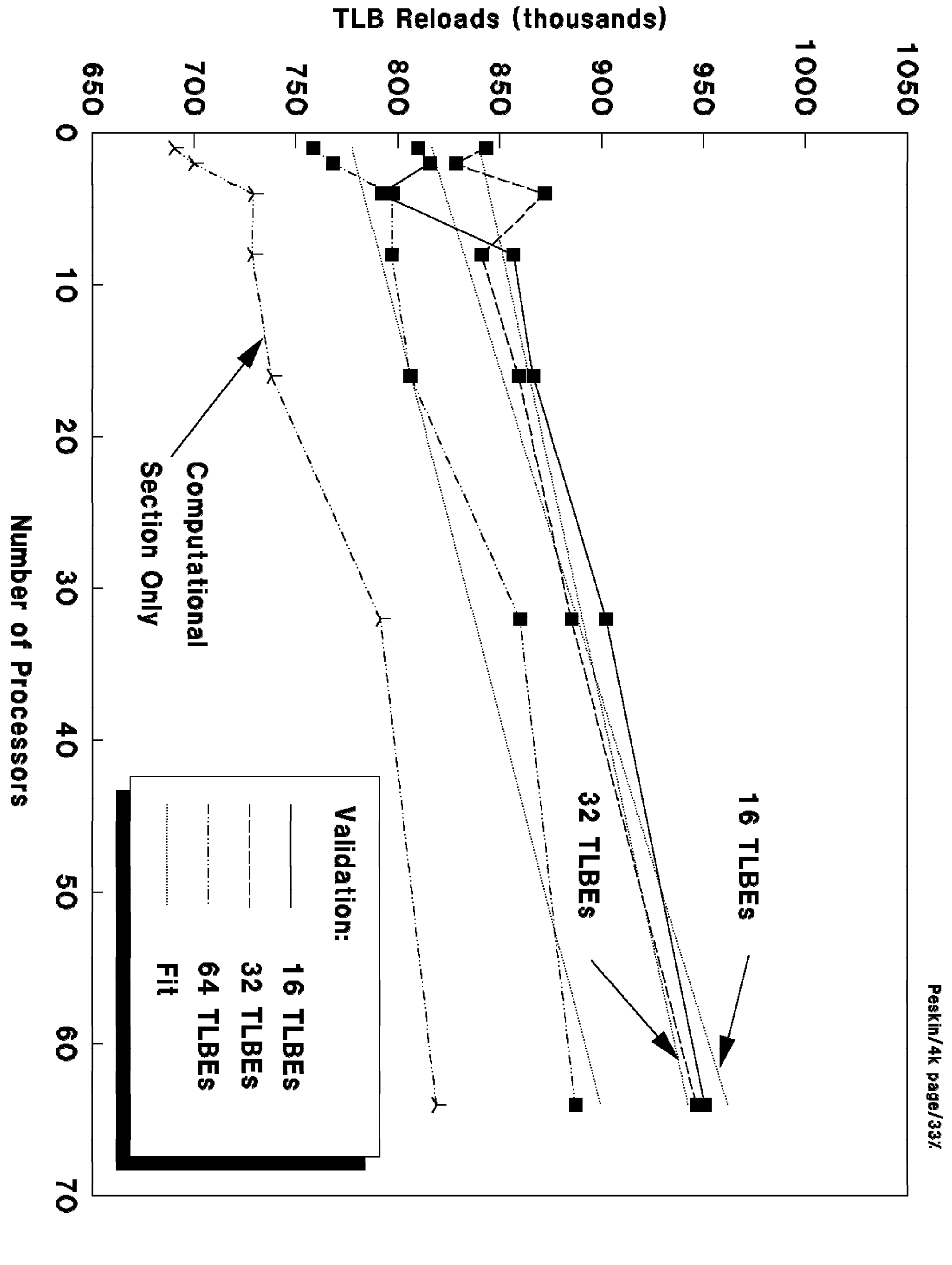
The Simple experiments for systems with equal size physical and virtual memory and, thus, minimal amounts of context switching, result in R growing linearly with N. By running analogous experiments for two iterations, we were able to measure R for the computational and termination sections of the program, without context switching. In these cases, the growth of R with N is different than that for the experiments reported in this thesis. Different task schedulings produce different program executions and workloads and, thus, different system behavior. The number of generated reloads is larger than for analogous experiments reported in this thesis, where context switching was much more prevalent. We believe this is because the probability of tasks being able to share processor-based TLB entries is increased when the granularity of the tasks that share pages is decreased.



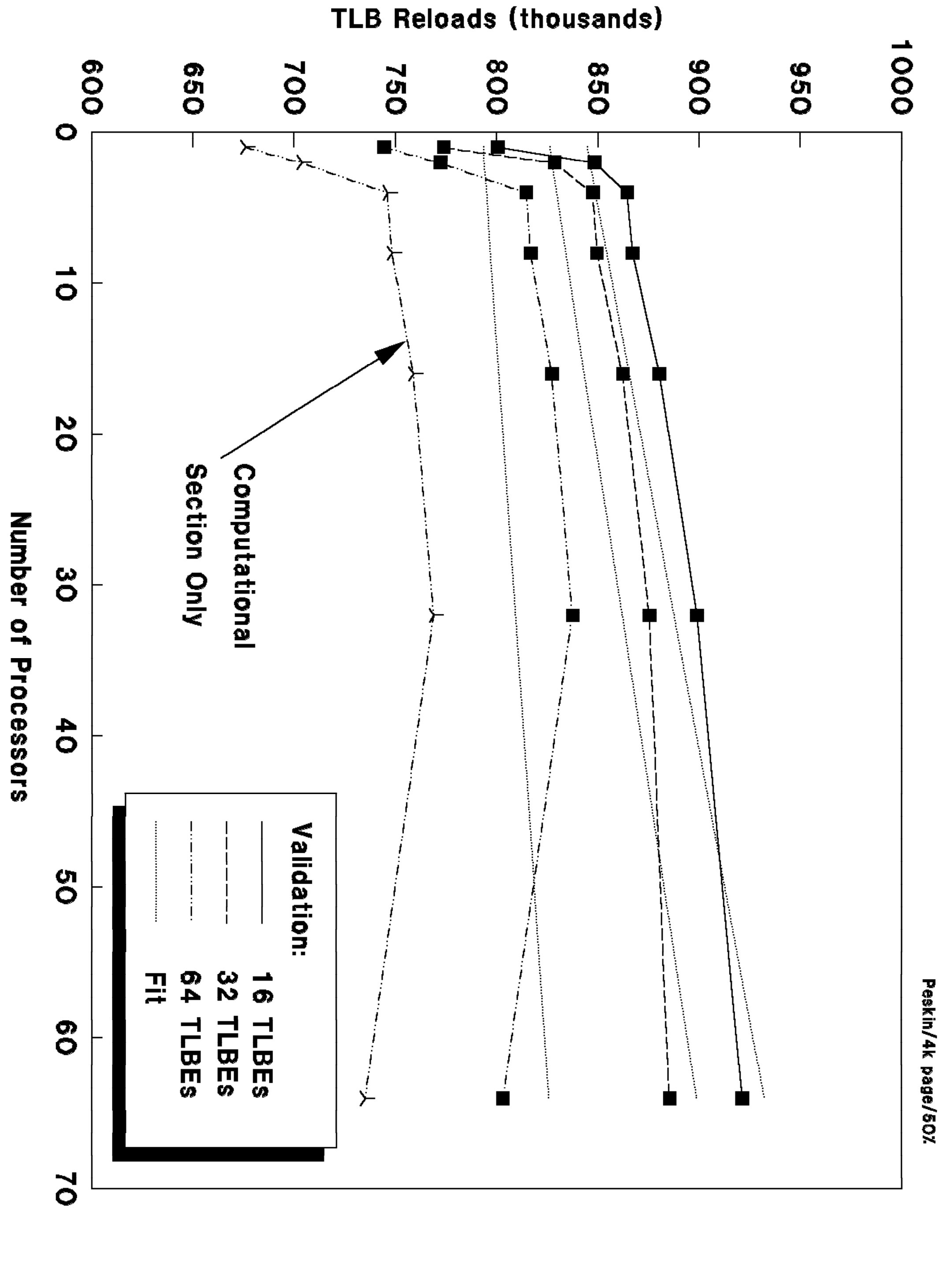
Graph 1. R_{Val} **vs.** N **– Simple 33%**



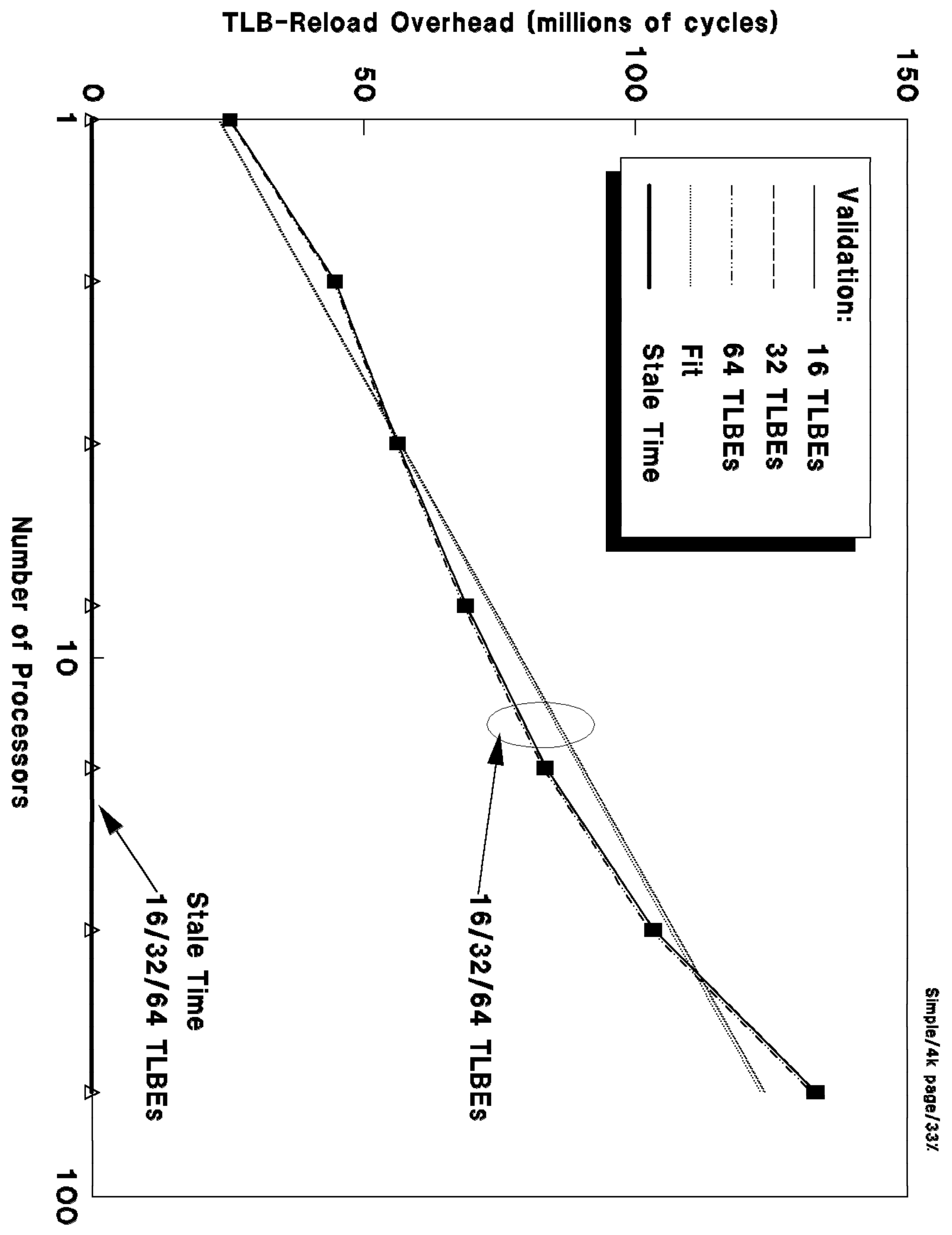
Graph 2. R_{Val} **vs.** N **– Simple 50%**



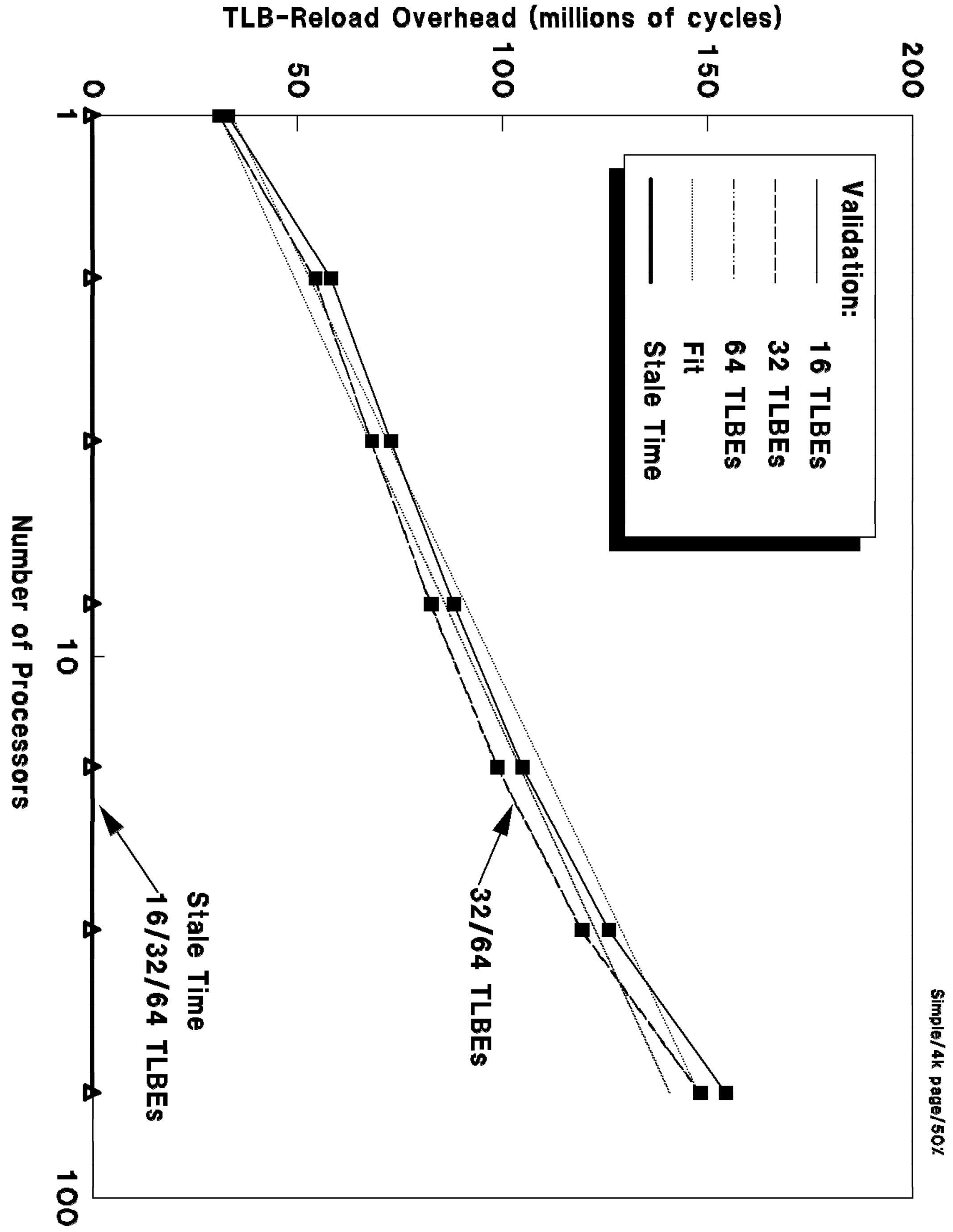
Graph 3. R_{Val} **vs.** N **– Peskin 33%**



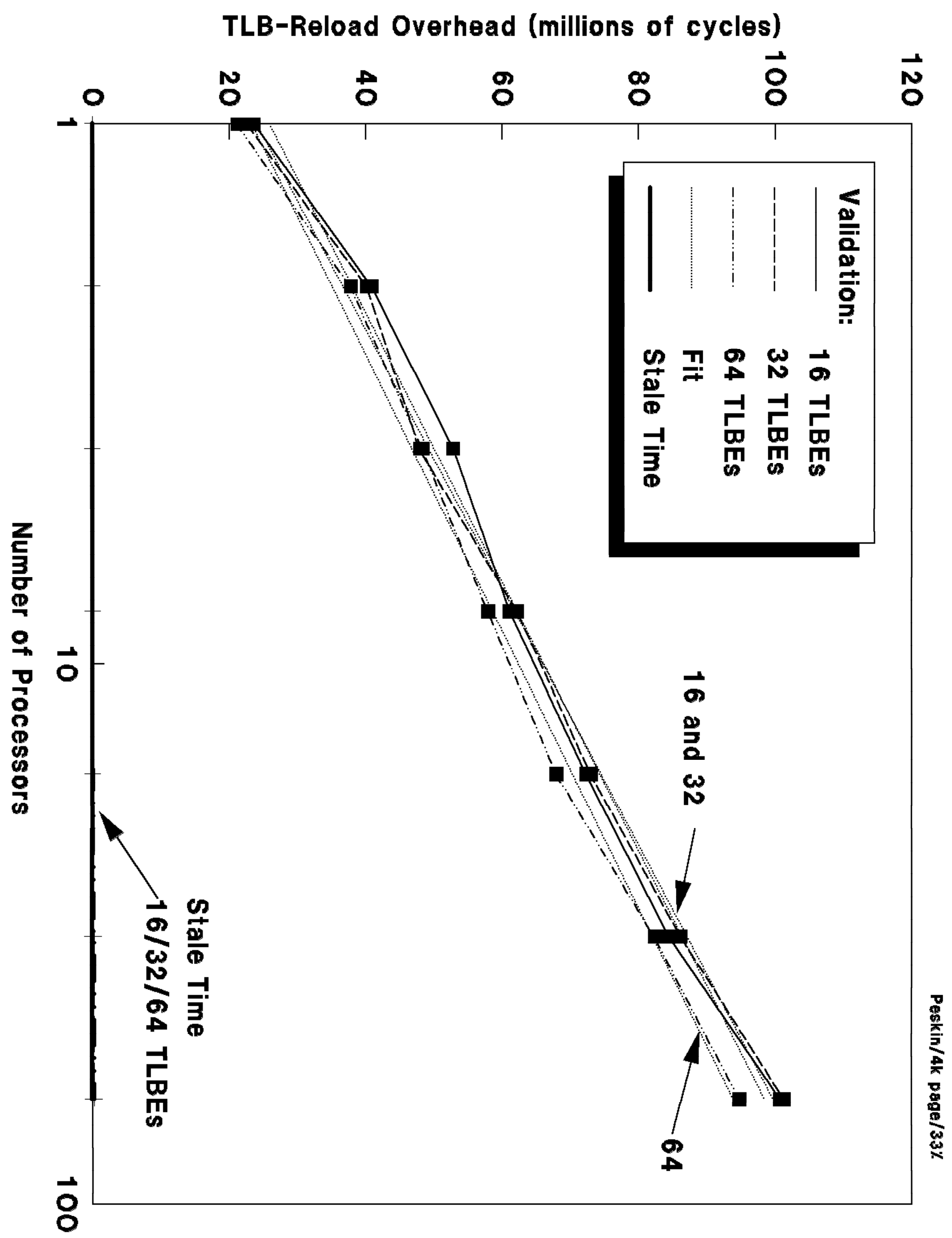
Graph 4. R_{Val} **vs.** N **– Peskin 50%**



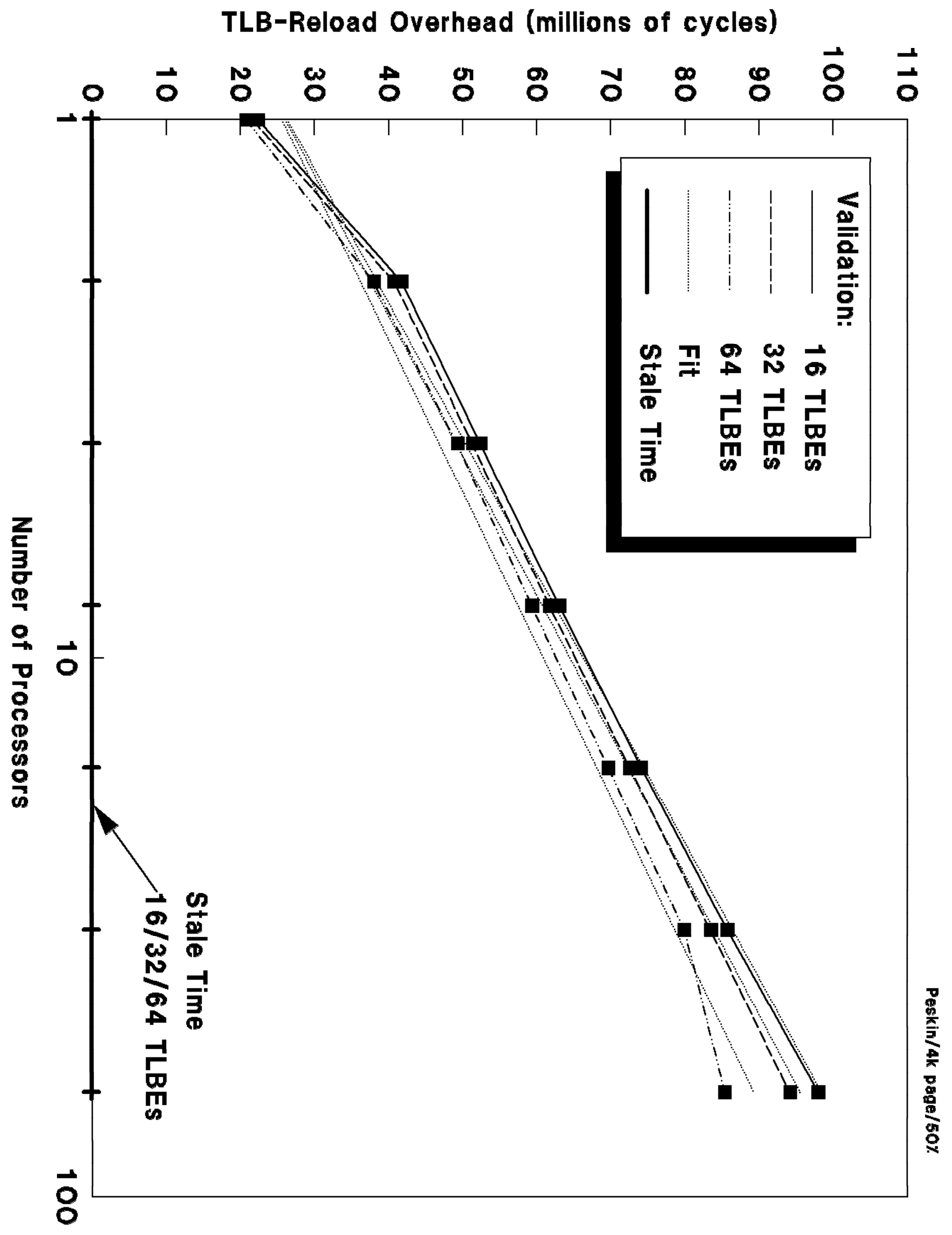
Graph 5. OH_{Val} **vs.** N **– Simple 33%**



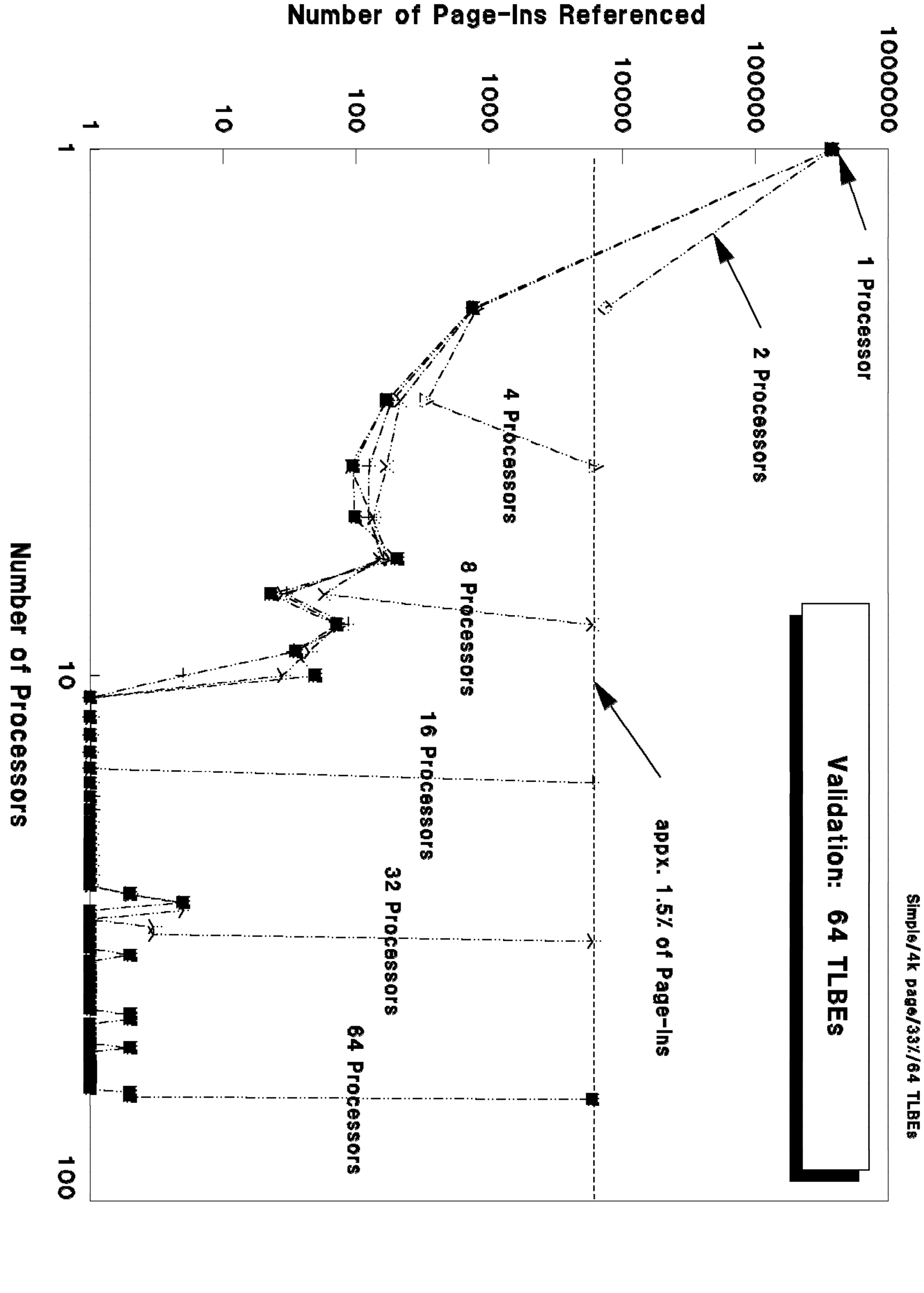
Graph 6. OH_{Val} **vs.** N **– Simple 50%**



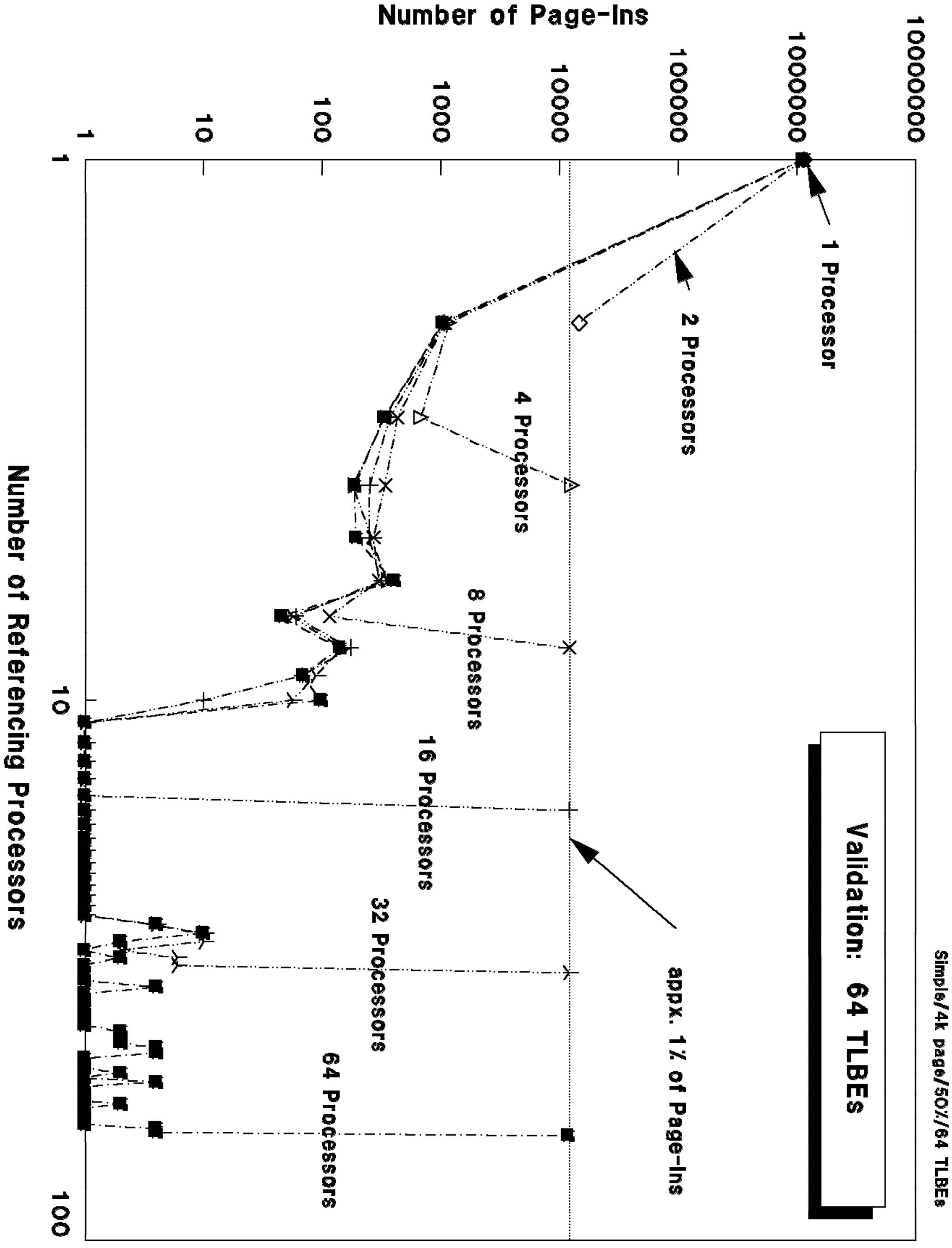
Graph 7. ***OH_{Val}* vs. *N* – Peskin 33%**



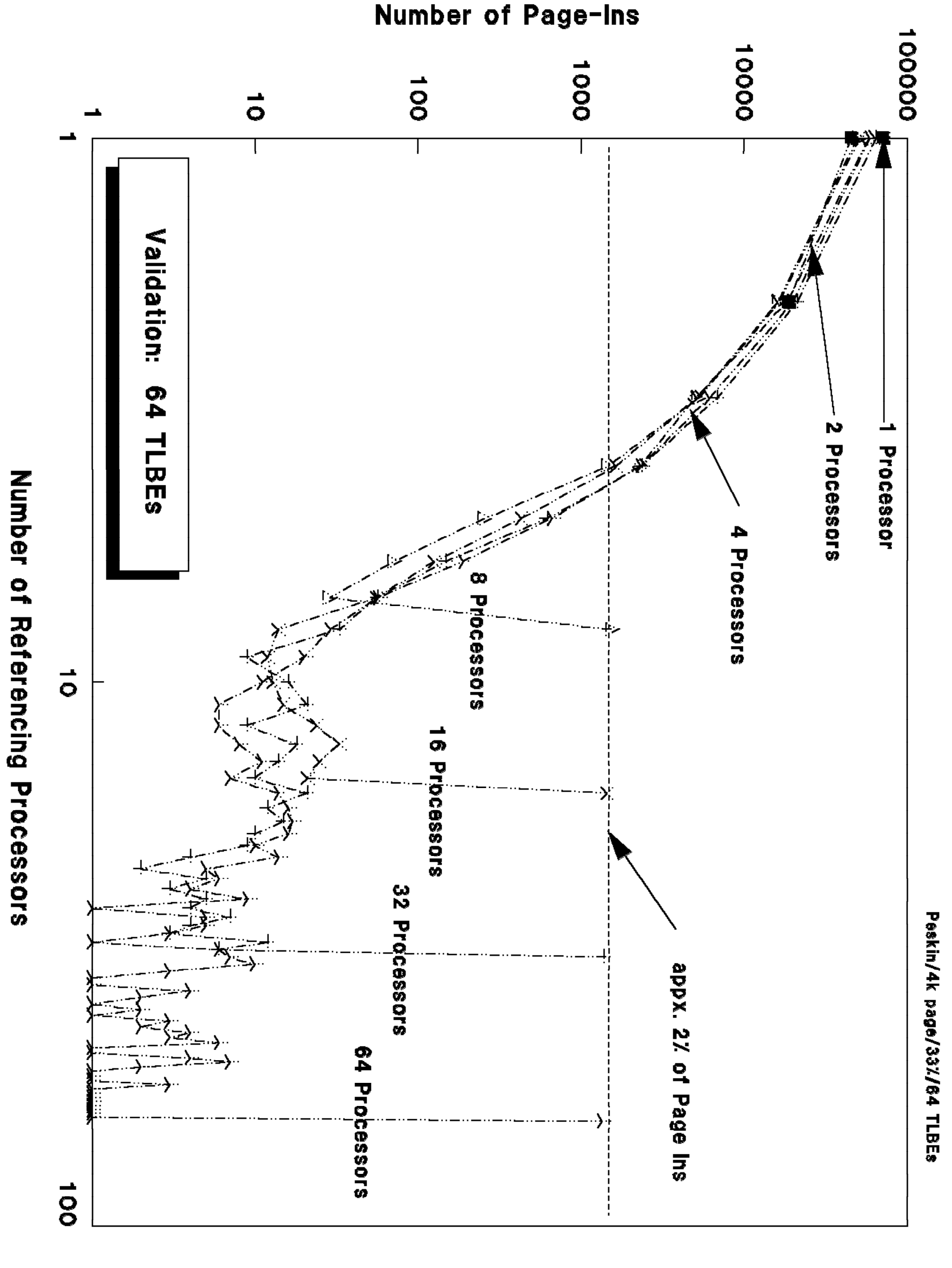
Graph 8. OH_{Val} vs. N – Peskin 50%



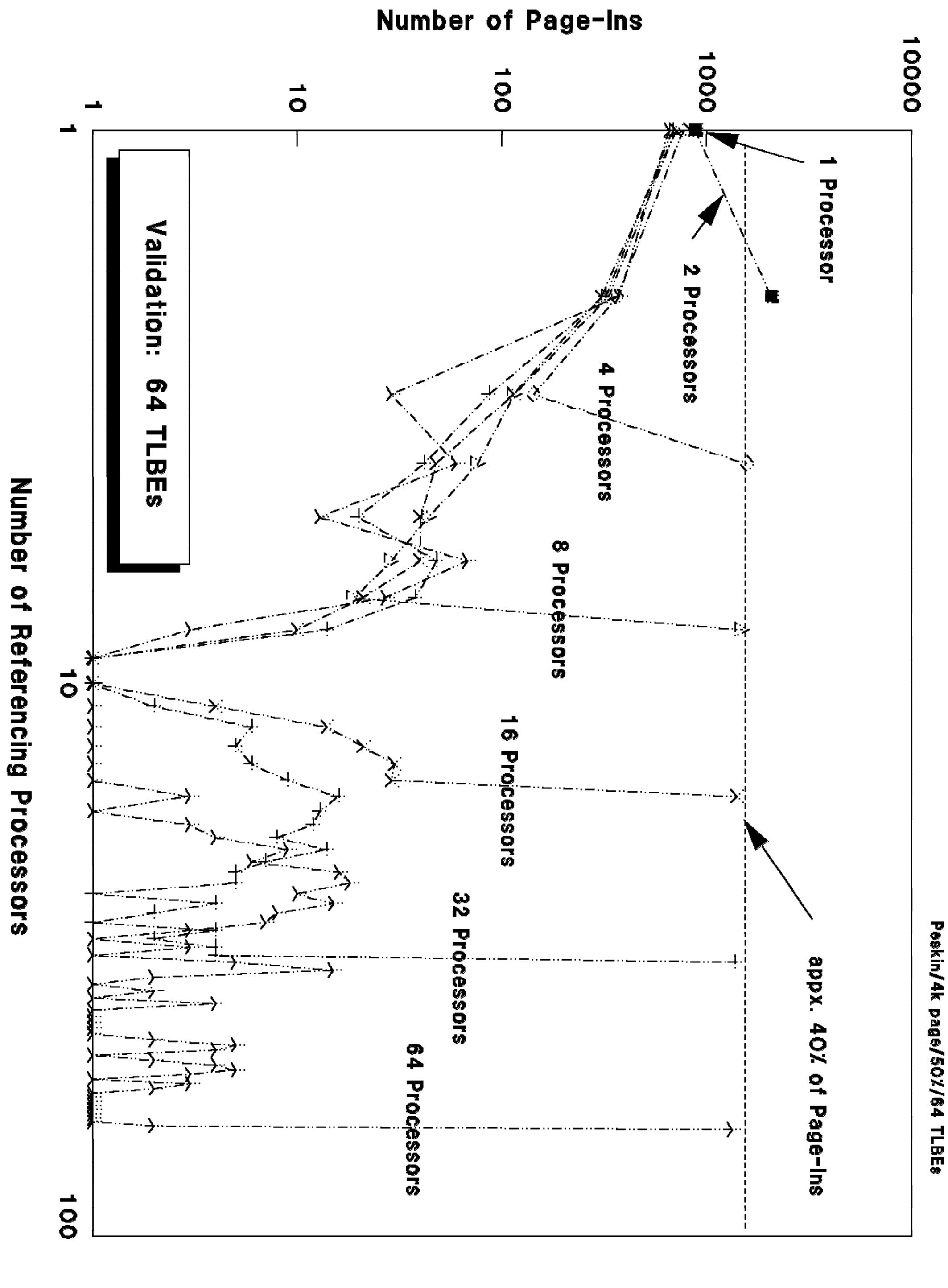
Graph 9. Processors Referencing Page-Ins – Simple 33%/64 TLBEs



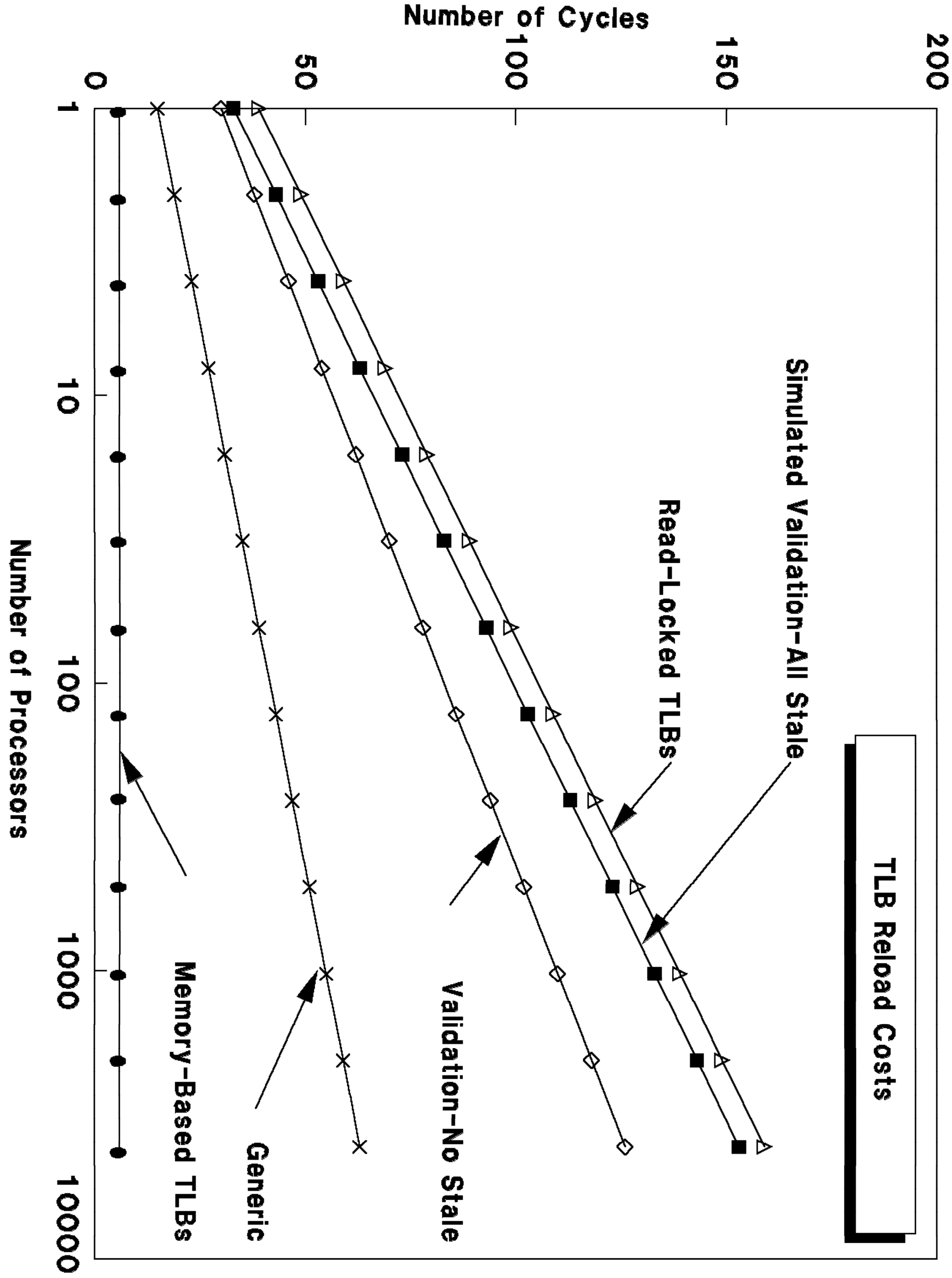
Graph 10. Processors Referencing Page-Ins – Simple 50%/64 TLBEs



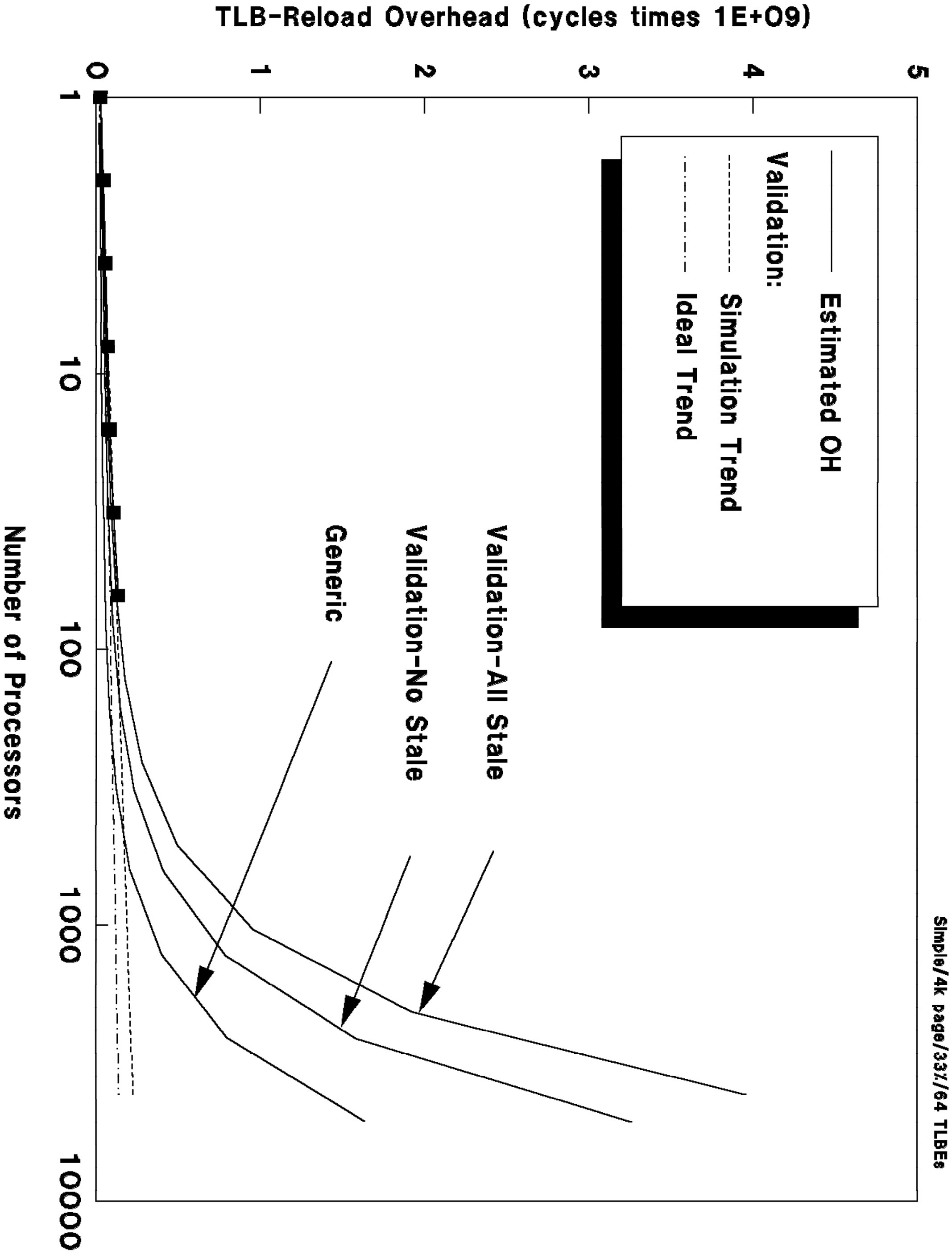
Graph 11. Processors Referencing Page-Ins – Peskin 33%/64 TLBEs



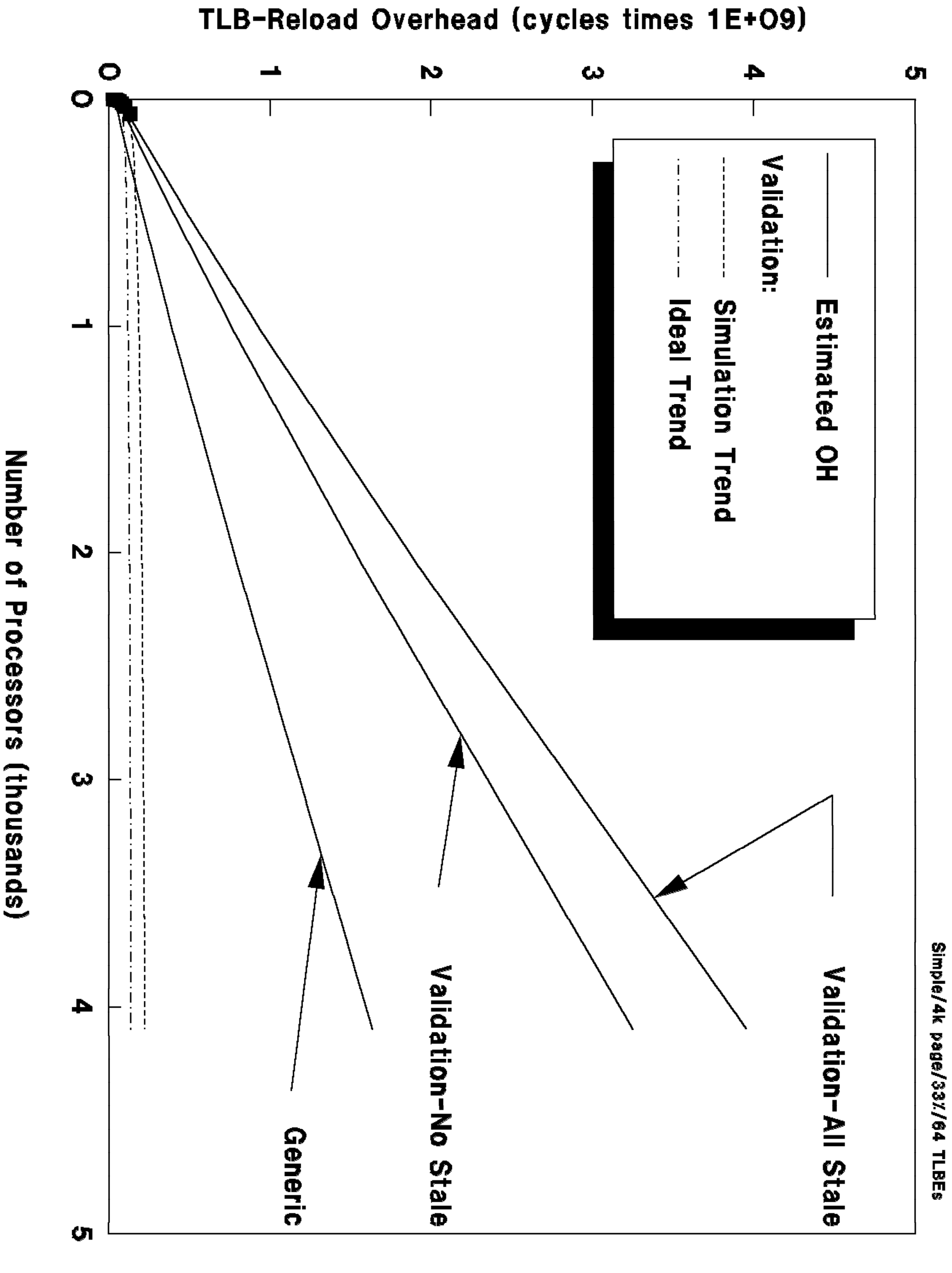
Graph 12. Processors Referencing Page-Ins – Peskin 50%/64 TLBEs



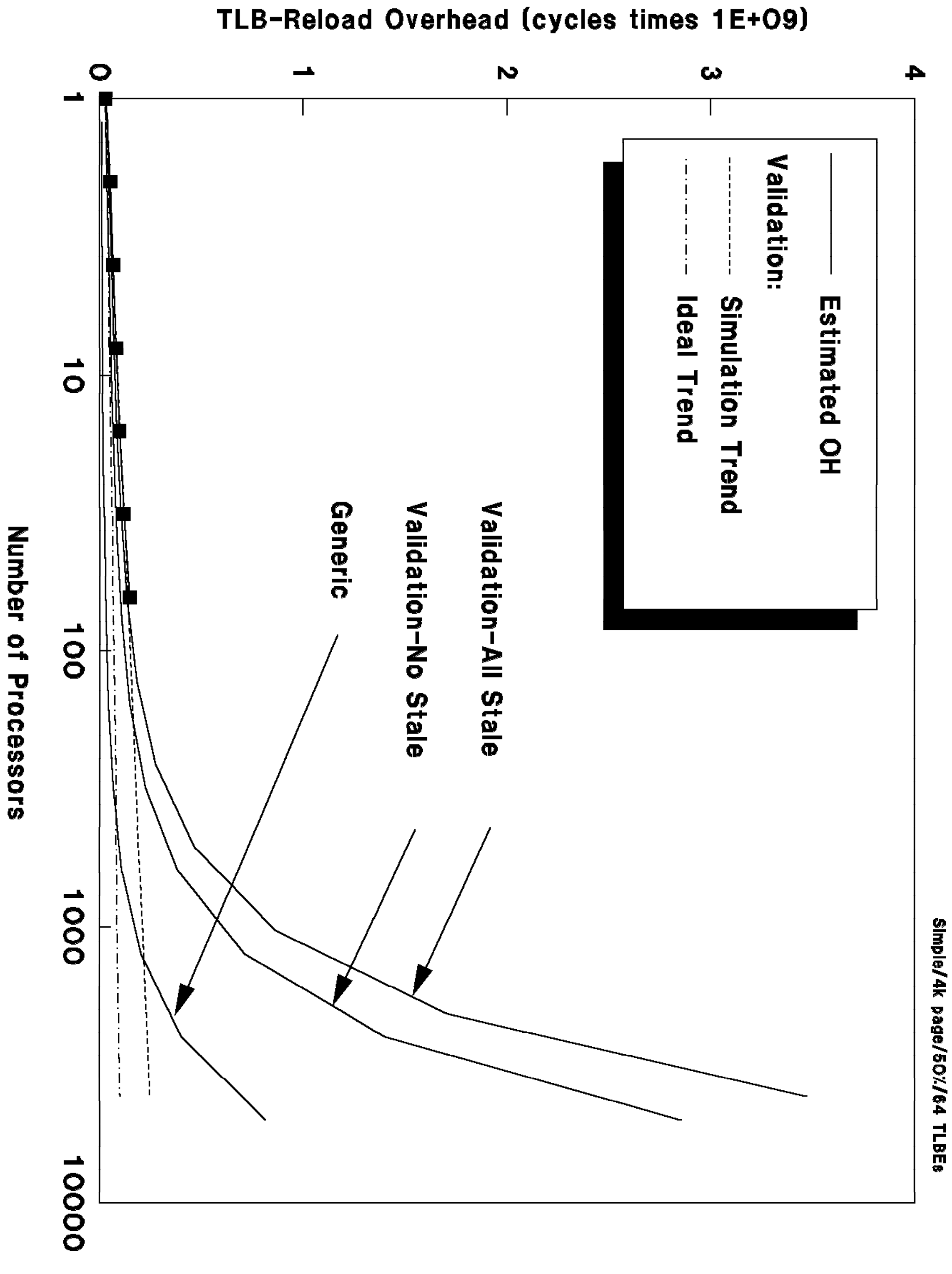
Graph 13. TLB Reload Costs vs. ***N***



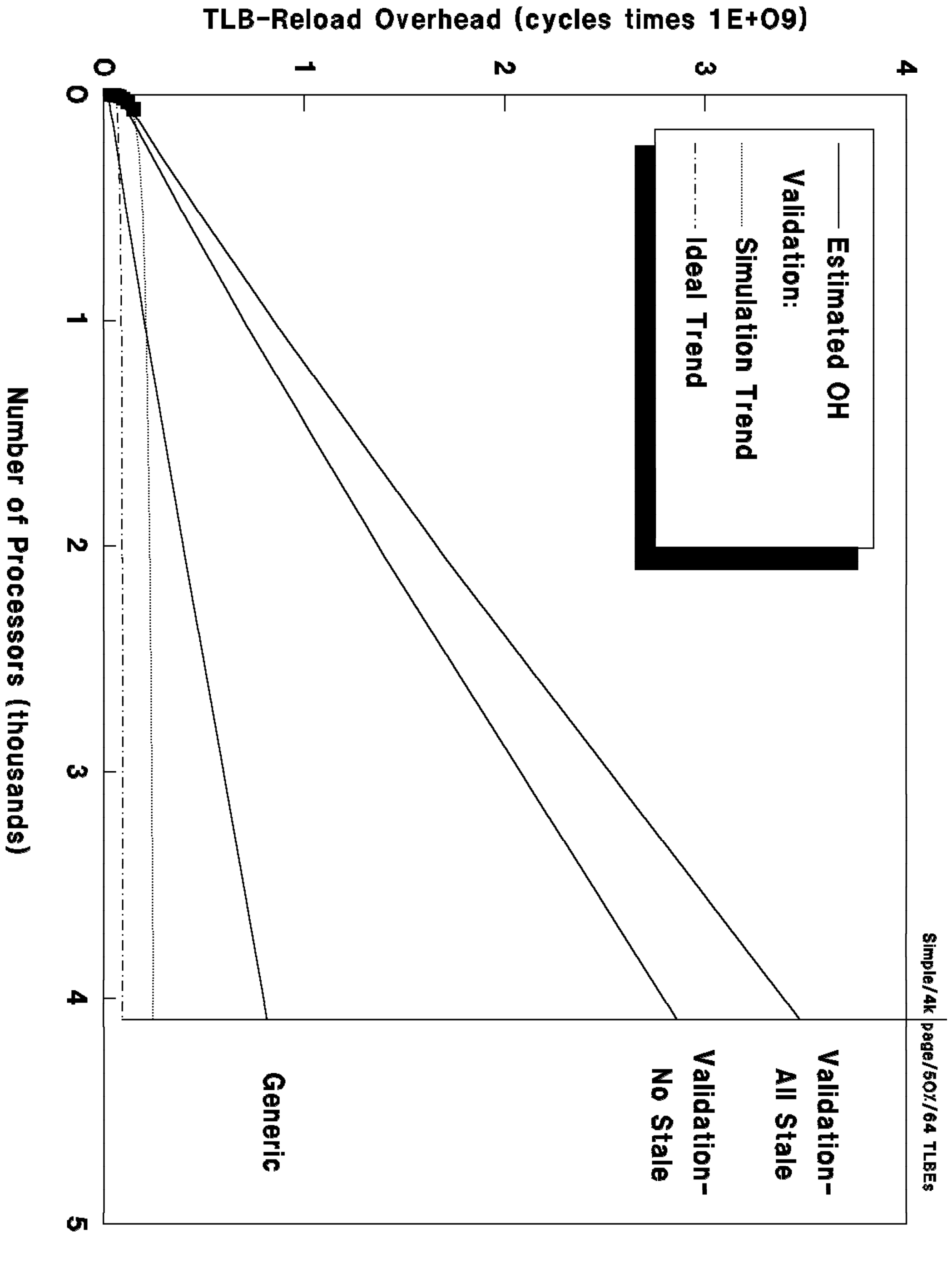
Graph 14. ***OH***$_{Val}$**: Results vs. Model – Simple 33%/64 TLBEs**



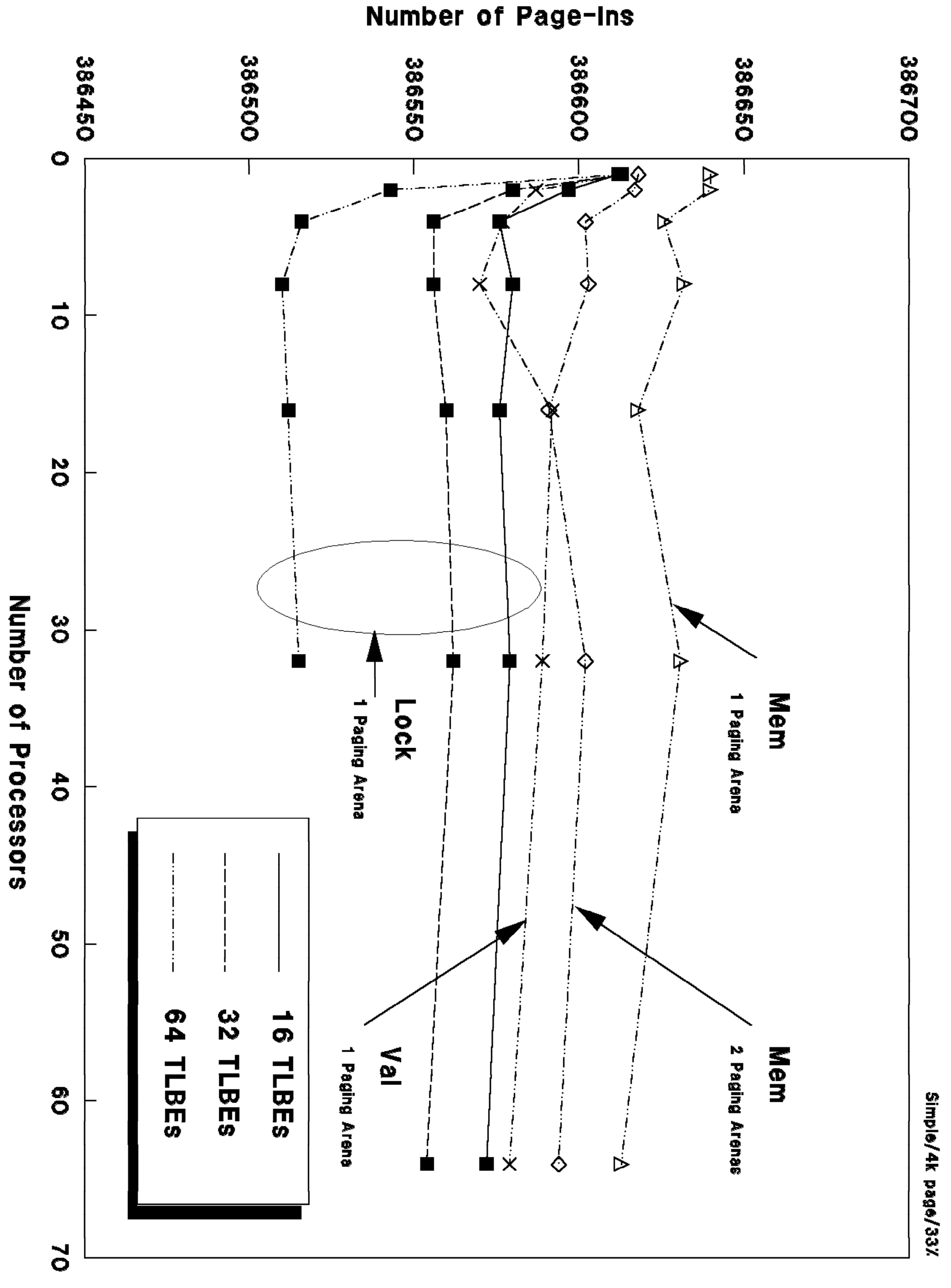
Graph 15. ***OH_{Val}*: Results vs. Model – Simple 33%/64 TLBEs**



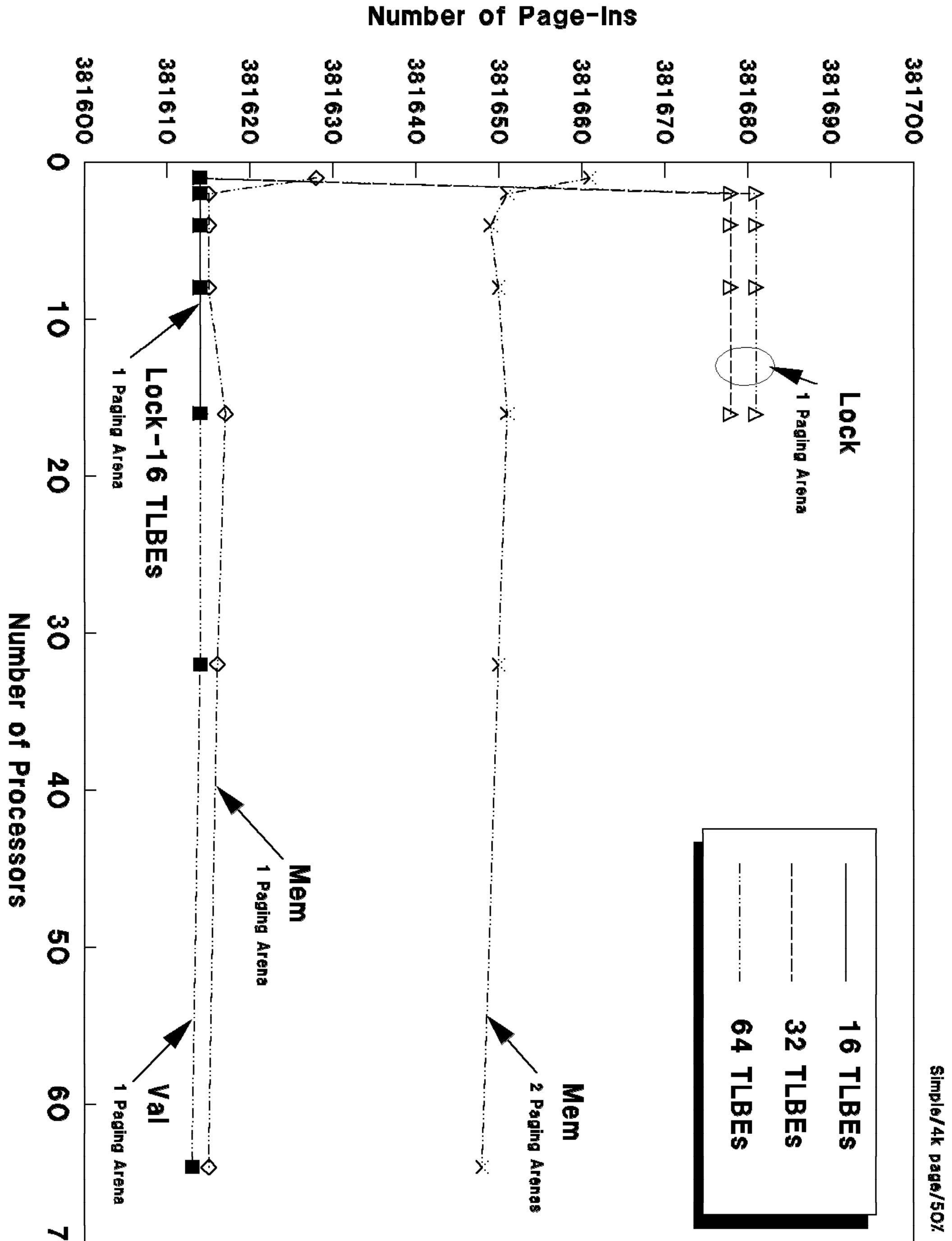
Graph 16. ***OH_{Val}*****: Results vs. Model – Simple 50%/64 TLBEs**



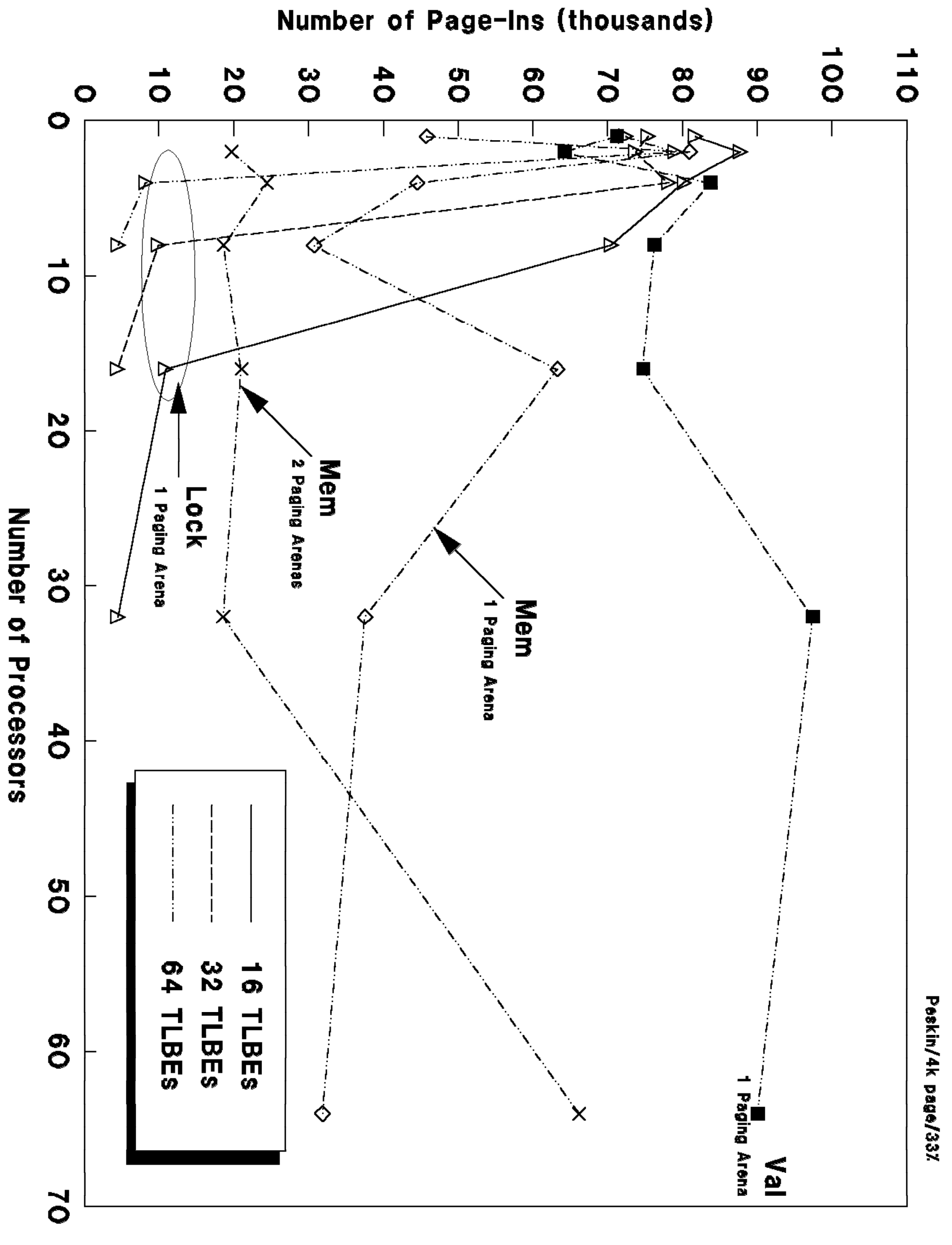
Graph 17. ***OH_{Val}*****: Results vs. Model – Simple 50%/64 TLBEs**



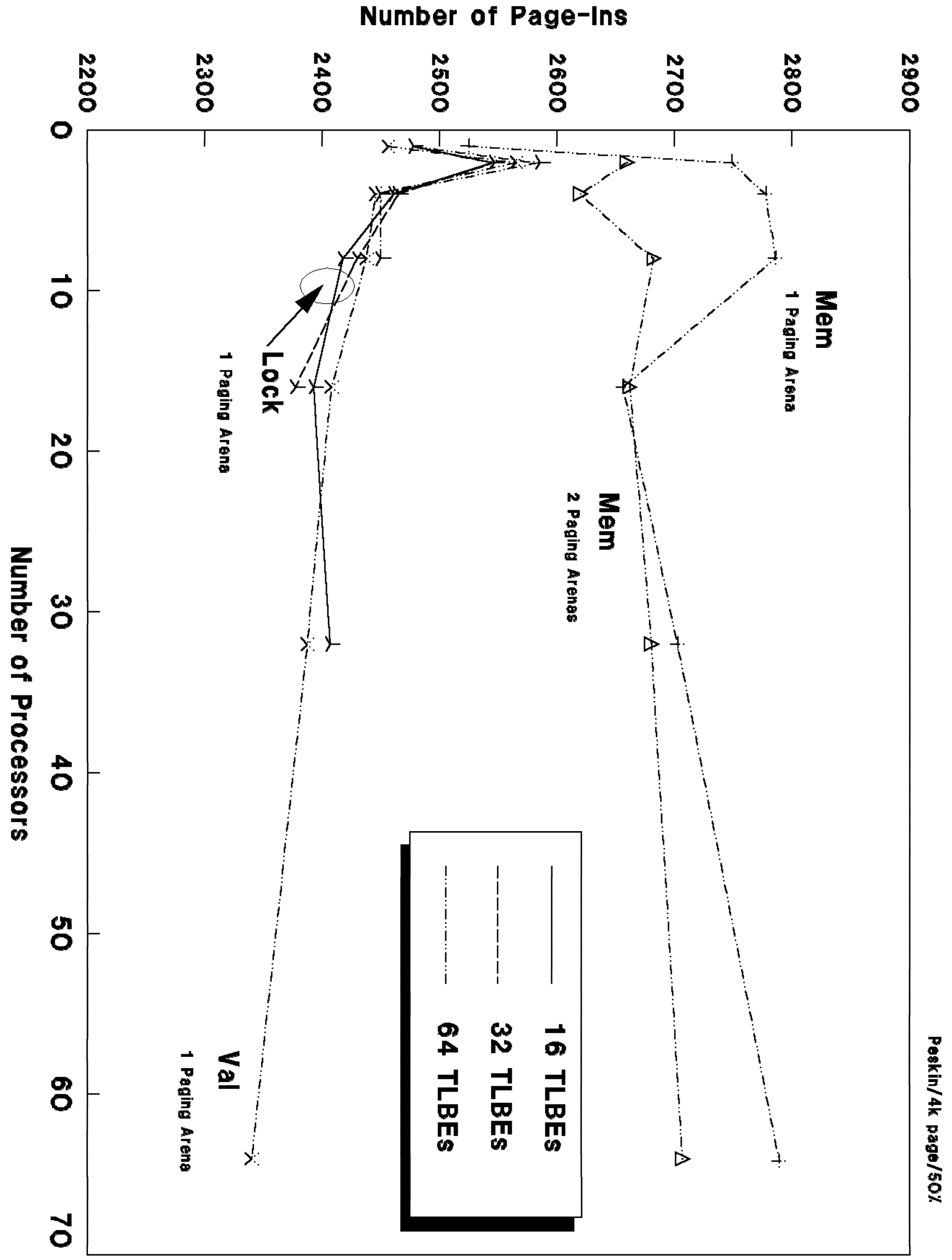
Graph 18. Page-Ins vs. *N*– Simple 33%



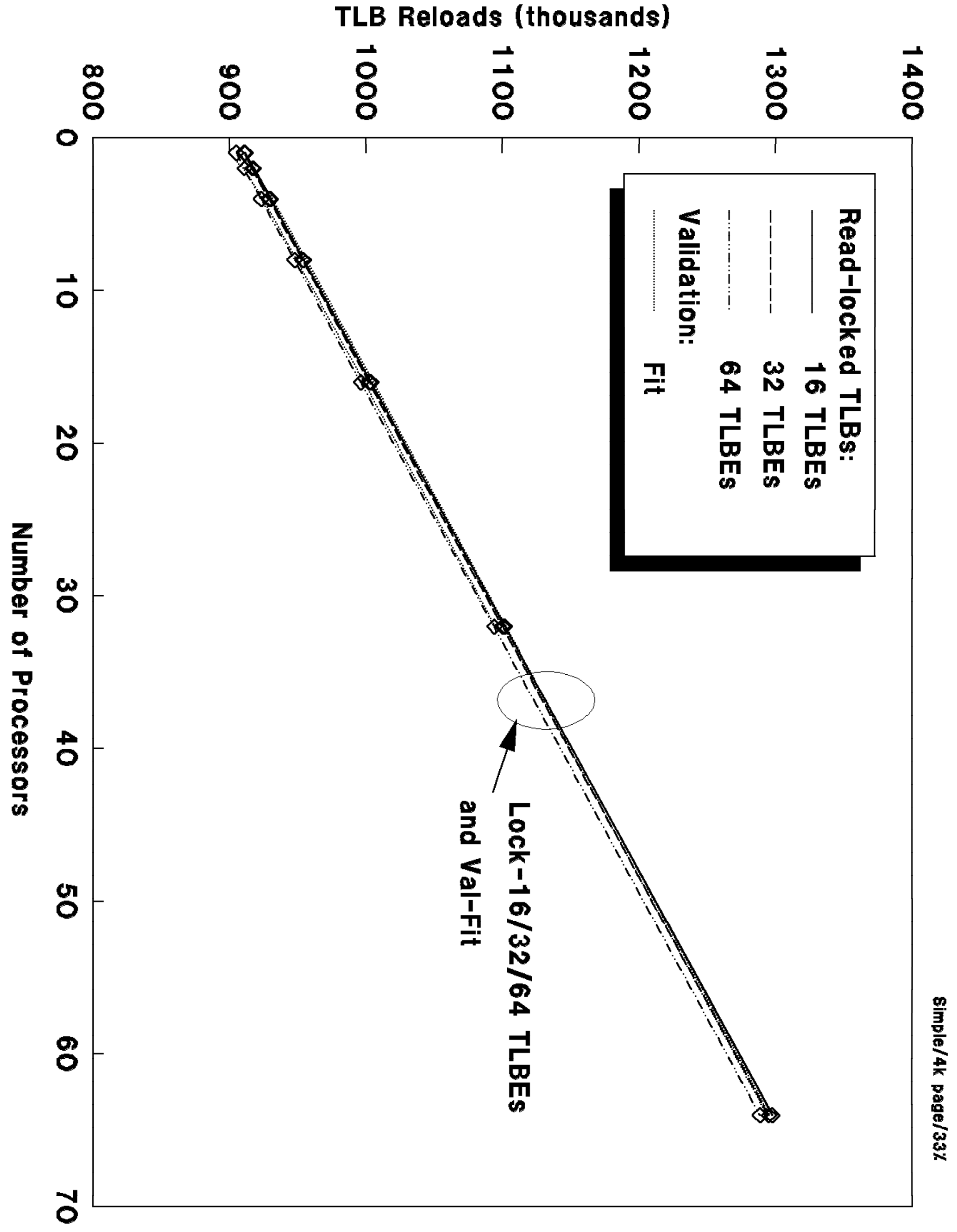
Graph 19. Page-Ins vs. *N*– Simple 50%



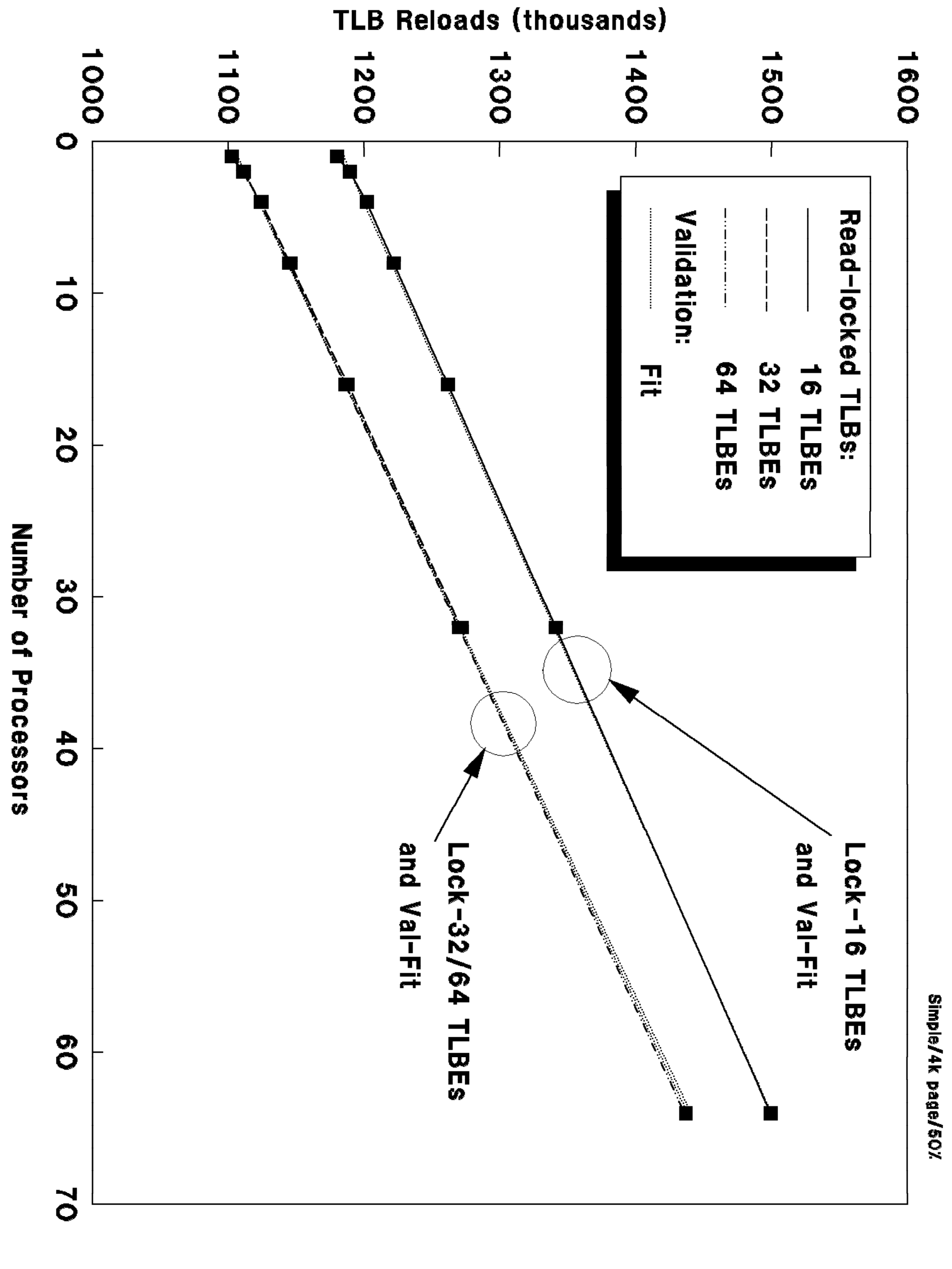
Graph 20. Page-Ins vs. *N*– Peskin 33%



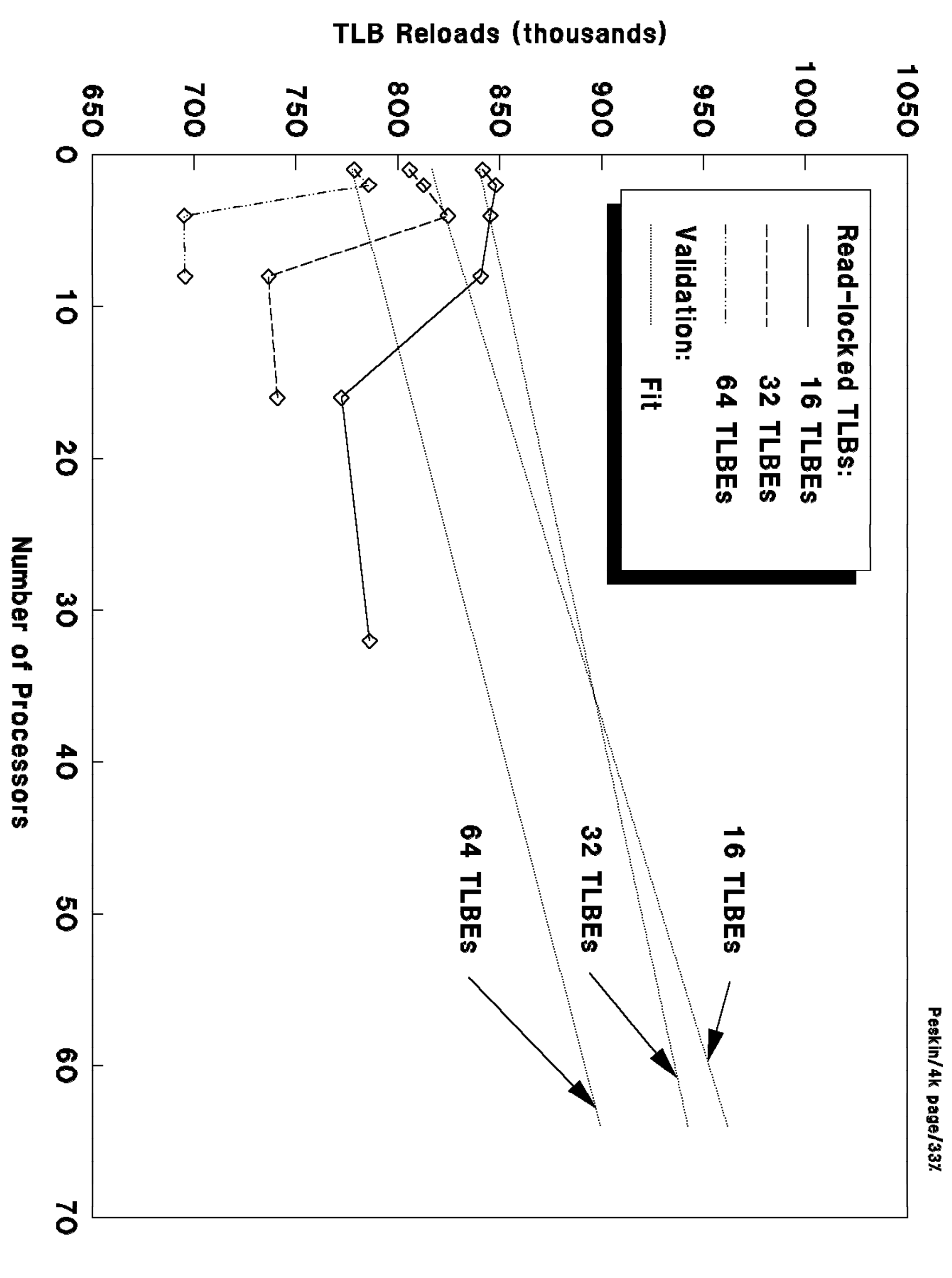
Graph 21. Page-Ins vs. *N*– Peskin 50%



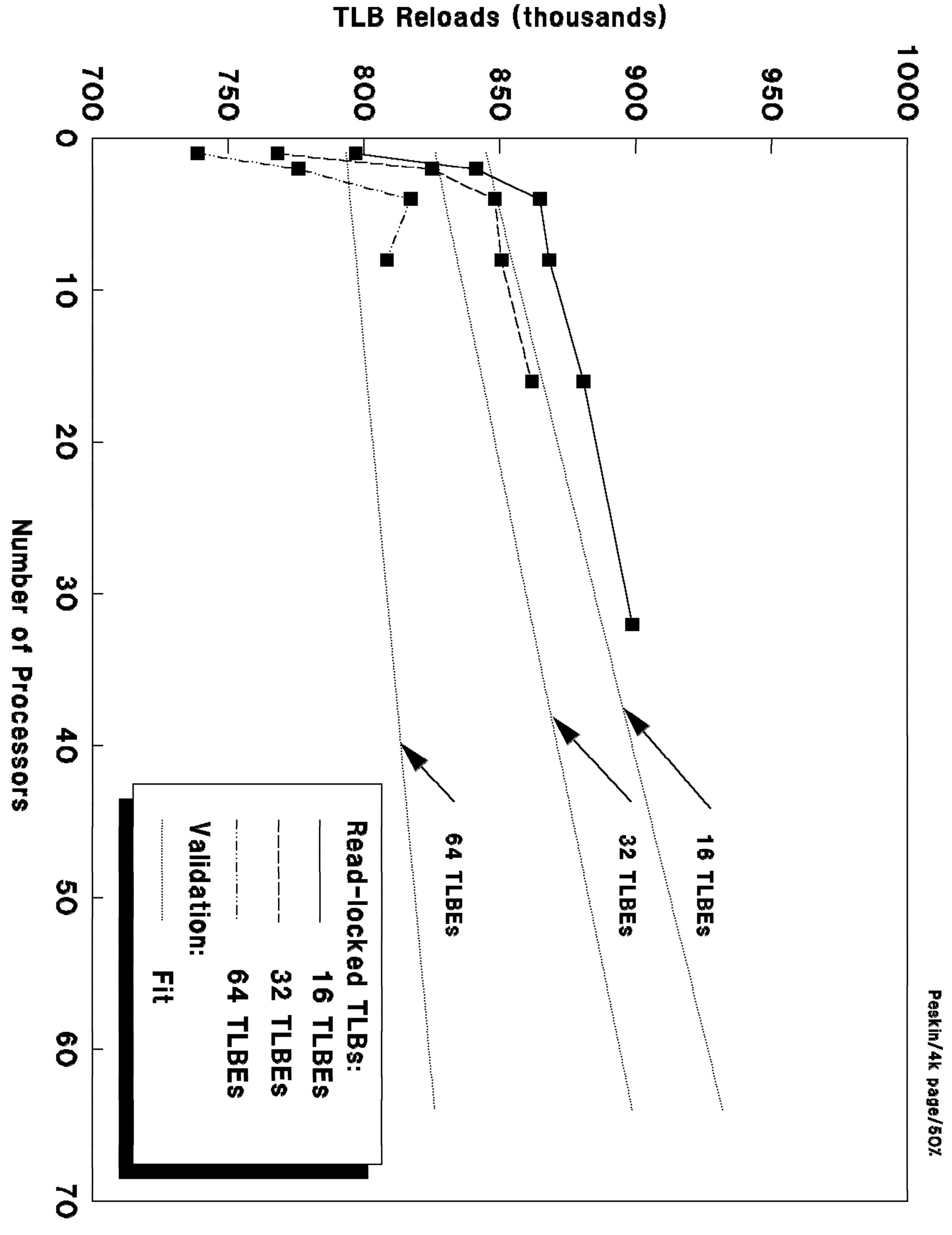
Graph 22. R_{Lock} **and** R_{Val} **vs.** N **– Simple 33%**



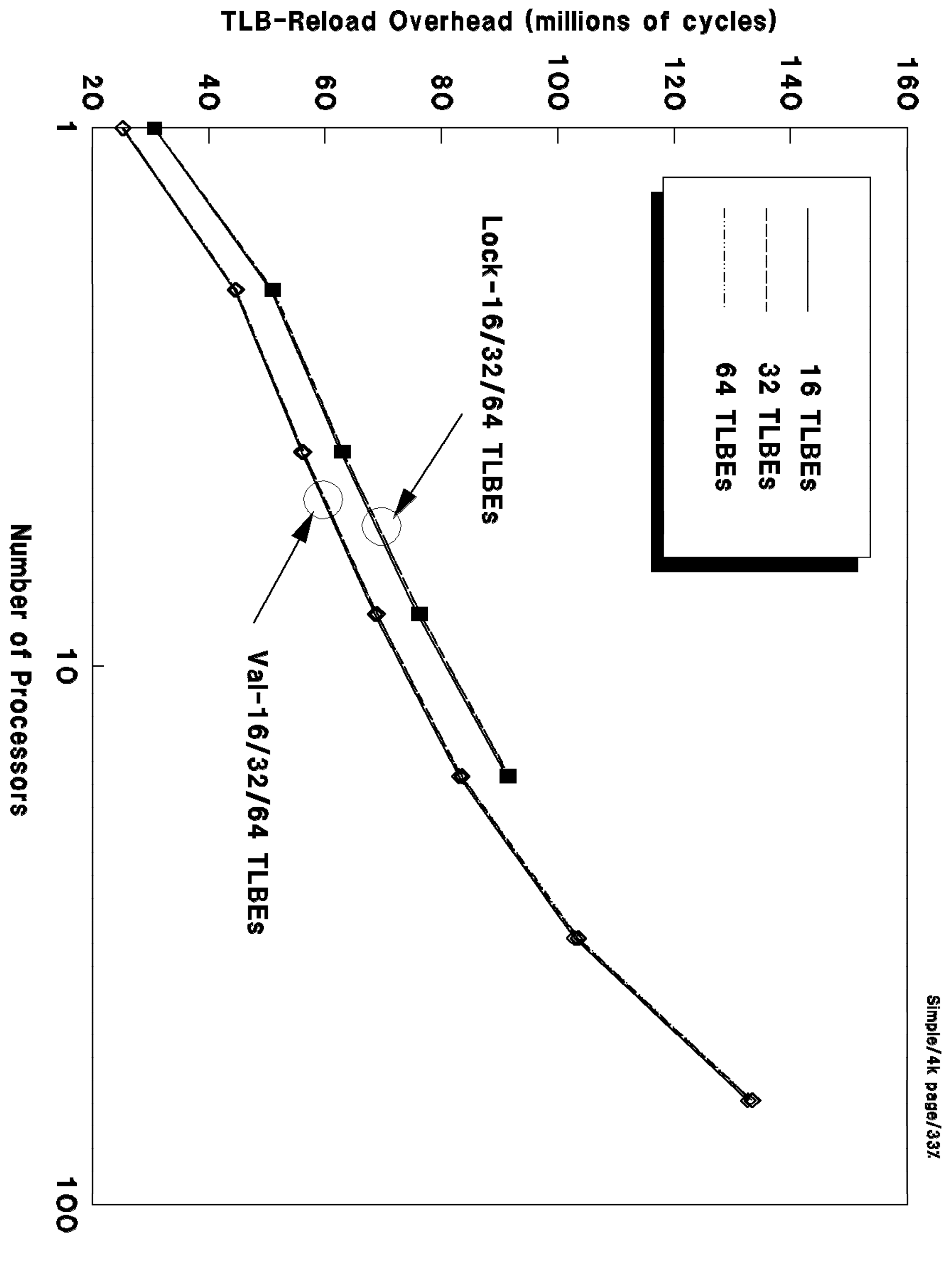
Graph 23. R_{Lock} and R_{Val} vs. N – Simple 50%



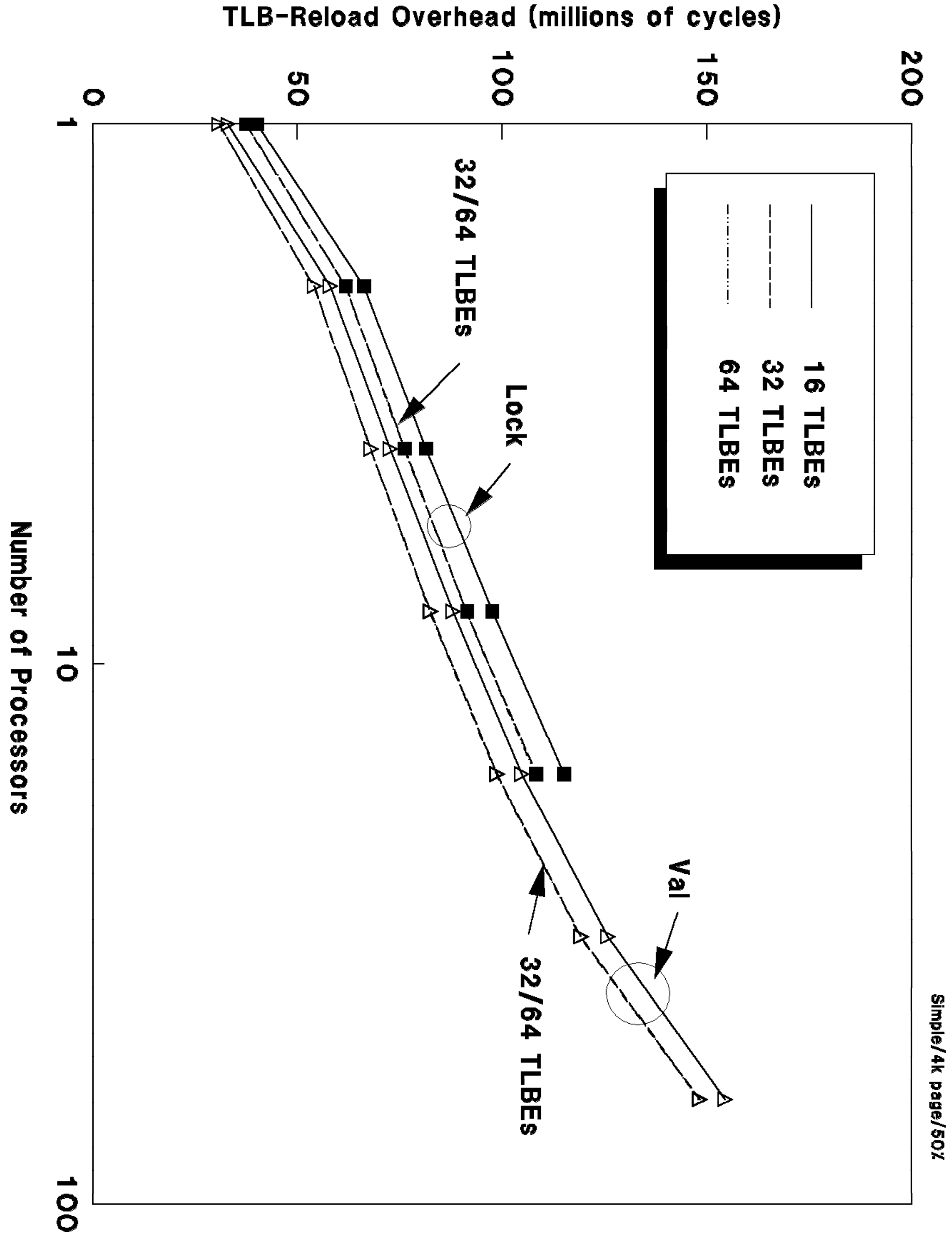
Graph 24. R_{Lock} and R_{Val} vs. N – Peskin 33%



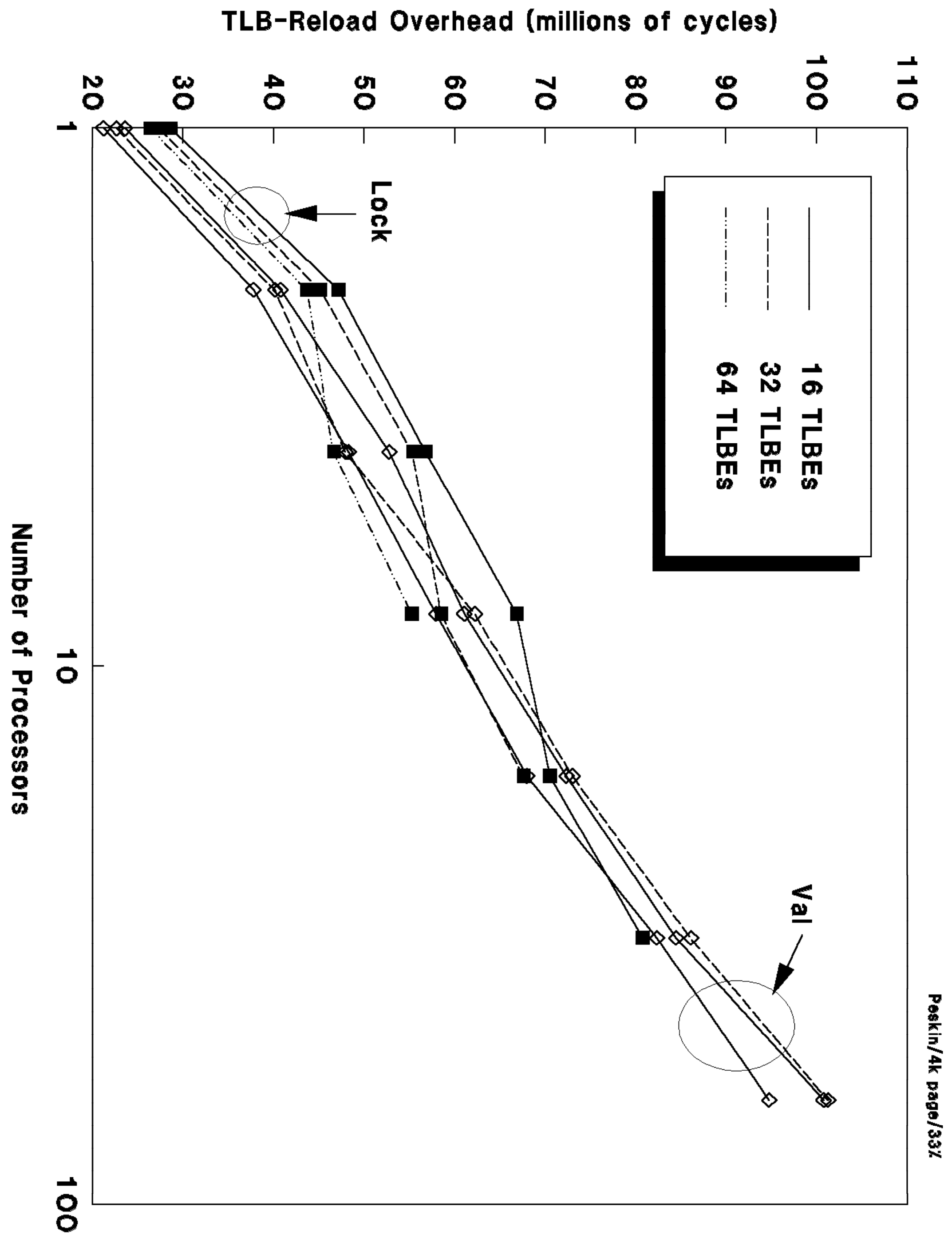
Graph 25. R_{Lock} **and** R_{Val} **vs.** N **– Peskin 50%**



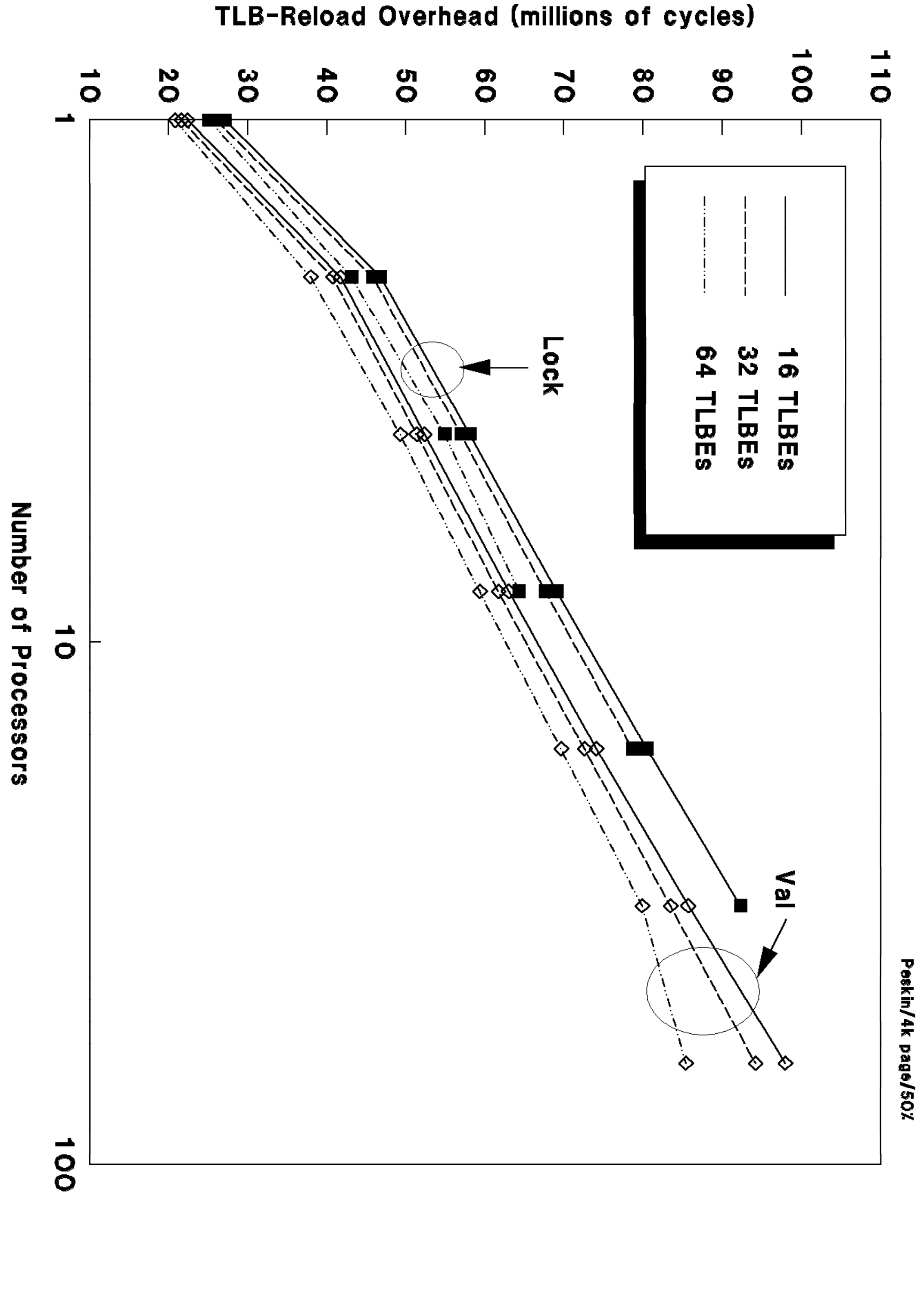
Graph 26. OH_{Lock} **and** OH_{Val} **vs.** N **– Simple 33%**



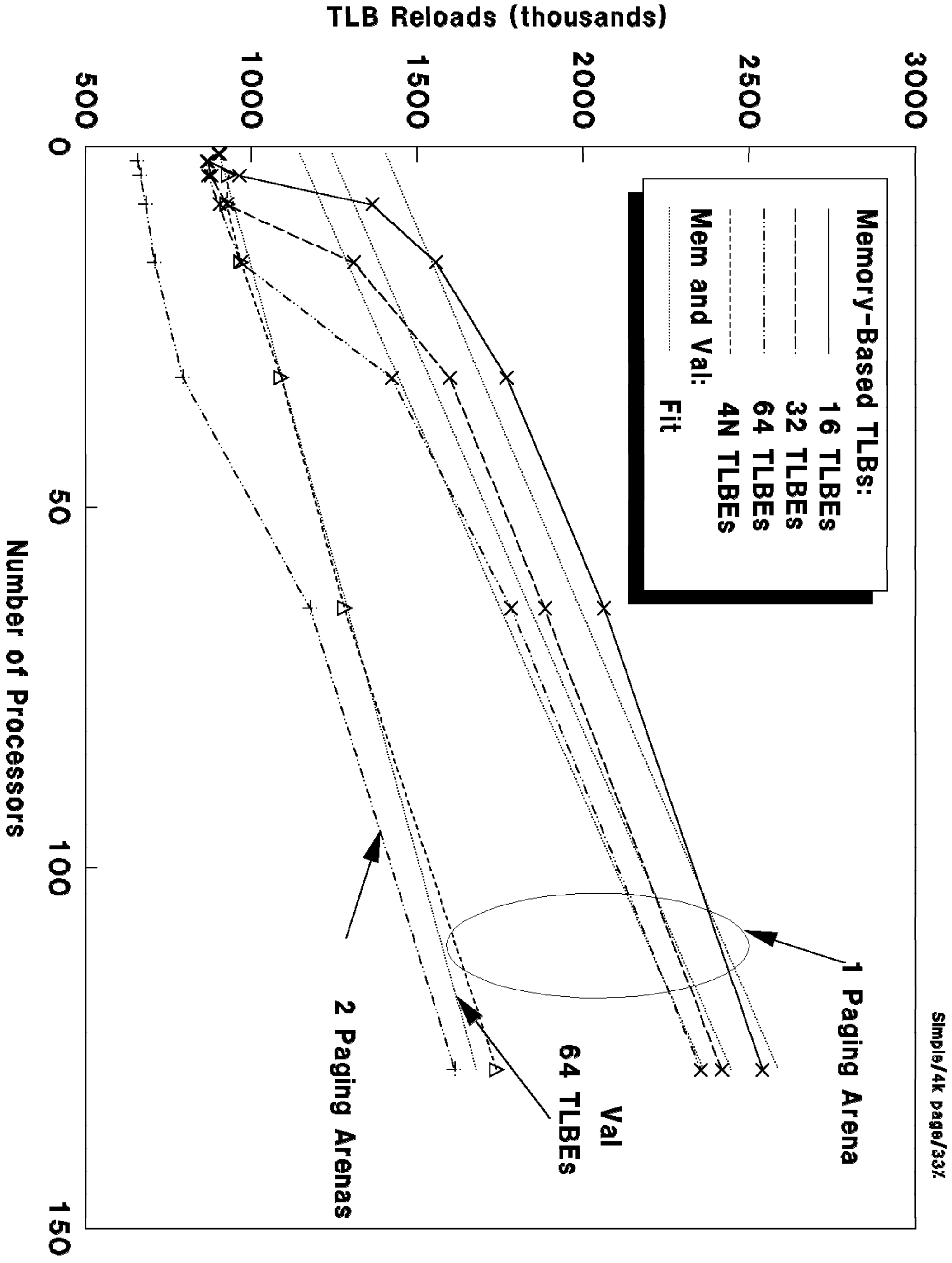
Graph 27. OH_{Lock} **and** OH_{Val} **vs.** N **– Simple 50%**



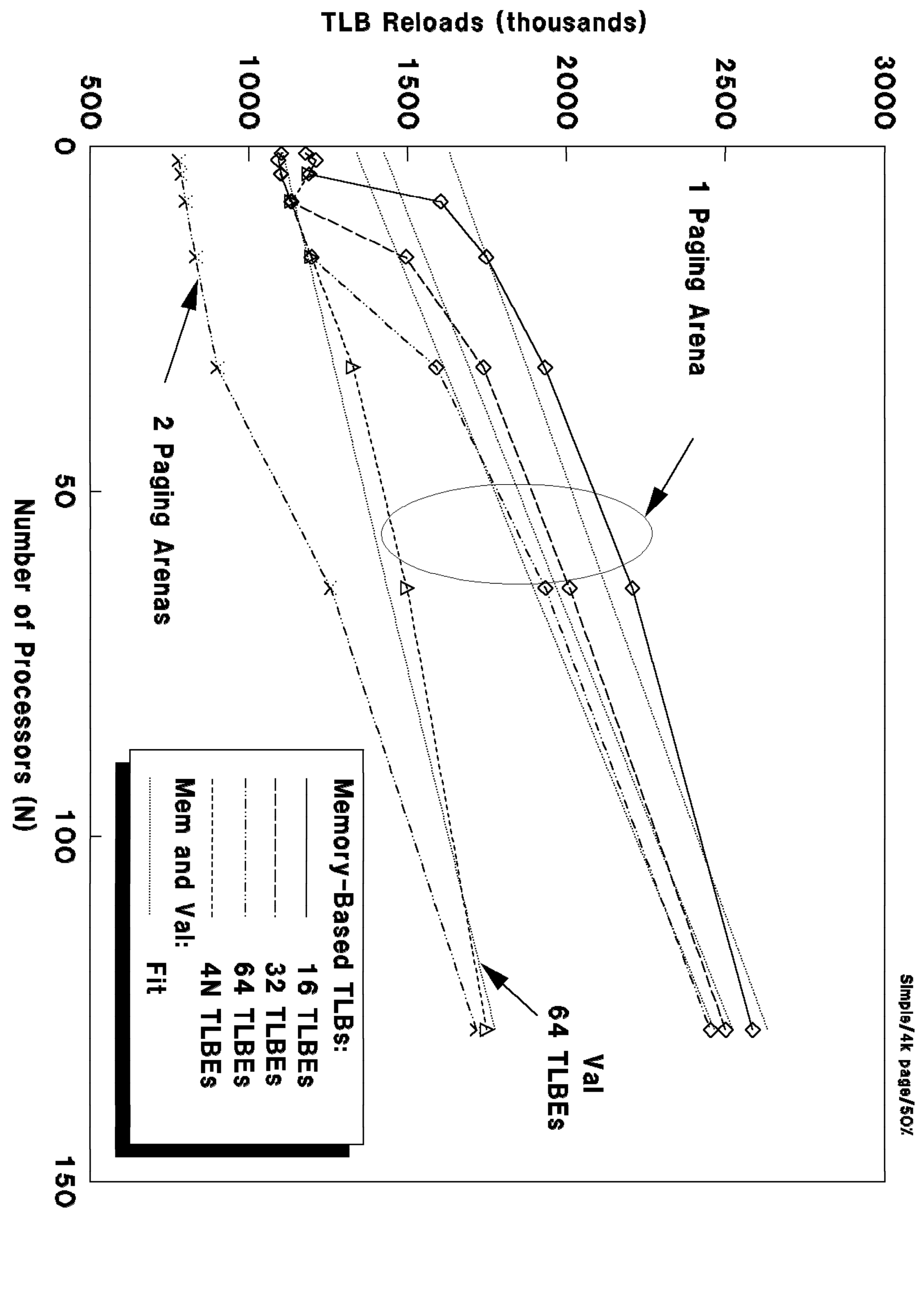
Graph 28. OH_{Lock} **and** OH_{Val} **vs.** N **– Peskin 33%**



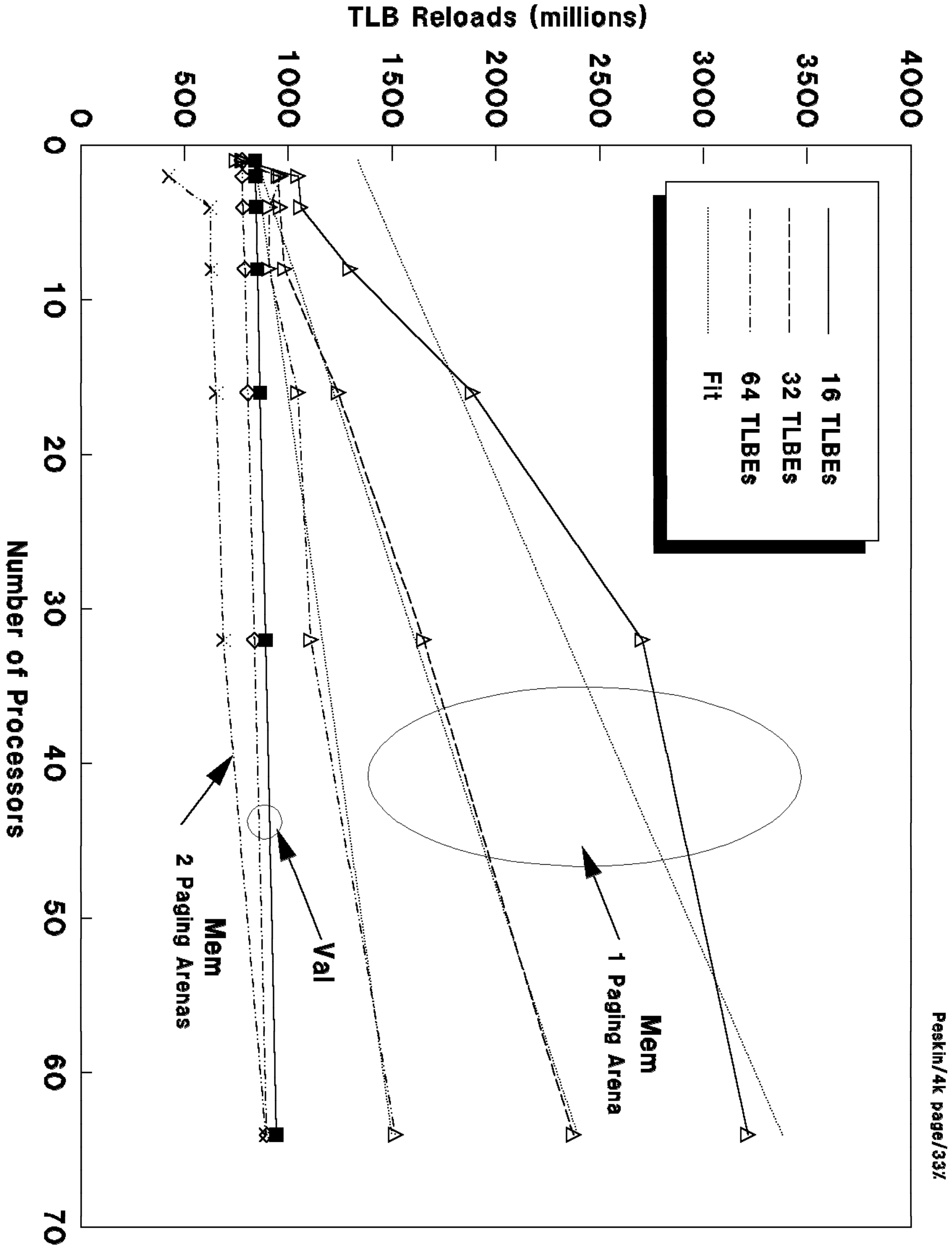
Graph 29. OH_{Lock} **and** OH_{Val} **vs.** N **– Peskin 50%**



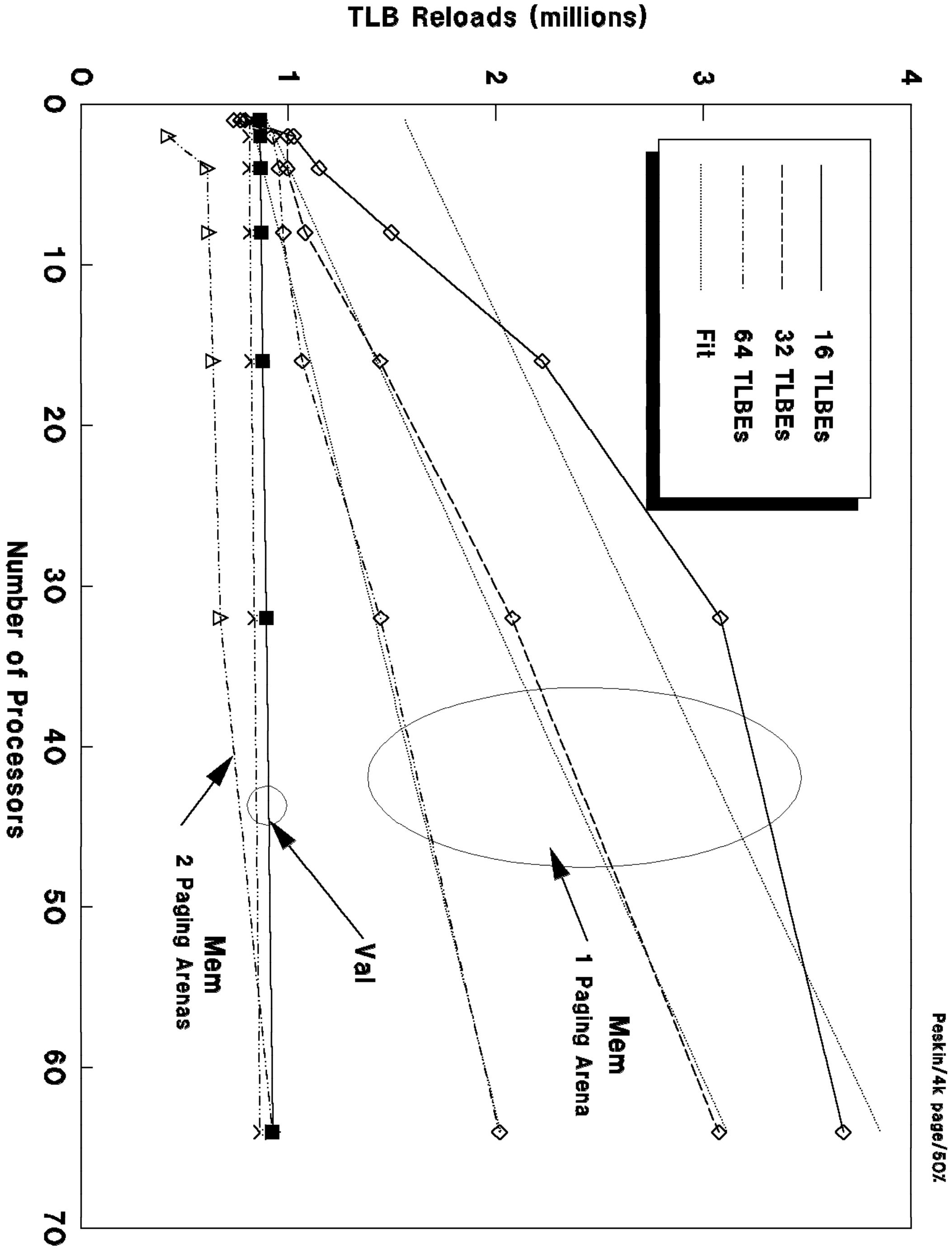
Graph 30. R_{Mem} **vs.** N**–Simple 33%**



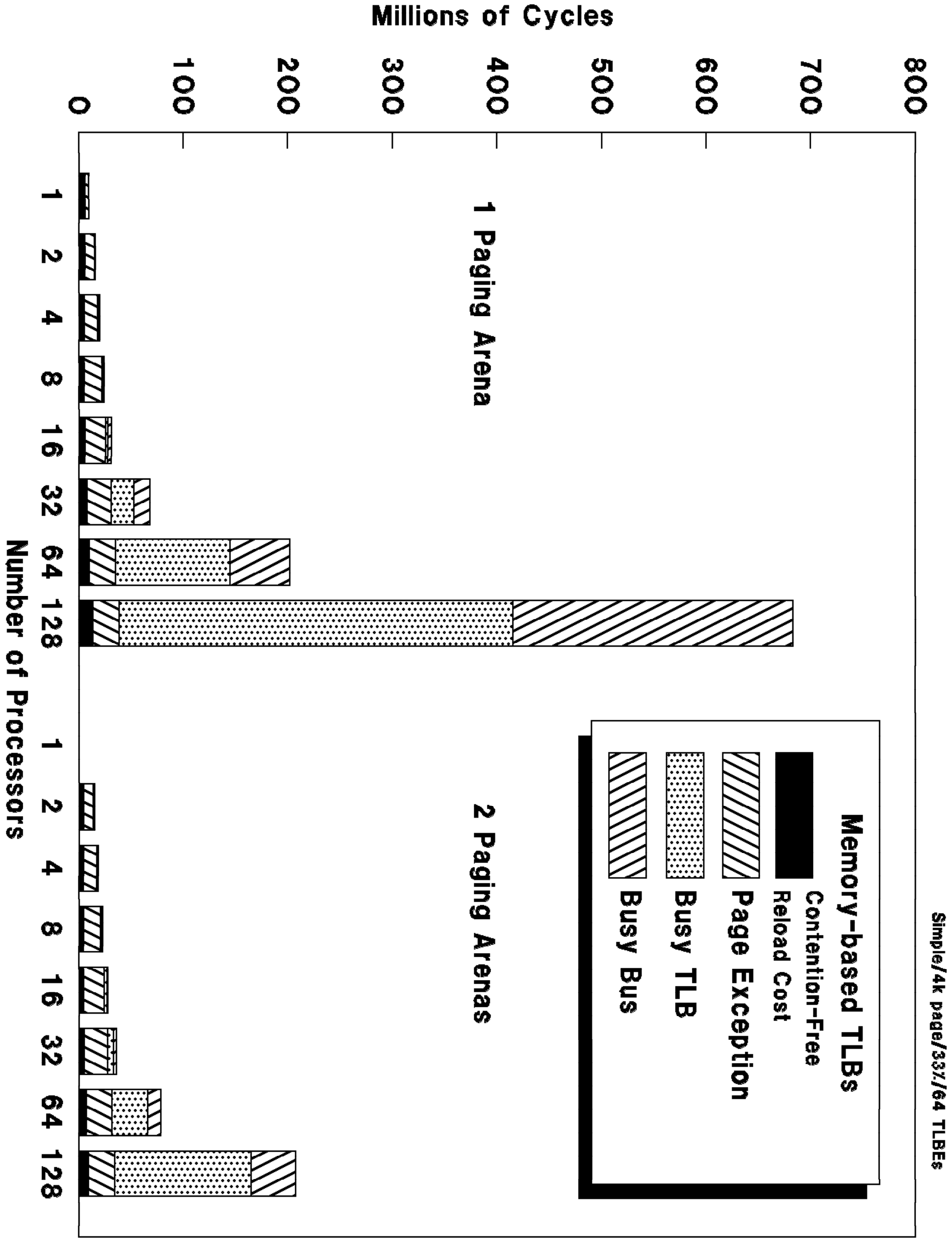
Graph 31. R_{Mem} **vs.** N**–Simple 50%**



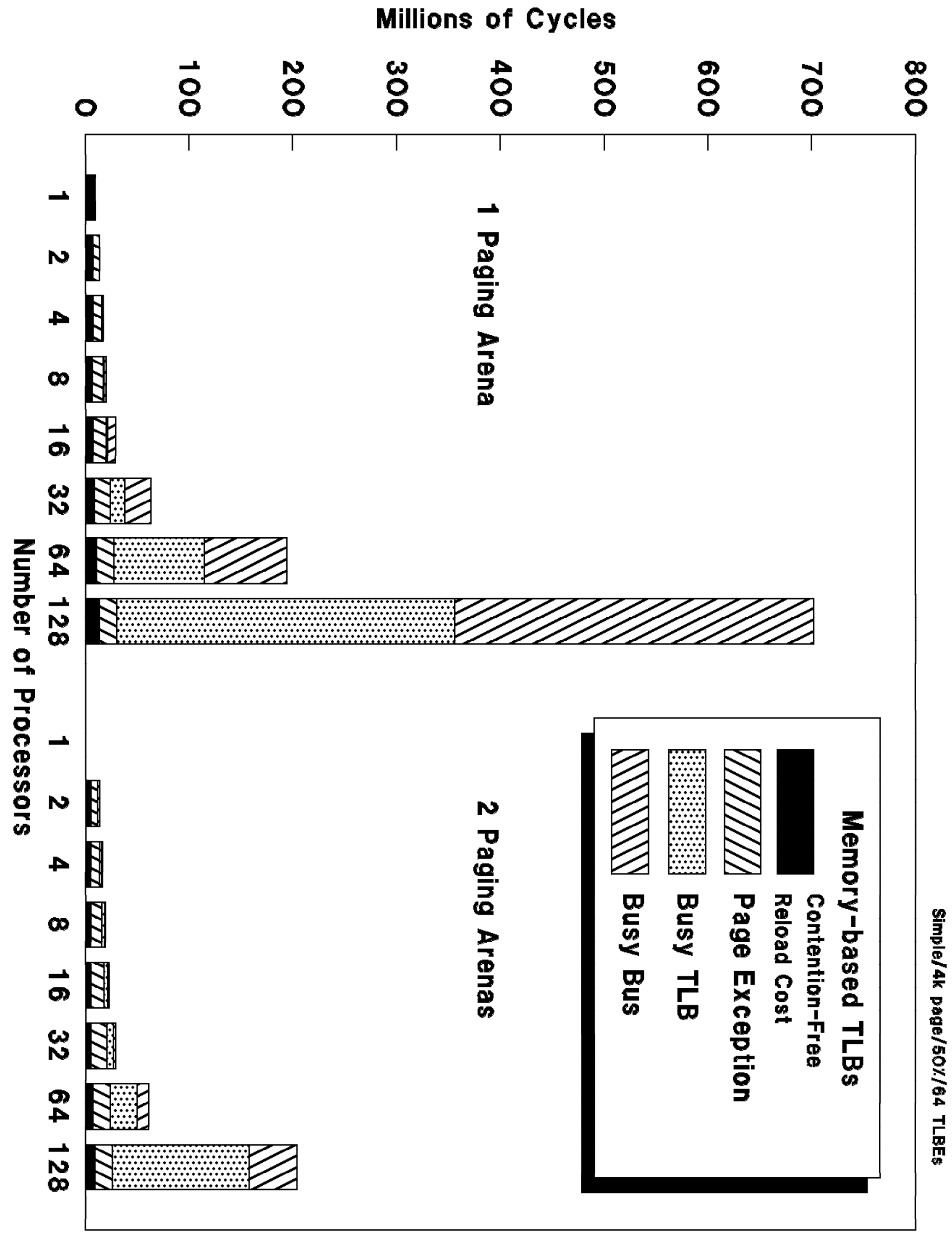
Graph 32. R_{Mem} **vs.** N**–Peskin 33%**



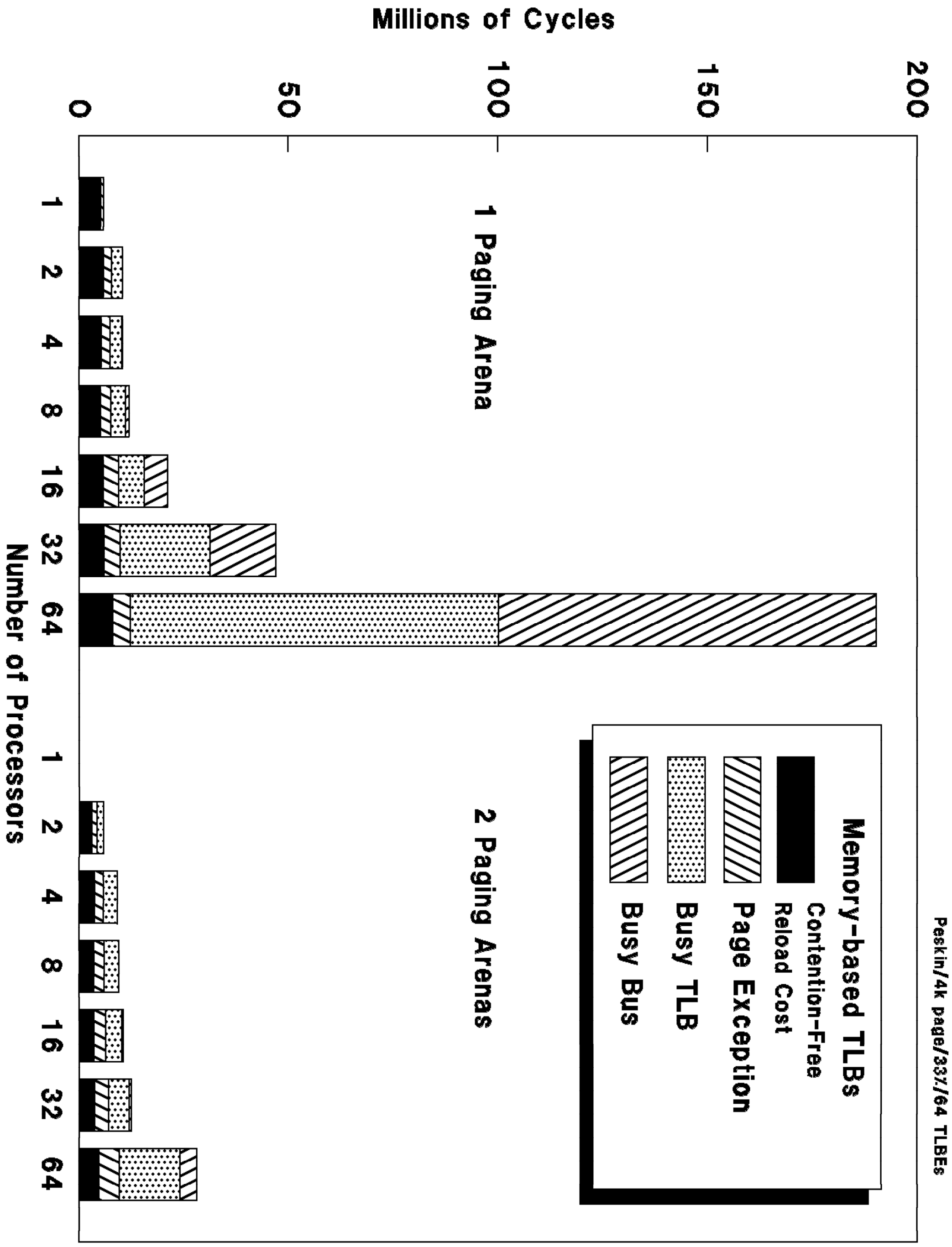
Graph 33. R_{Mem} **vs.** N**–Peskin 50%**



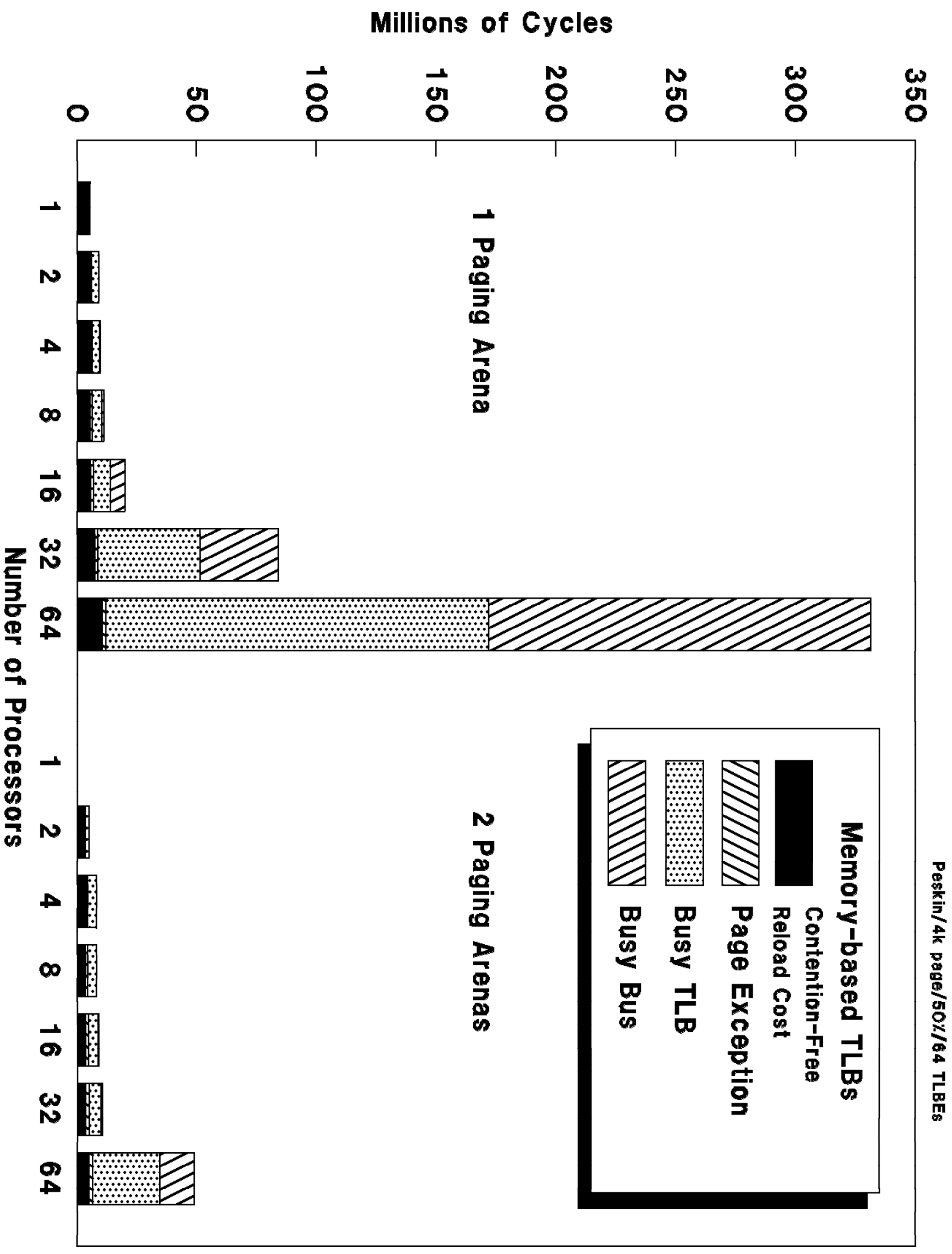
Graph 34. Components of OH_{Mem} vs. N–Simple 33%



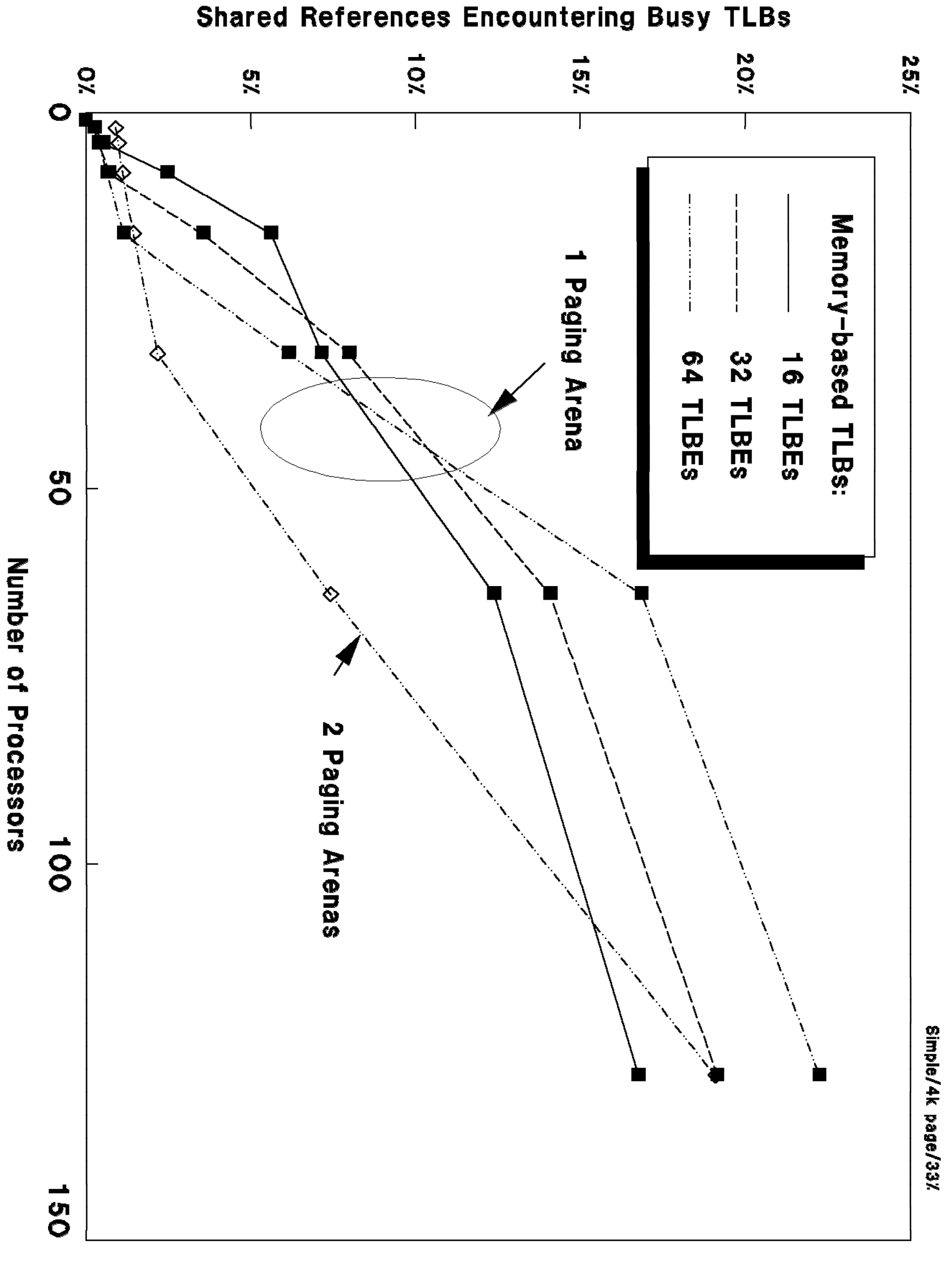
Graph 35. Components of OH_{Mem} vs. N–Simple 50%



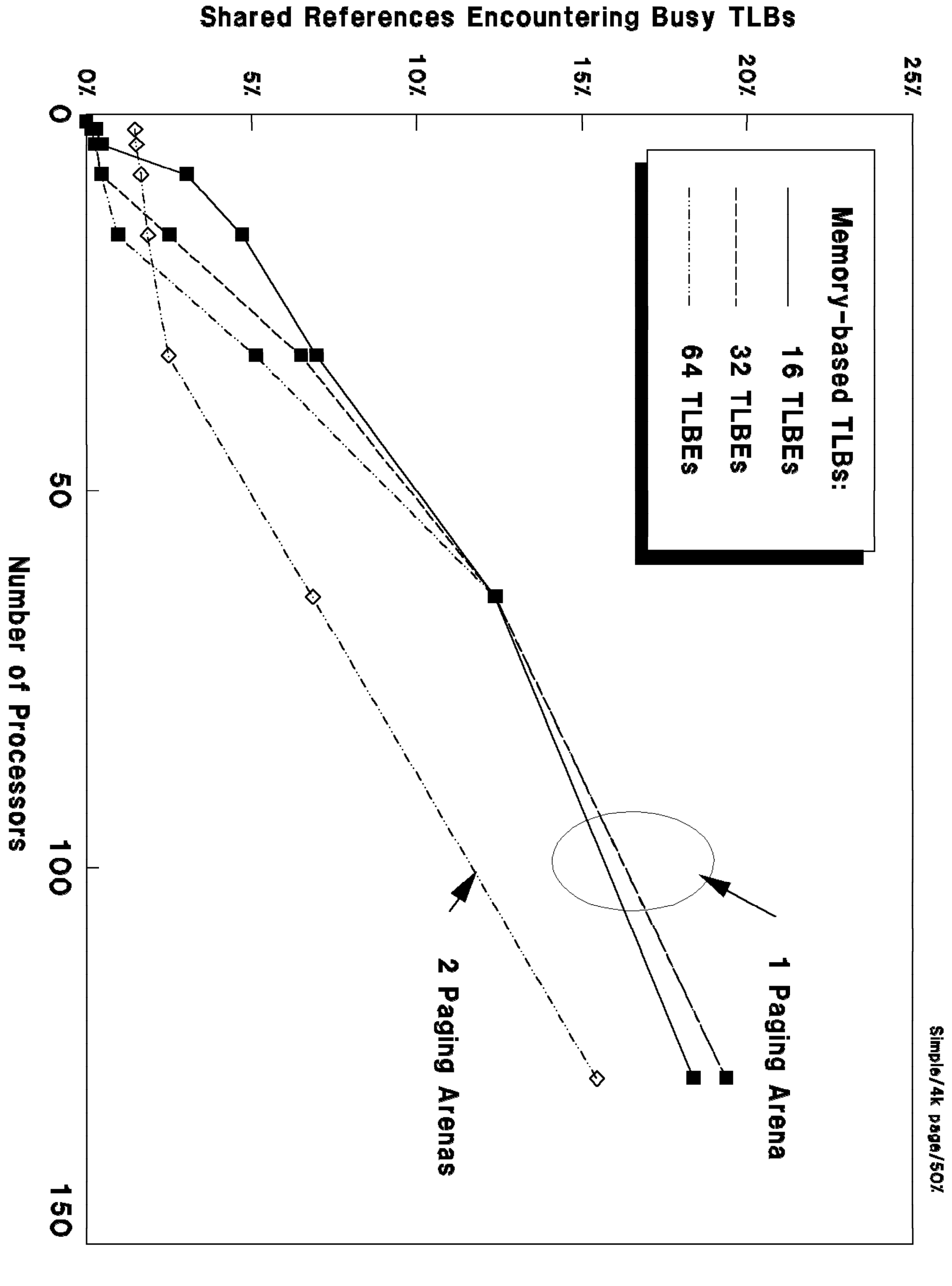
Graph 36. Components of OH_{Mem} vs. N–Peskin 33%



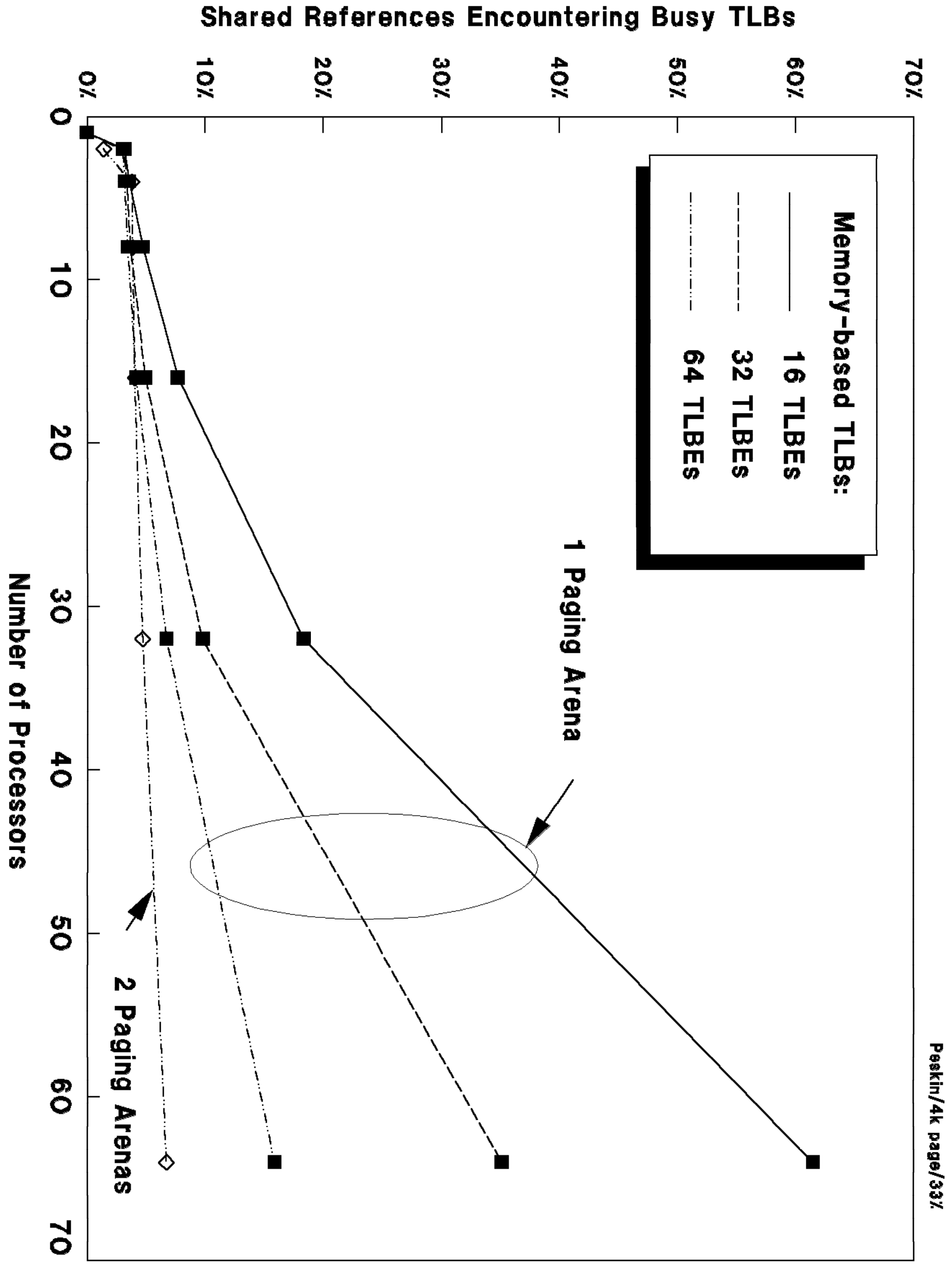
Graph 37. Components of OH_{Mem} vs. N–Peskin 50%



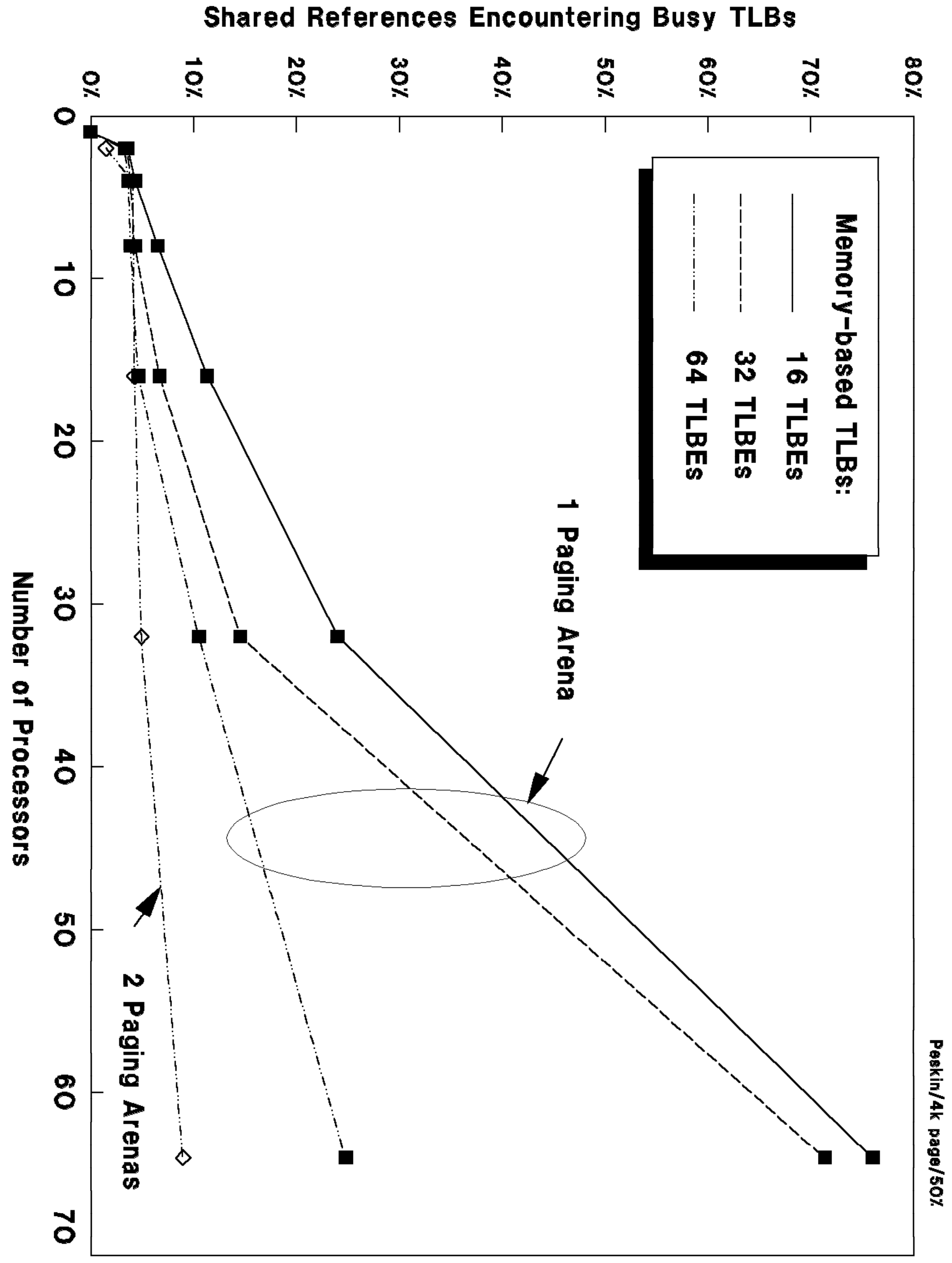
Graph 38. Shared References for Busy TLBs vs. *N*–Simple 33%



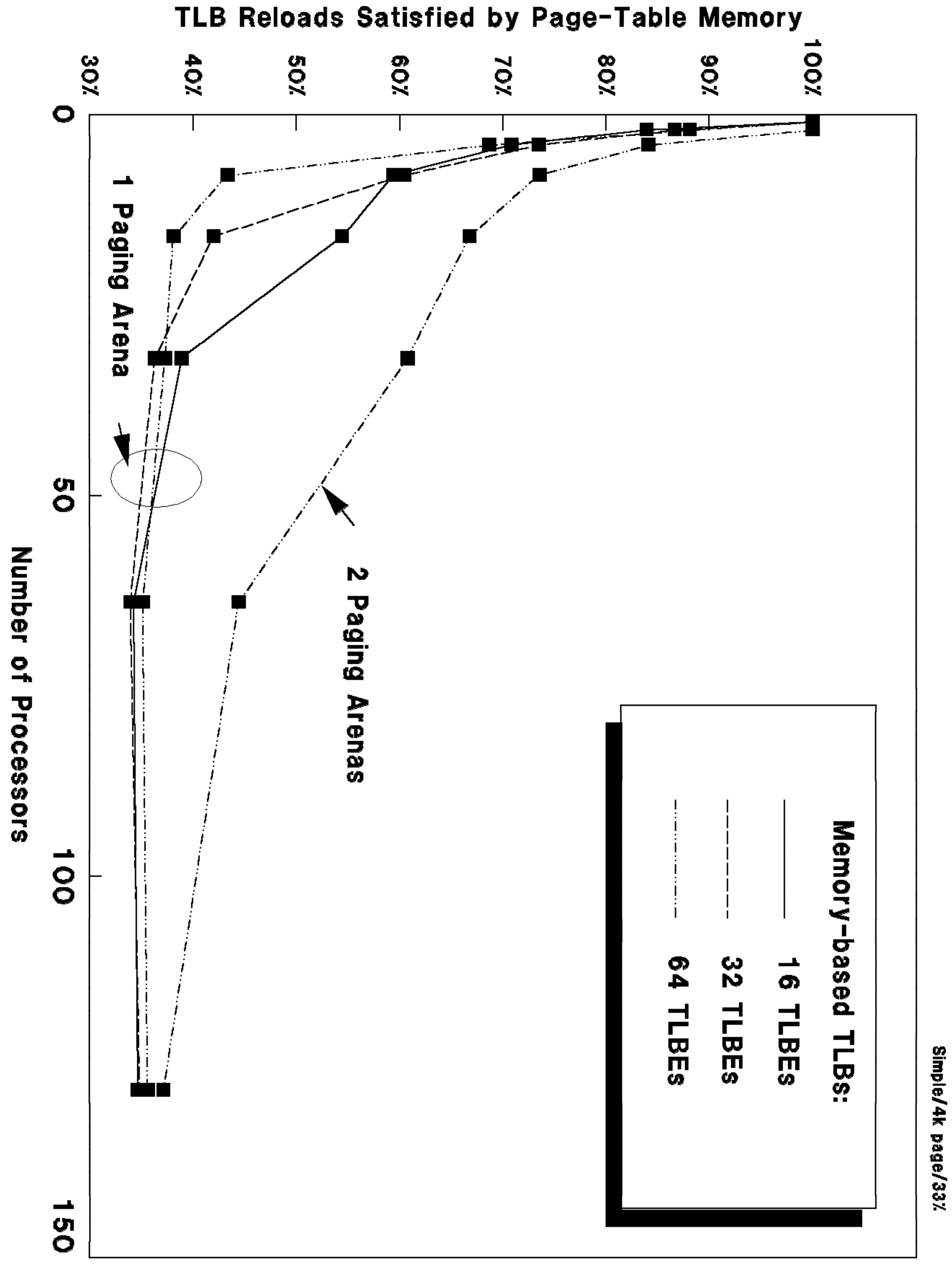
Graph 39. Shared References for Busy TLBs vs. *N*–Simple 50%



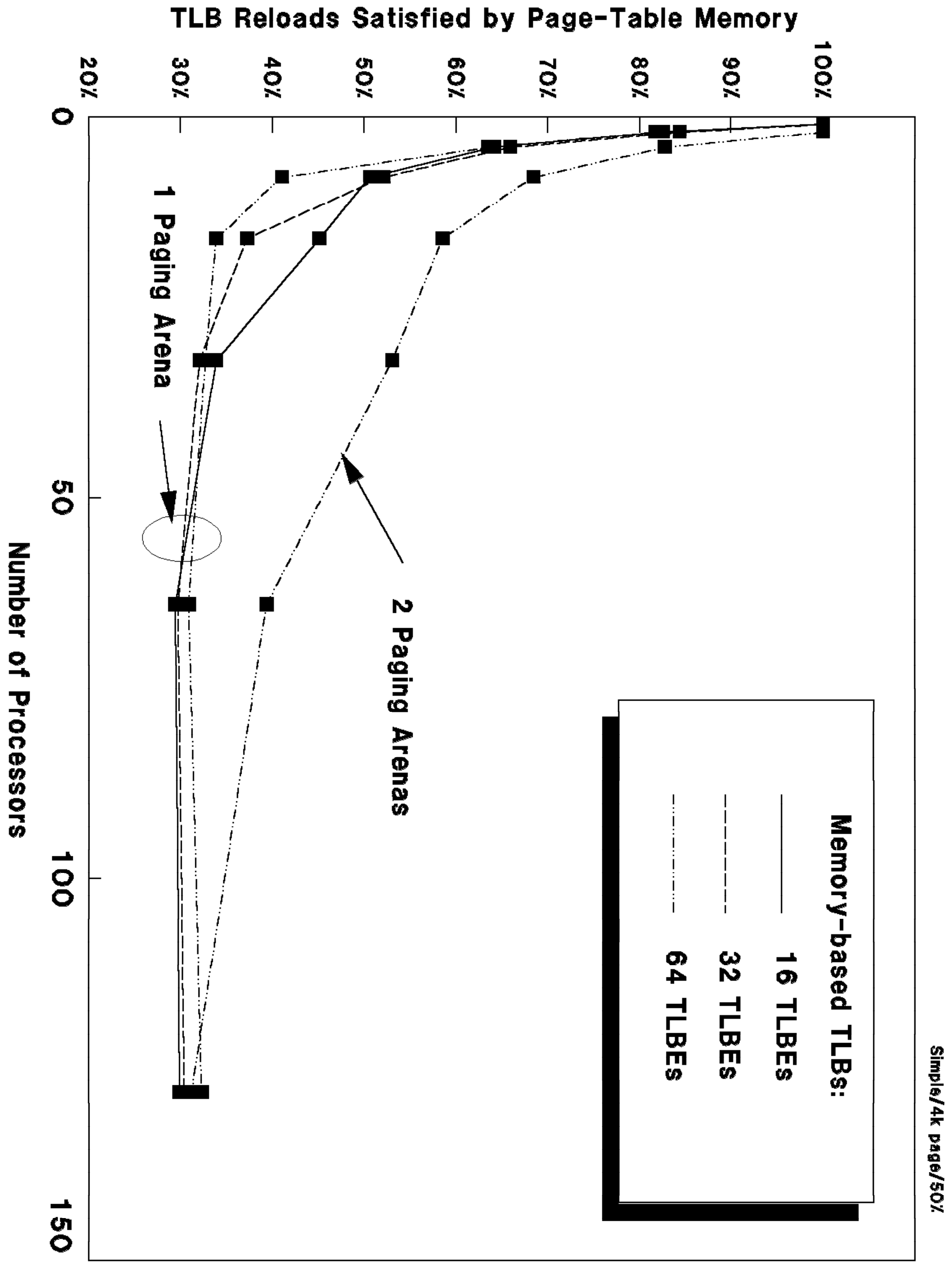
Graph 40. Shared References for Busy TLBs vs. *N*–Peskin 33%



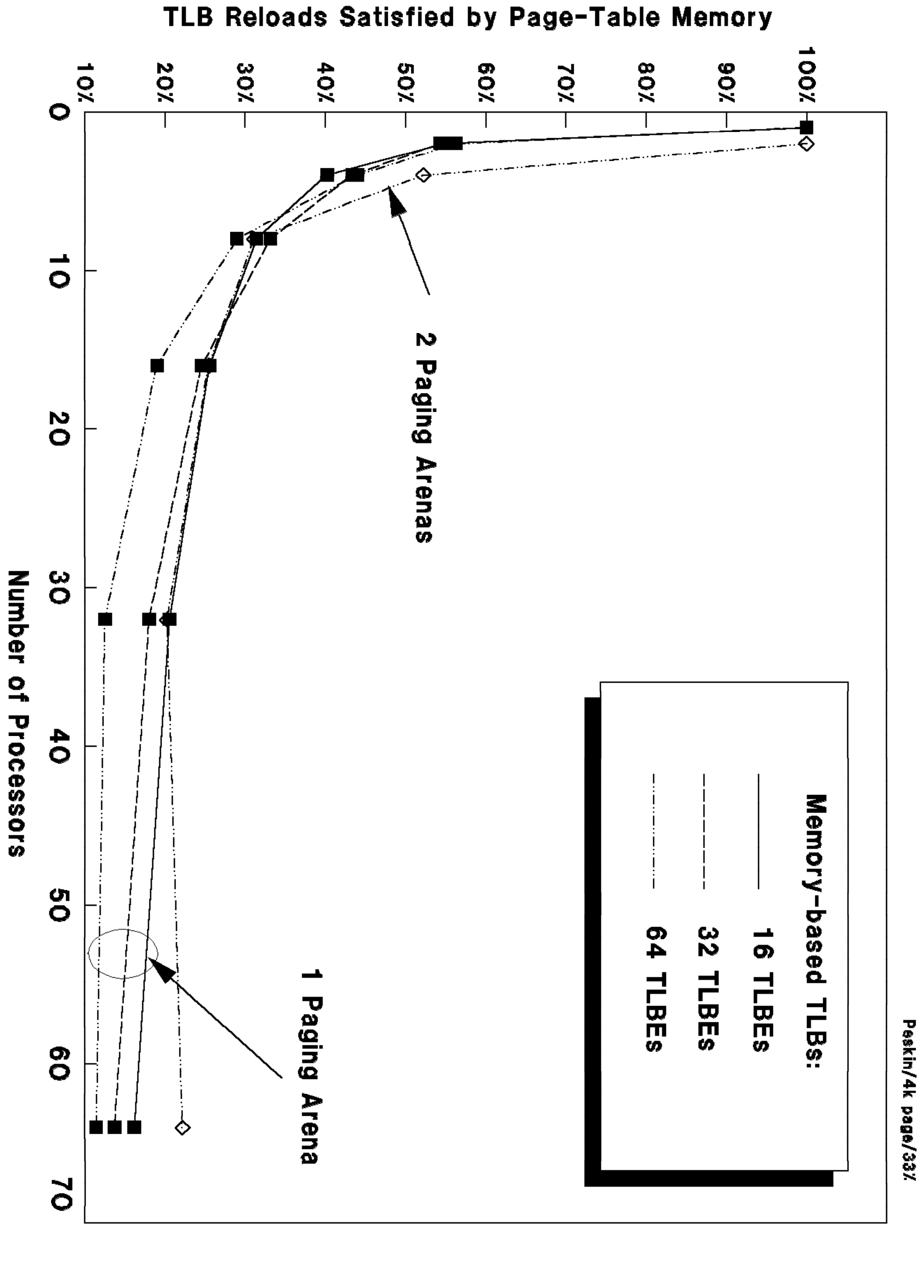
Graph 41. Shared References for Busy TLBs vs. *N*–Peskin 50%



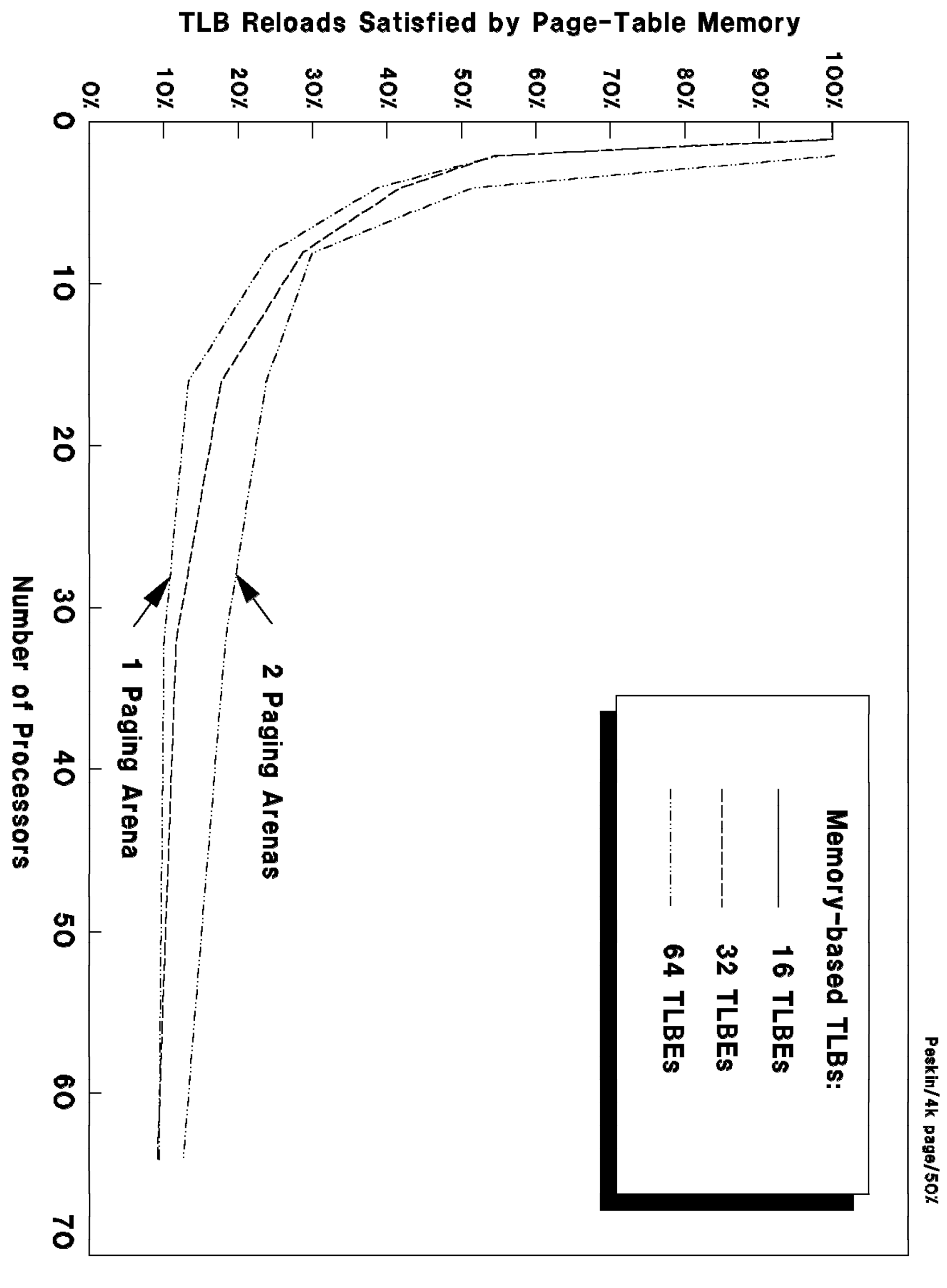
Graph 42. TLB Reloads via Page-Table Memory vs. *N*–Simple 33%



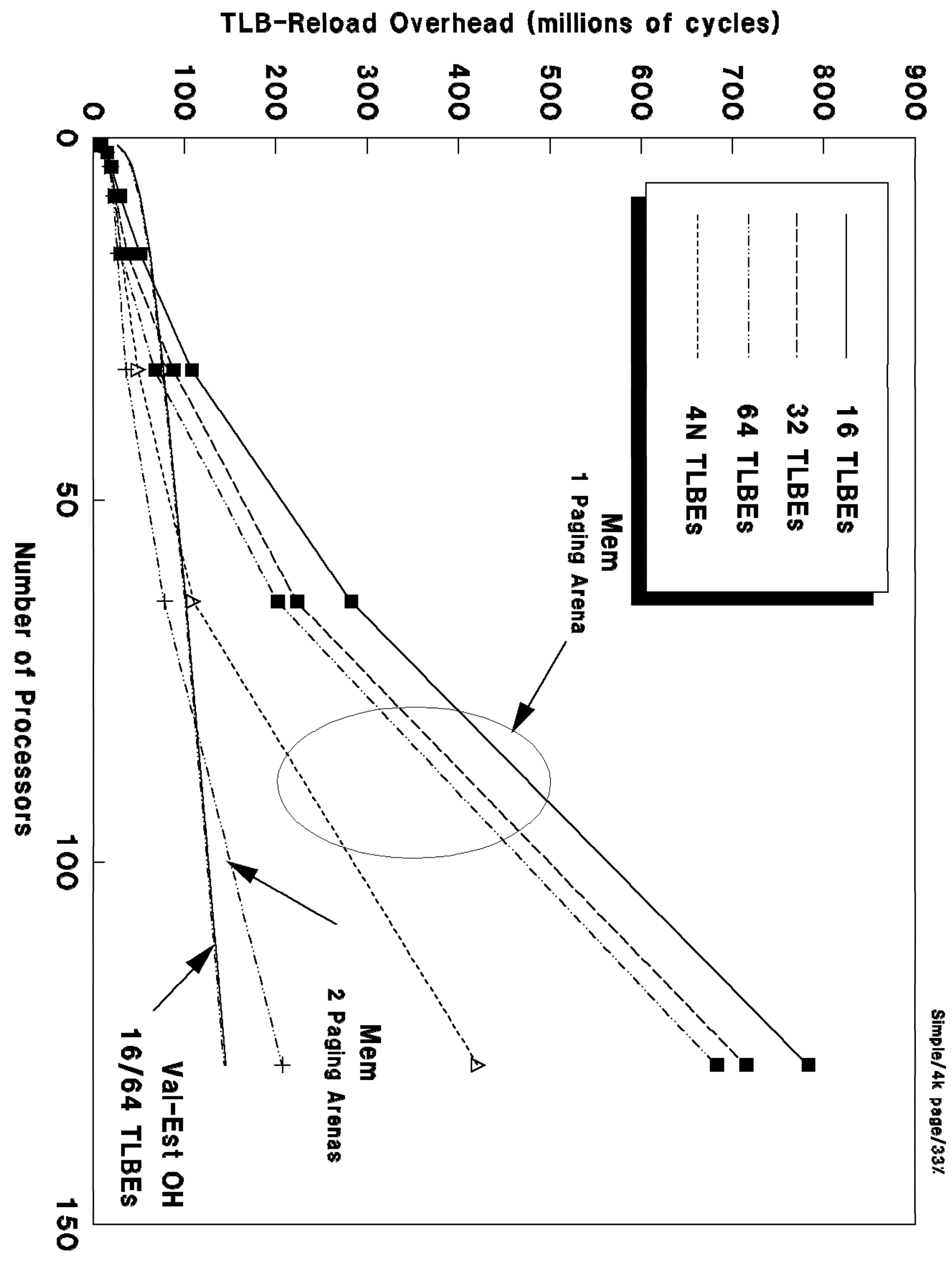
Graph 43. TLB Reloads via Page-Table Memory vs. *N*–Simple 50%



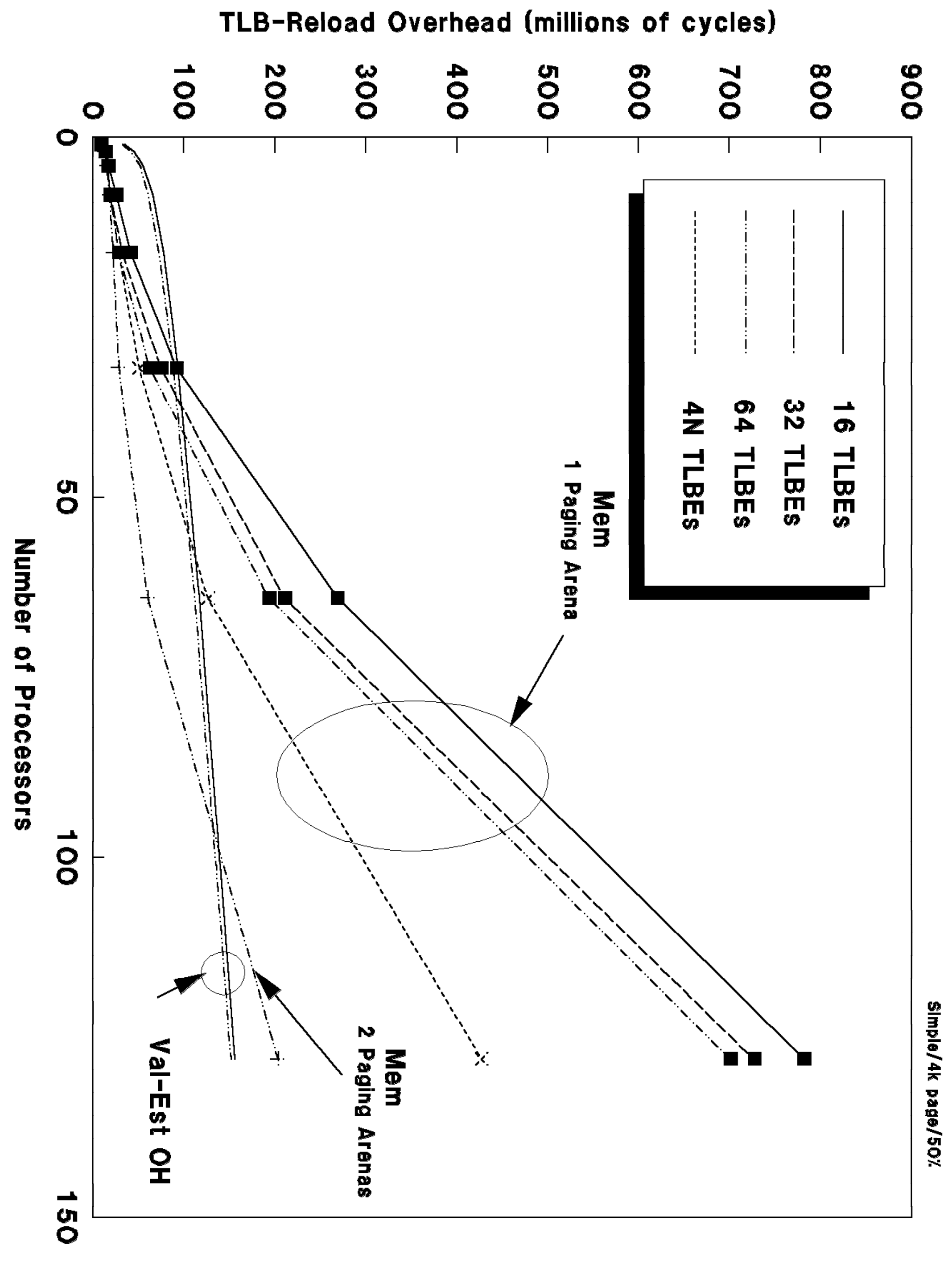
Graph 44. TLB Reloads via Page-Table Memory vs. *N*–Peskin 33%



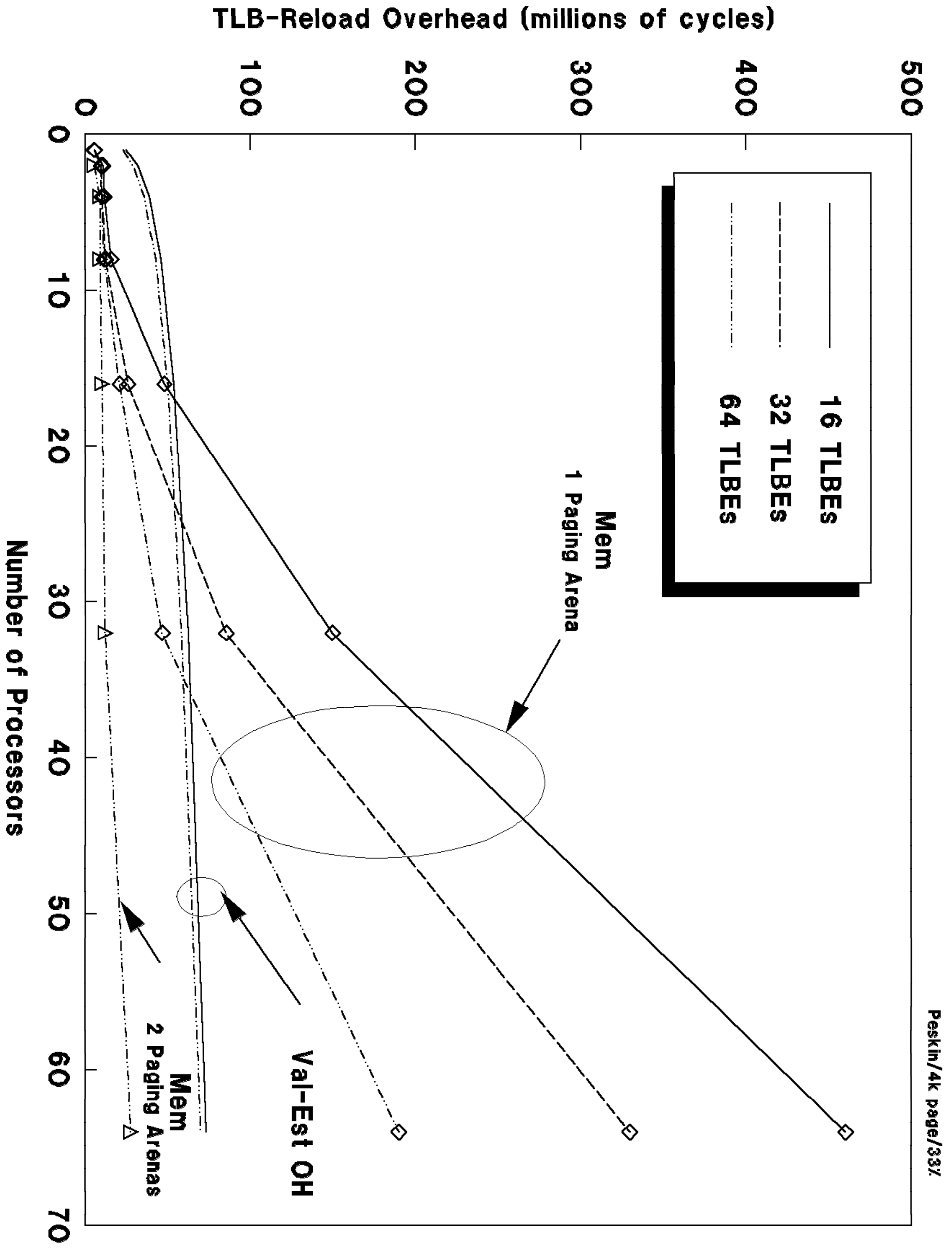
Graph 45. TLB Reloads via Page-Table Memory vs. *N*–Peskin 50%



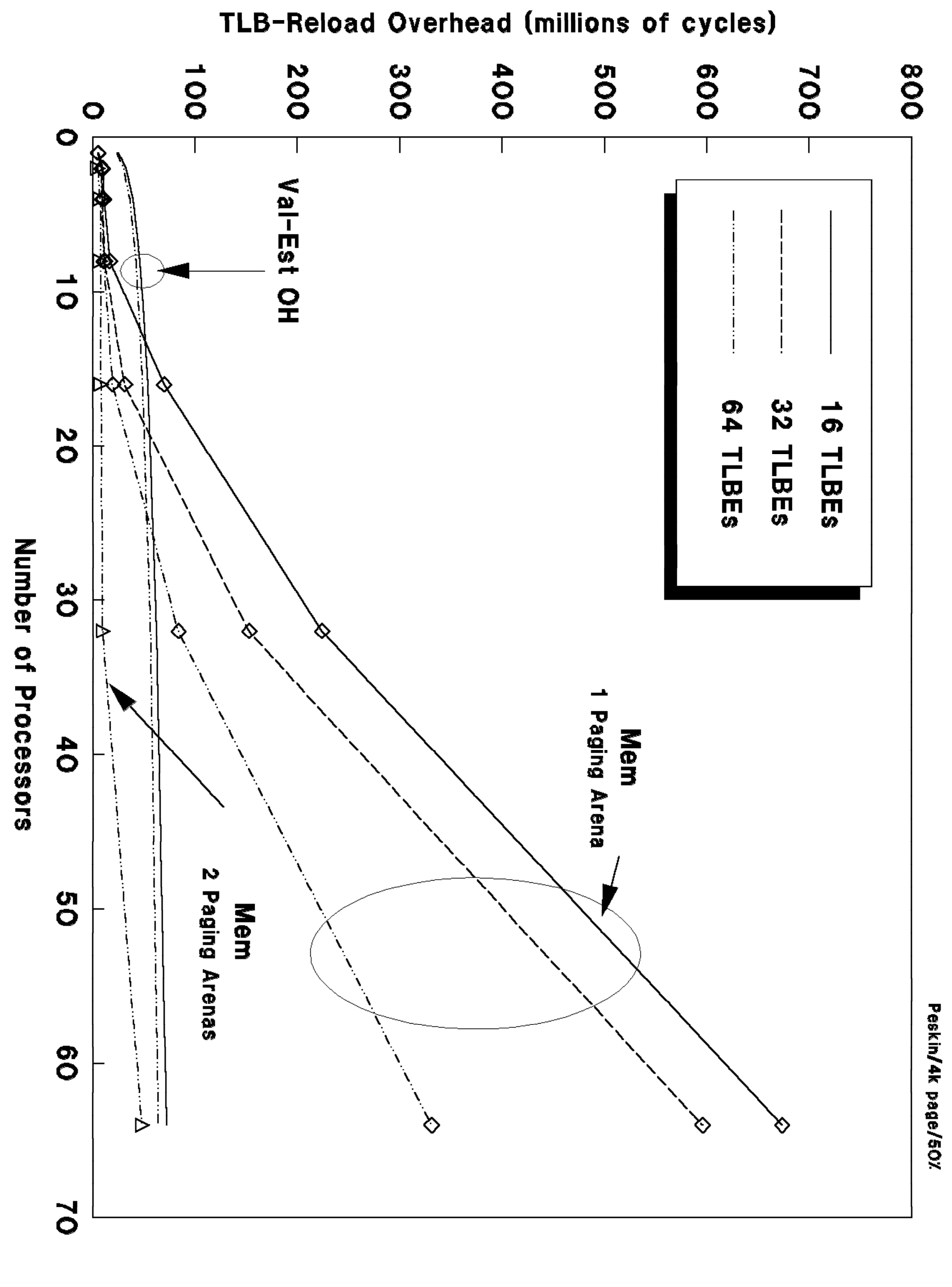
Graph 46. ***OH_{Mem}* vs. *N*–Simple 33%**



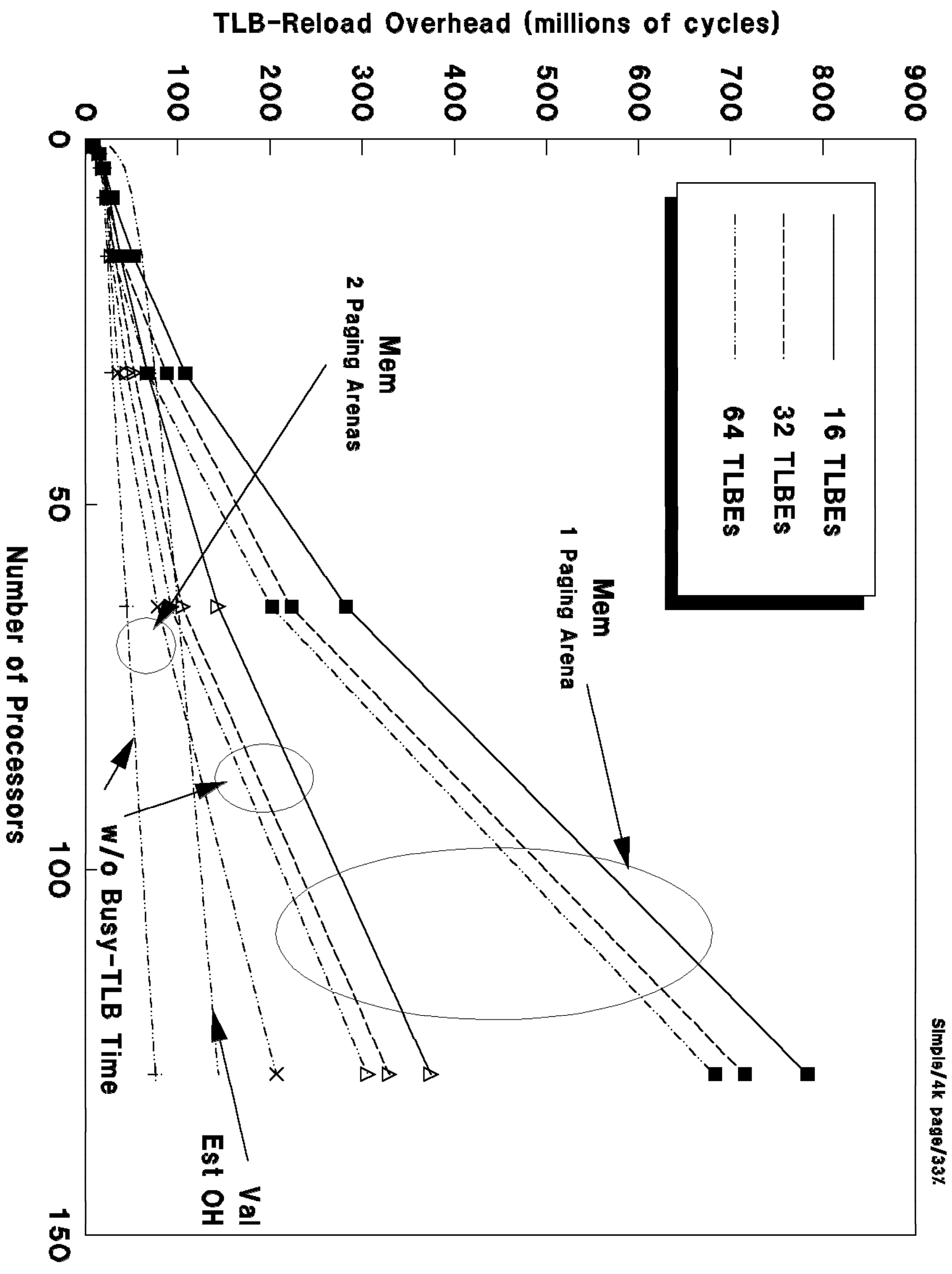
Graph 47. ***OH_{Mem}*** **vs.** ***N*****–Simple 50%**



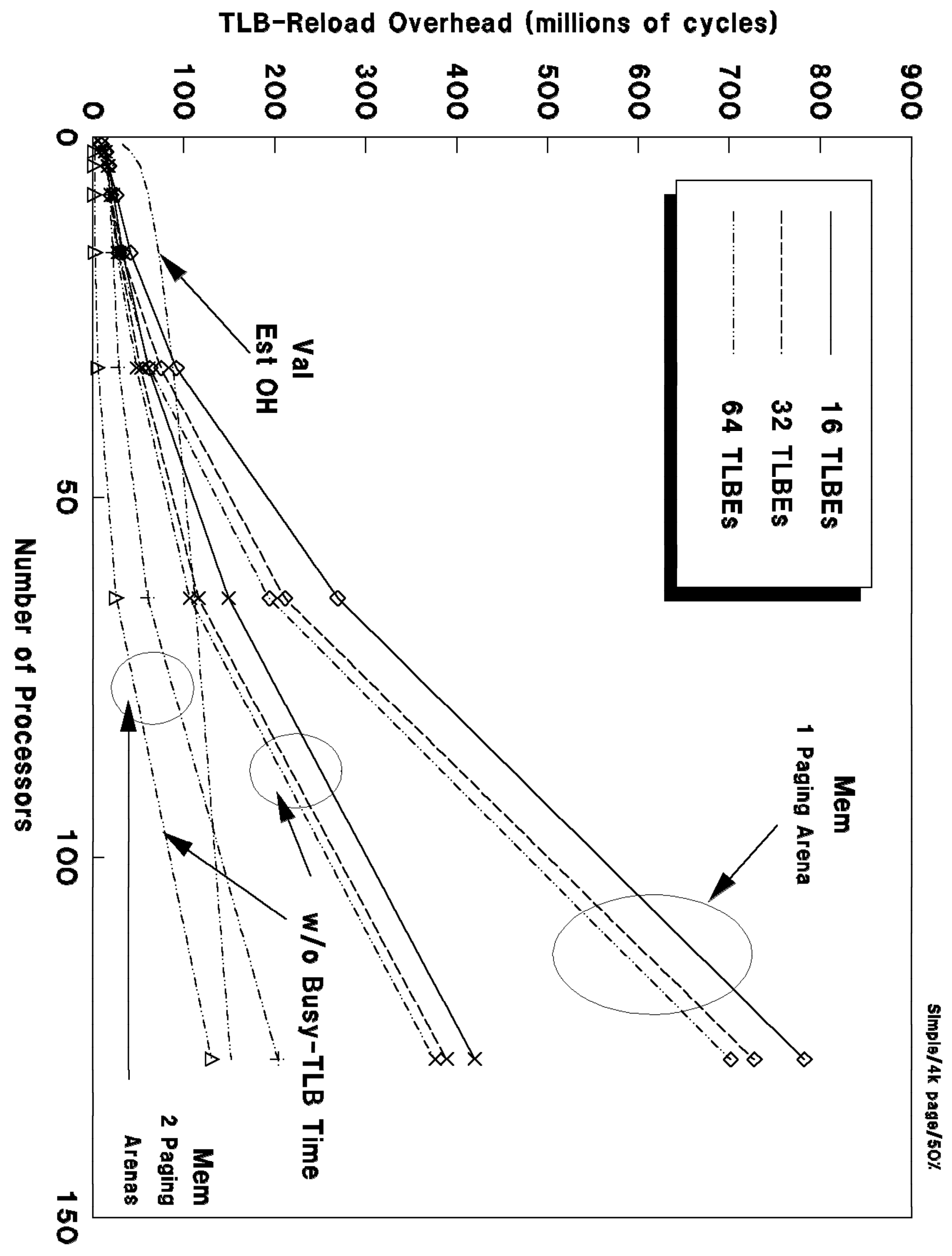
Graph 48. ***OH_{Mem}*** **vs.** ***N*****–Peskin 33%**



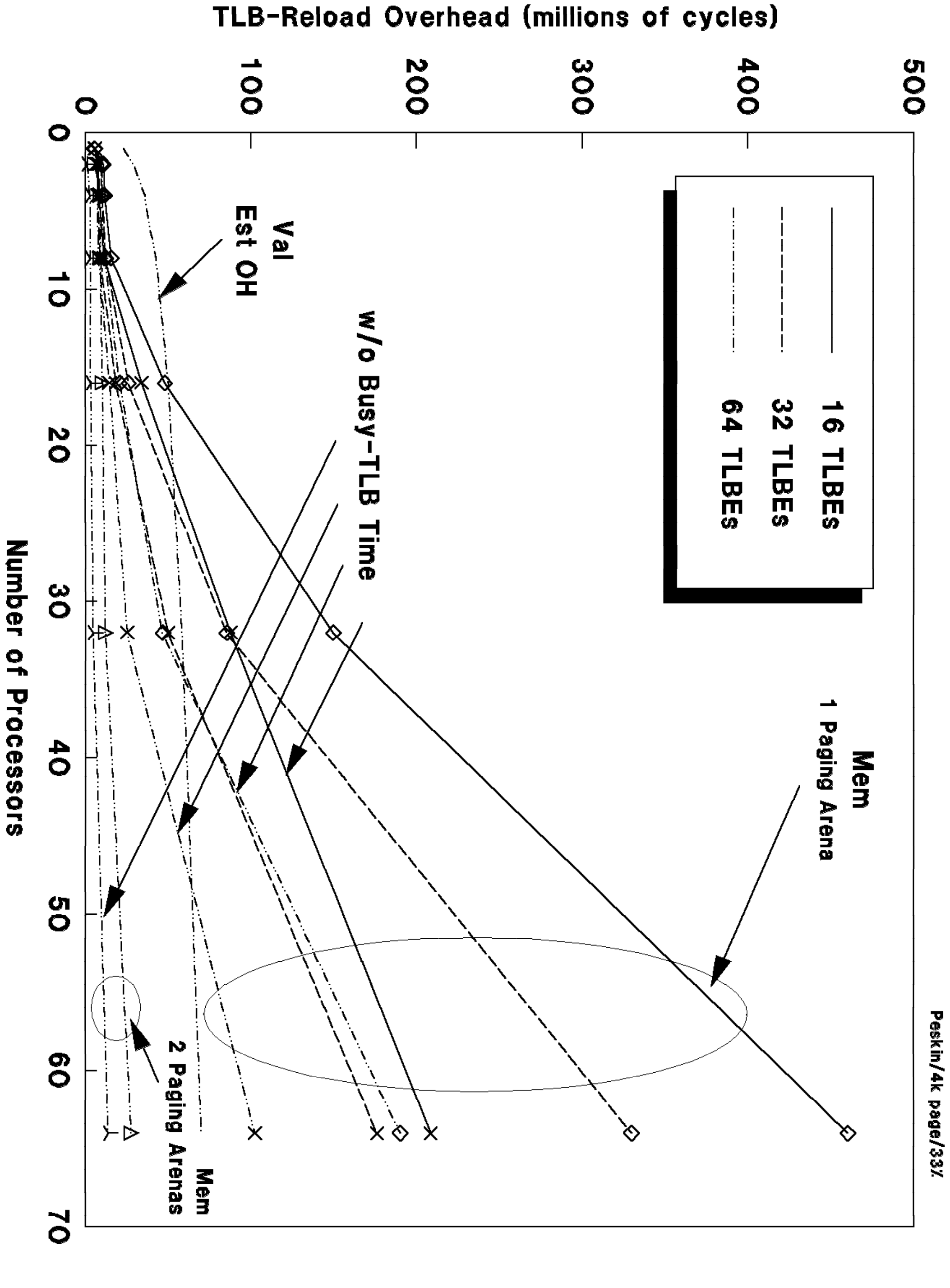
Graph 49. OH_{Mem} **vs.** N**–Peskin 50%**



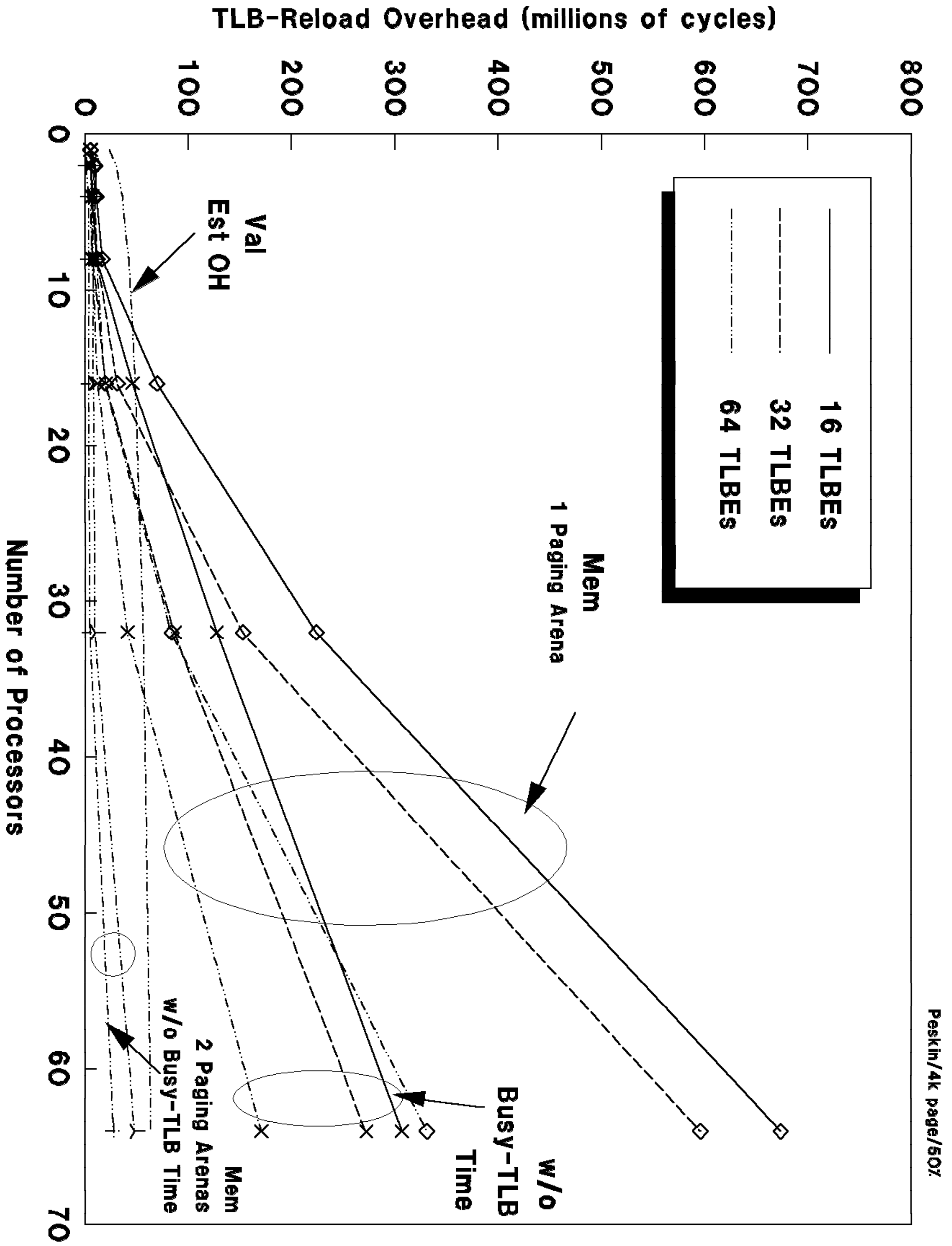
Graph 50. ***OH*$_{Mem}$ with and without Busy-TLB Time vs. *N*–Simple 33%**



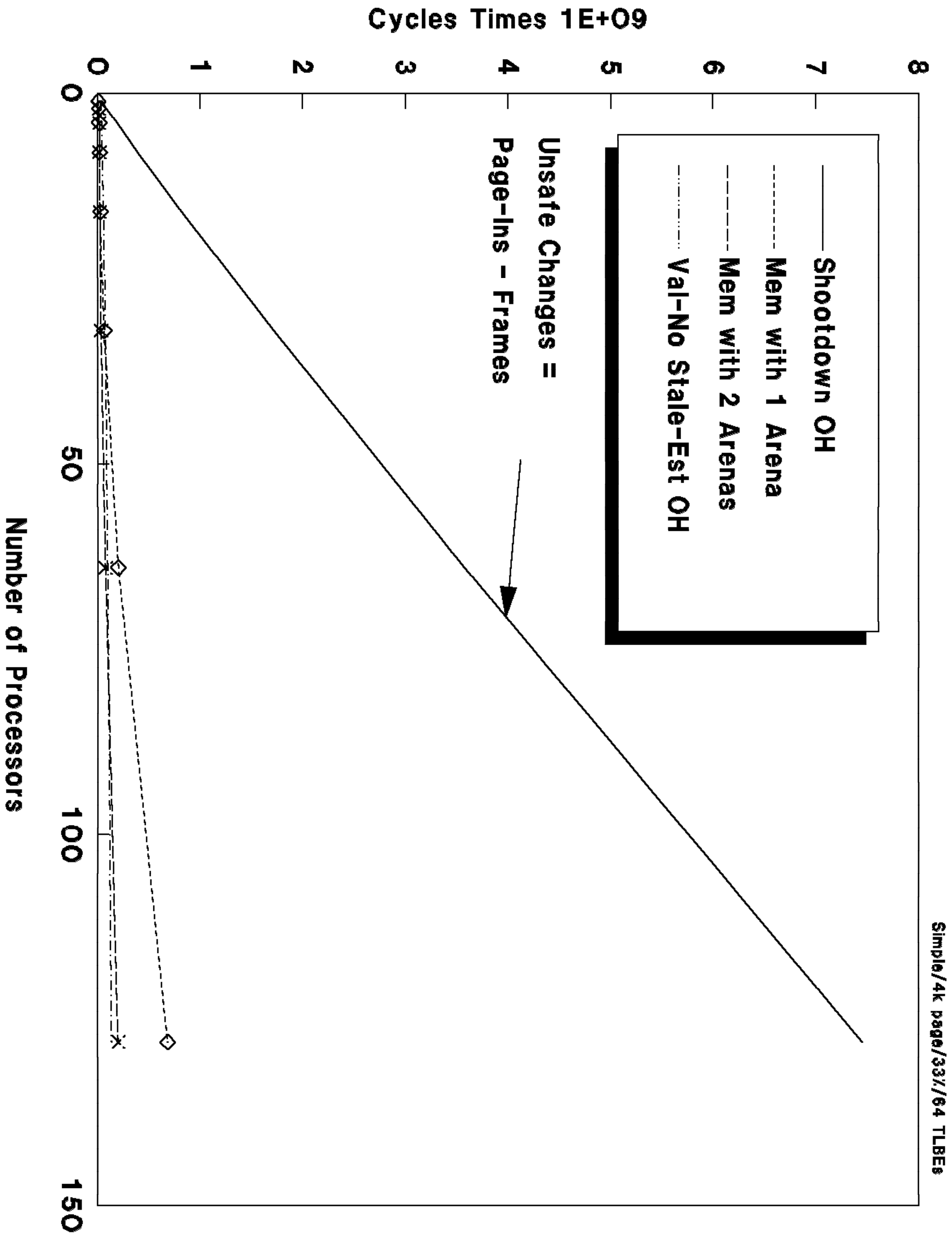
Graph 51. ***OH_{Mem}*** **with and without Busy-TLB Time vs.** ***N*****–Simple 50%**



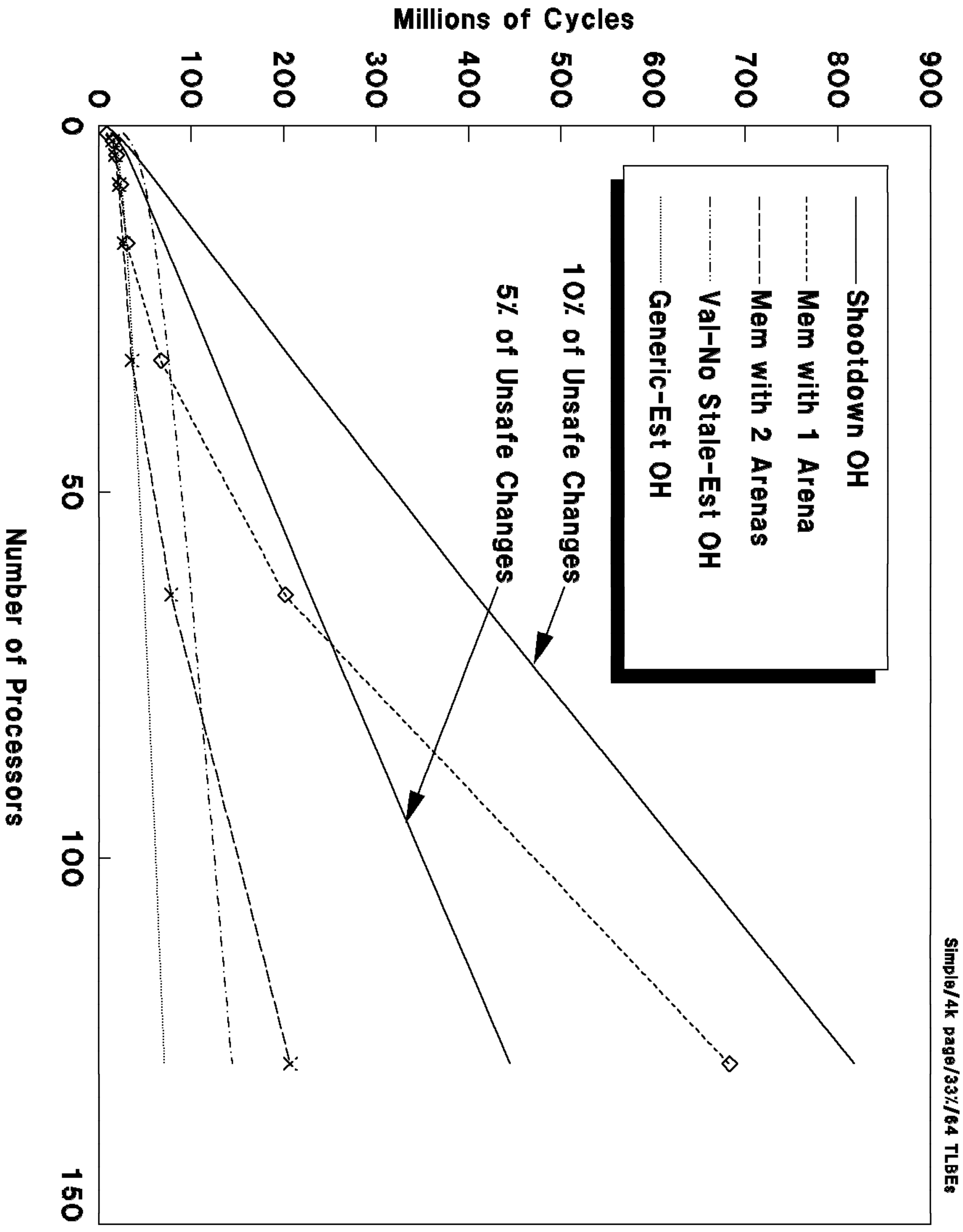
Graph 52. ***OH_{Mem}*** **with and without Busy-TLB Time vs.** ***N*****–Peskin 33%**



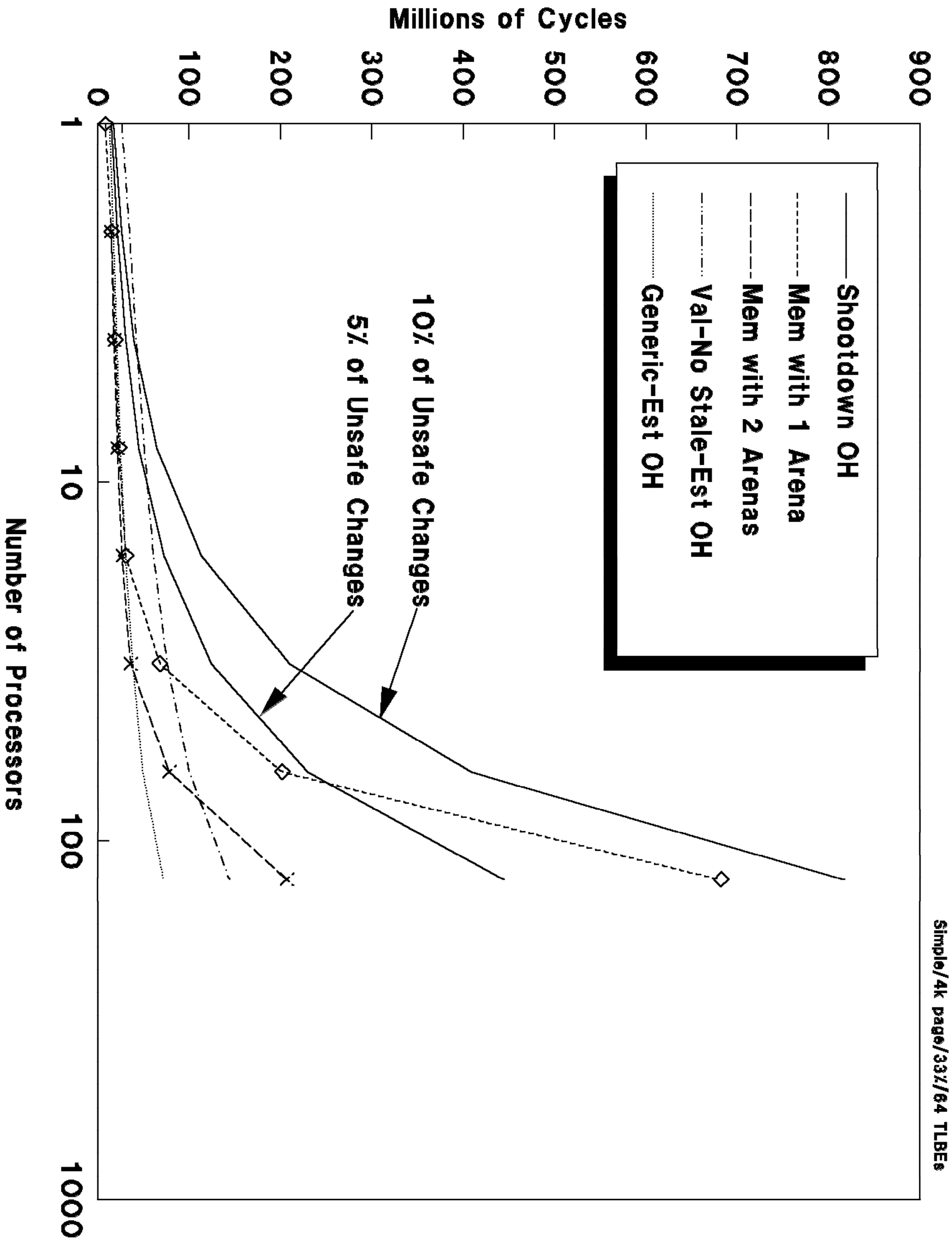
Graph 53. ***OH_{Mem}*** **with and without Busy-TLB Time vs.** ***N*****–Peskin 50%**



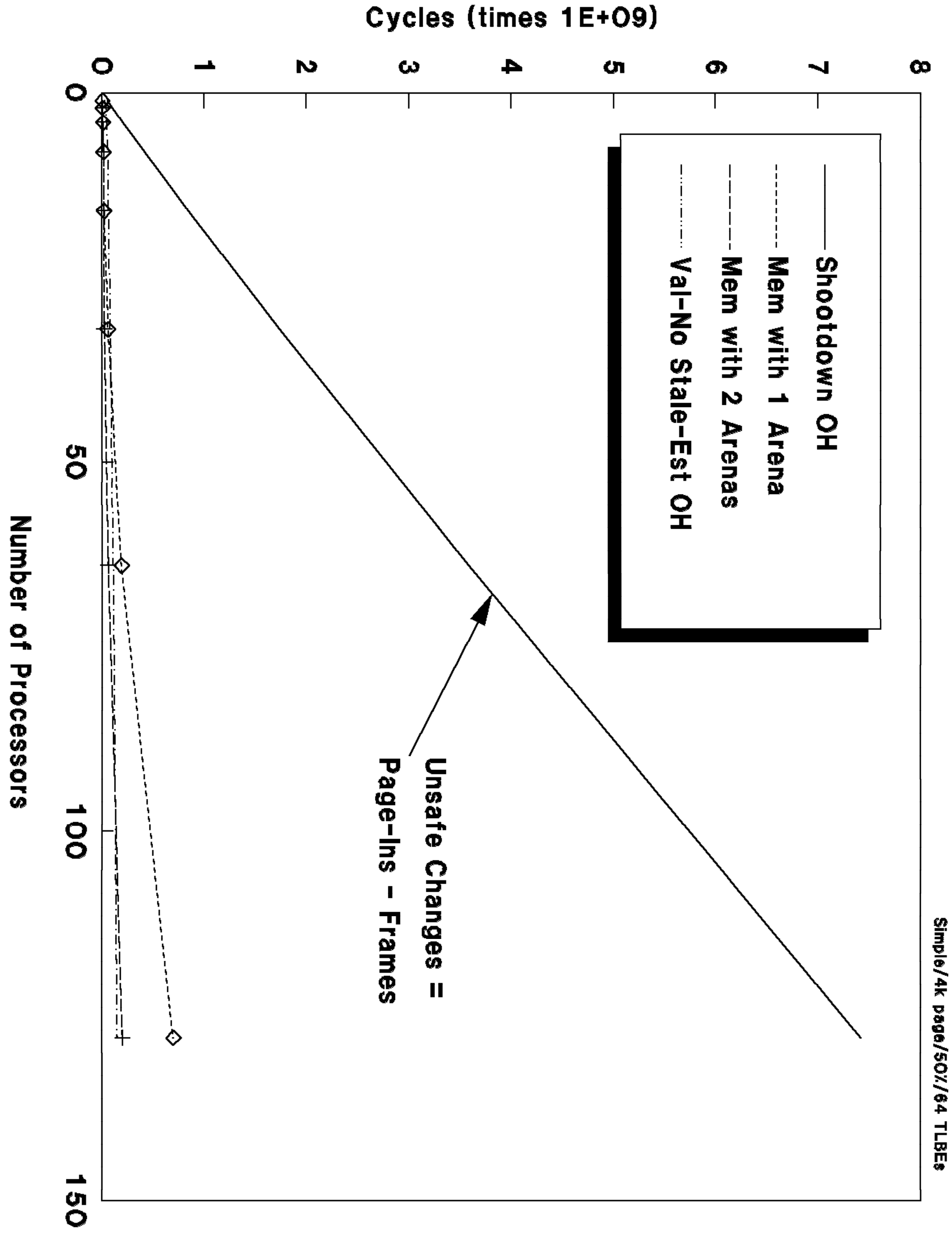
Graph 54. OH_{Shoot}, OH_{Val}, **and** OH_{Mem} **vs.** N **–Simple/33%/64 TLBEs**



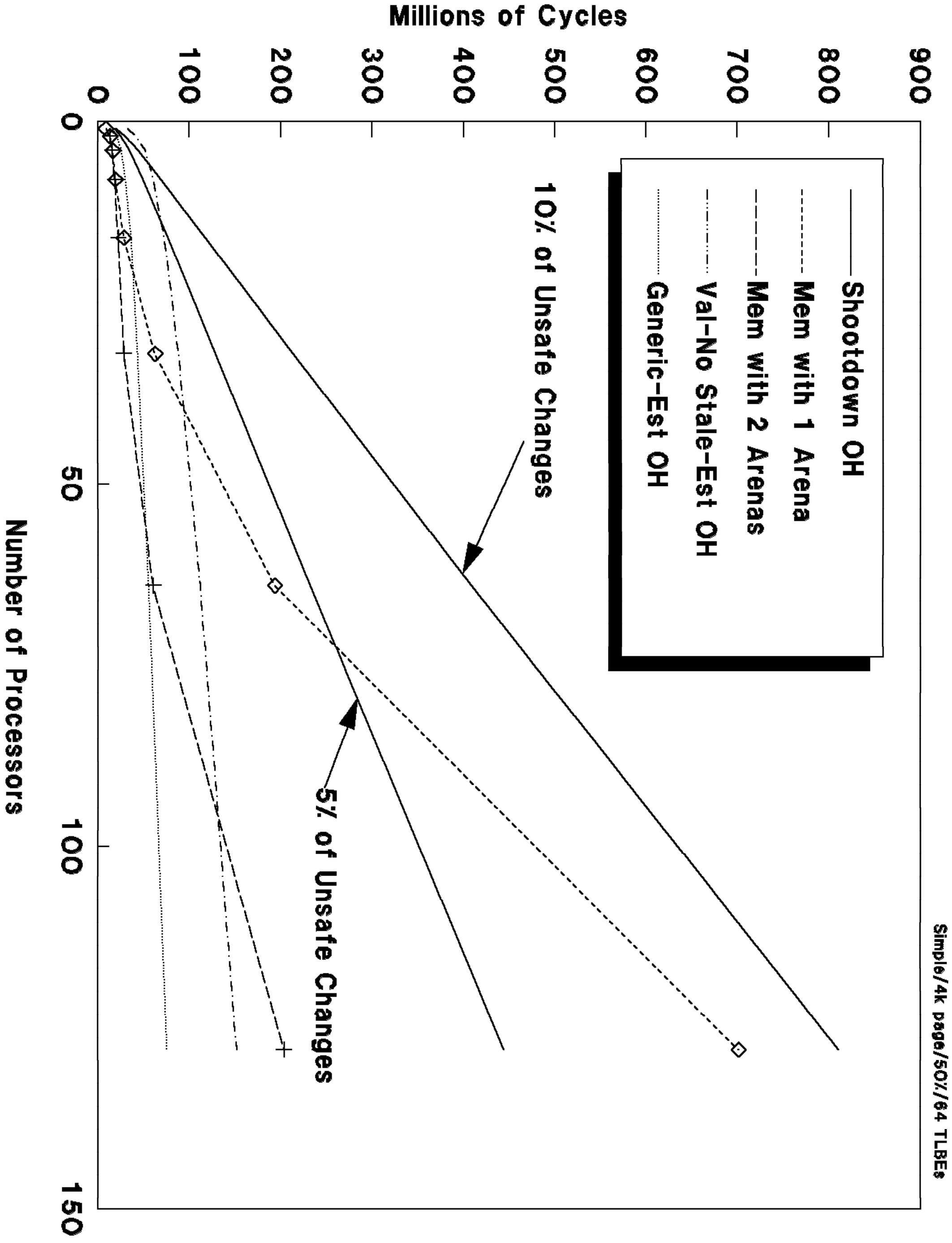
Graph 55. OH_{Shoot}, OH_{Val}, **and** OH_{Mem} **vs.** N **–Simple/33%/64 TLBEs**



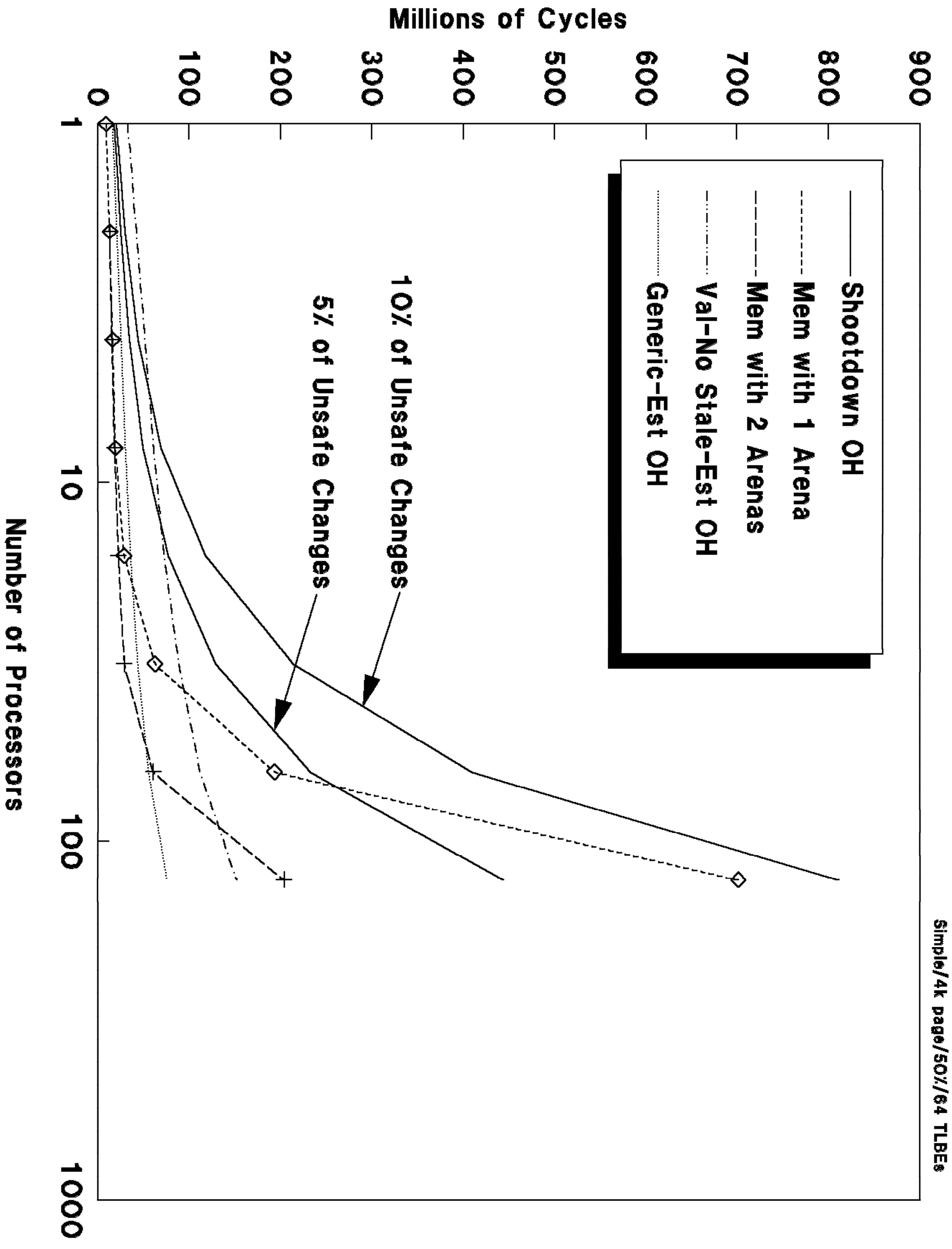
Graph 56. OH_{Shoot}, OH_{Val}, **and** OH_{Mem} **vs.** N **–Simple/33%/64 TLBEs**



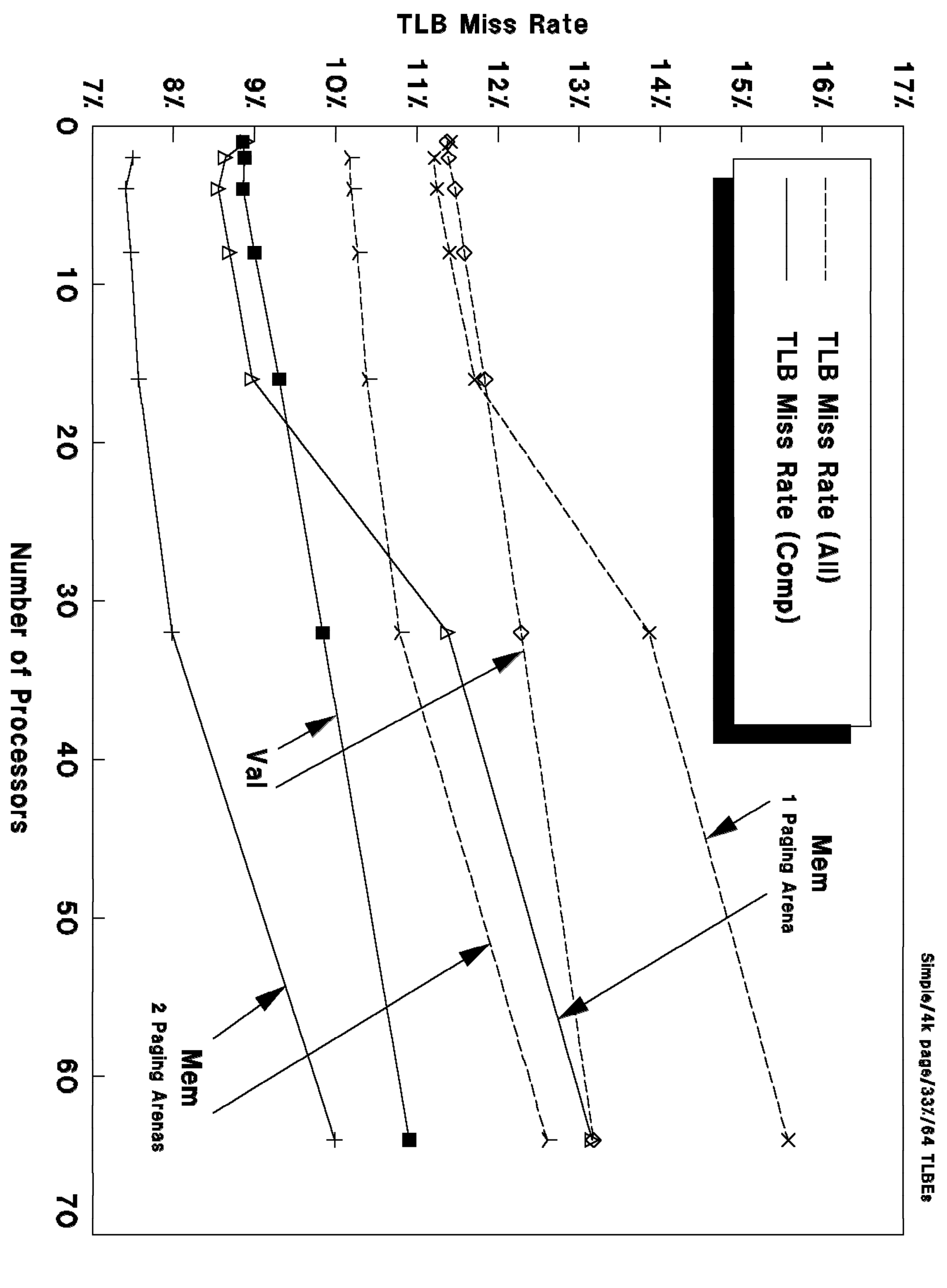
Graph 57. OH_{Shoot}, OH_{Val}, **and** OH_{Mem} **vs.** N **–Simple/50%/64 TLBEs**



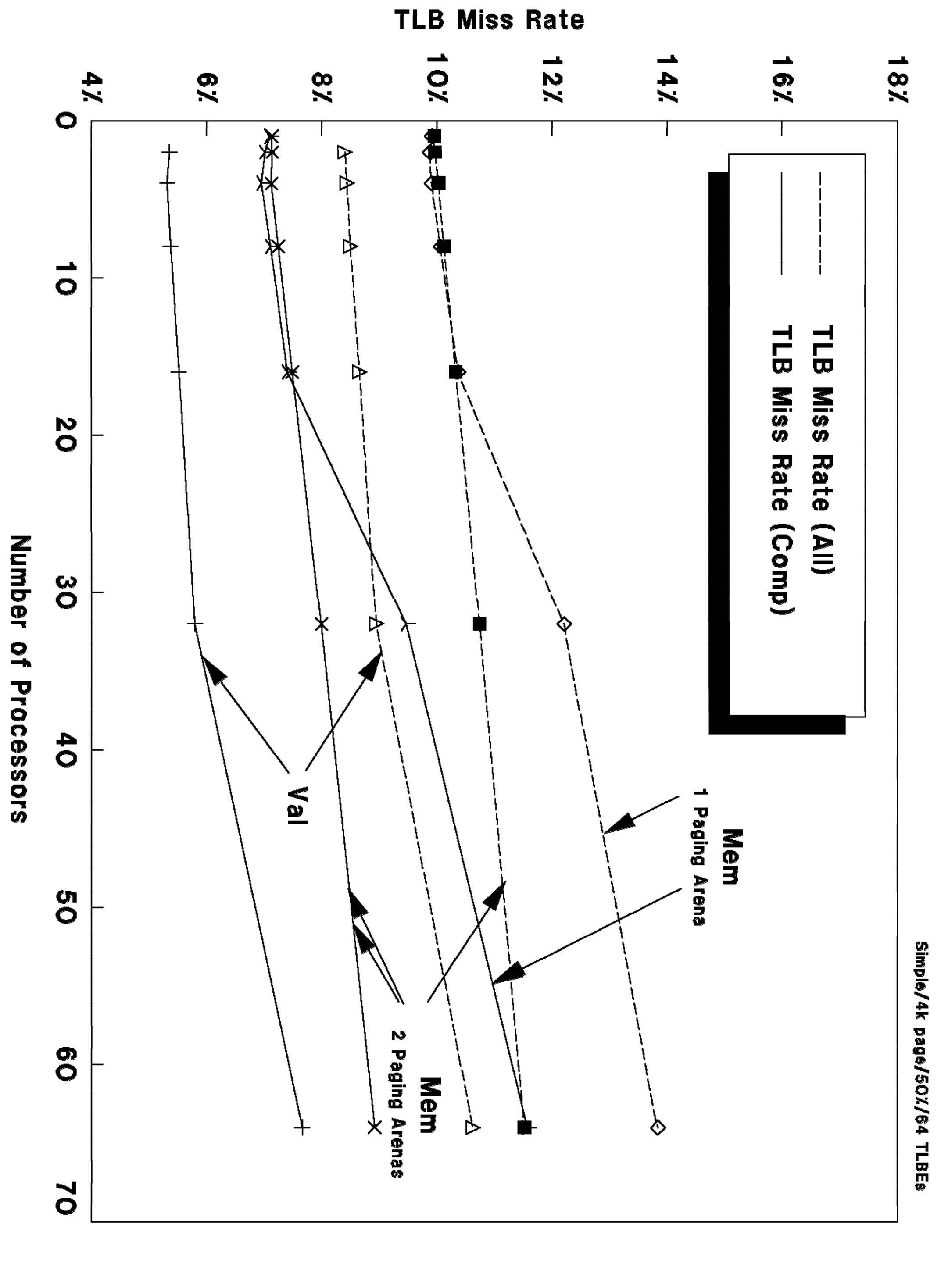
Graph 58. ***OH_{Shoot}*, *OH_{Val}*, and *OH_{Mem}* vs. *N* –Simple/50%/64 TLBEs**



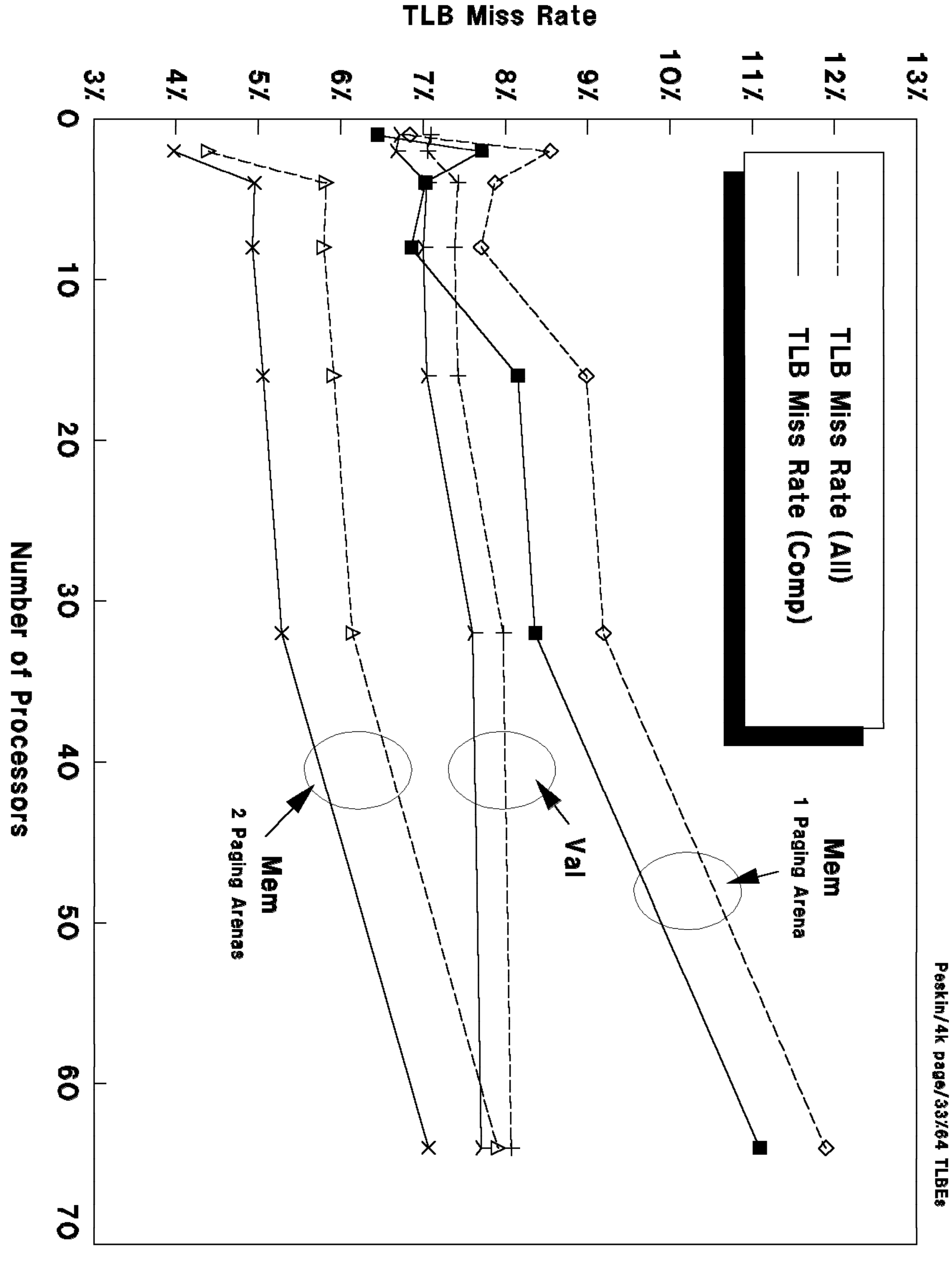
Graph 59. ***OH_{Shoot}*, *OH_{Val}*, and *OH_{Mem}* vs. *N* –Simple/50%/64 TLBEs**



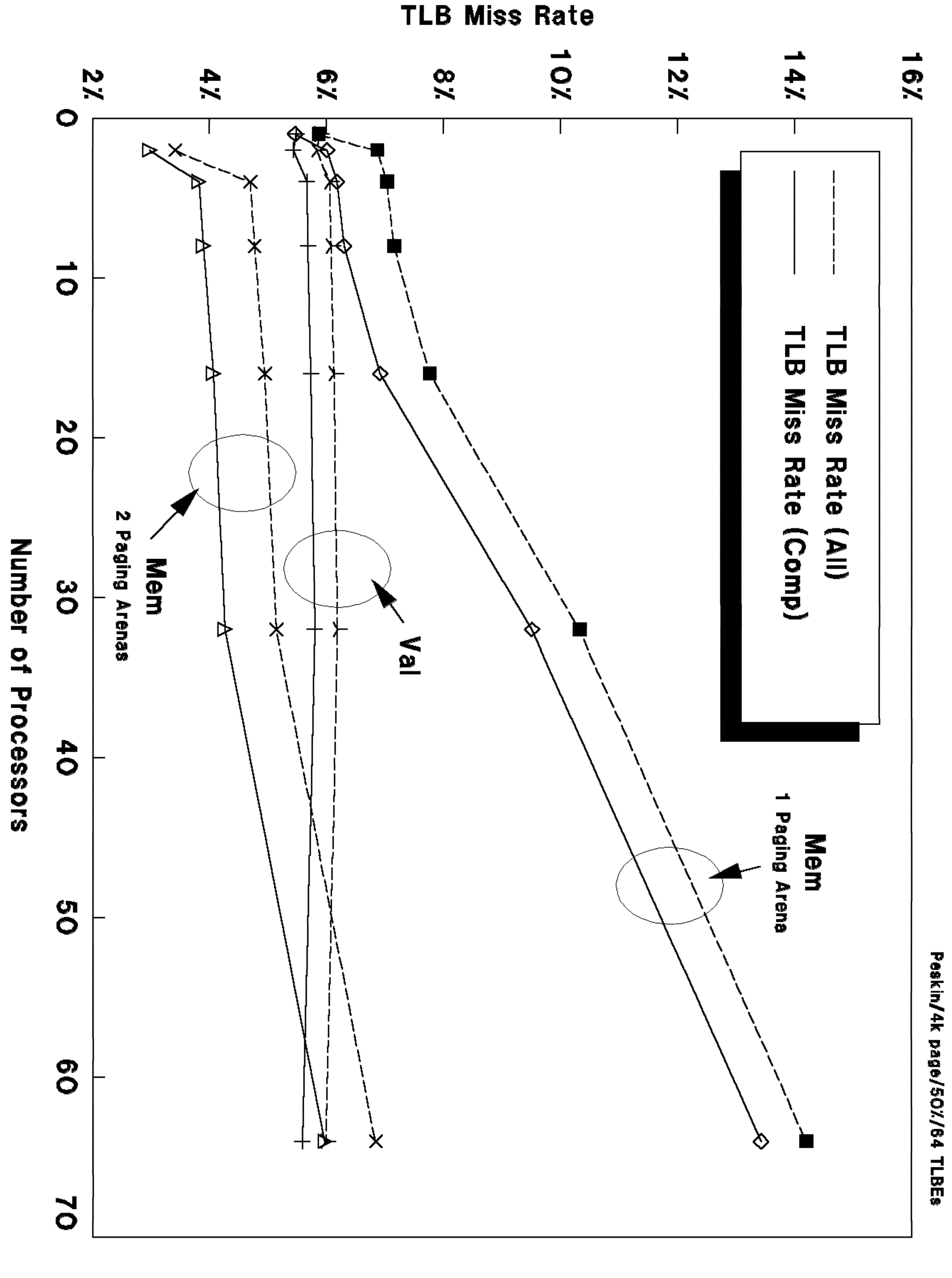
Graph 60. TLB Miss Rates vs. *N*–Simple/33%/64 TLBEs



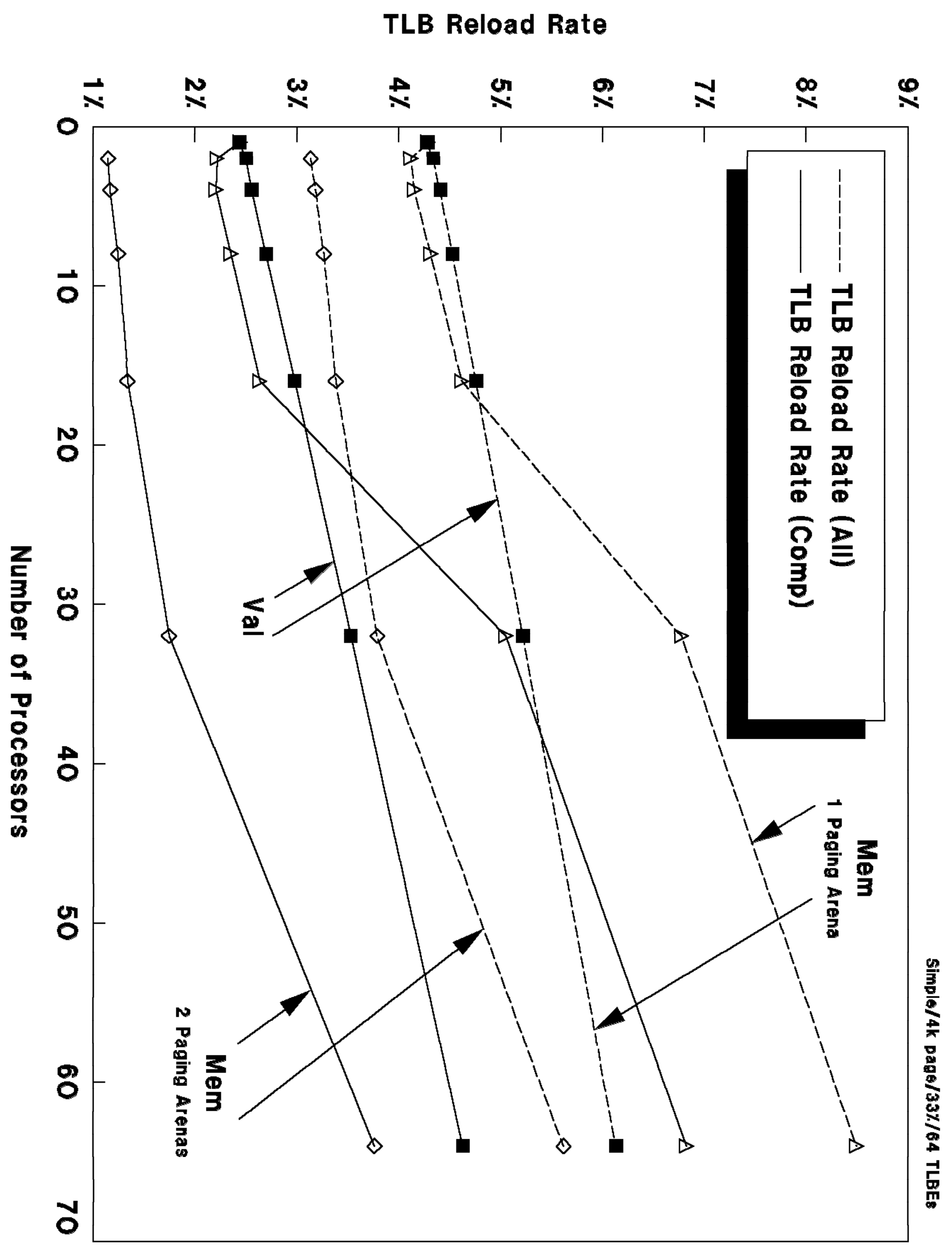
Graph 61. TLB Miss Rates vs. *N*–Simple/50%/64 TLBEs



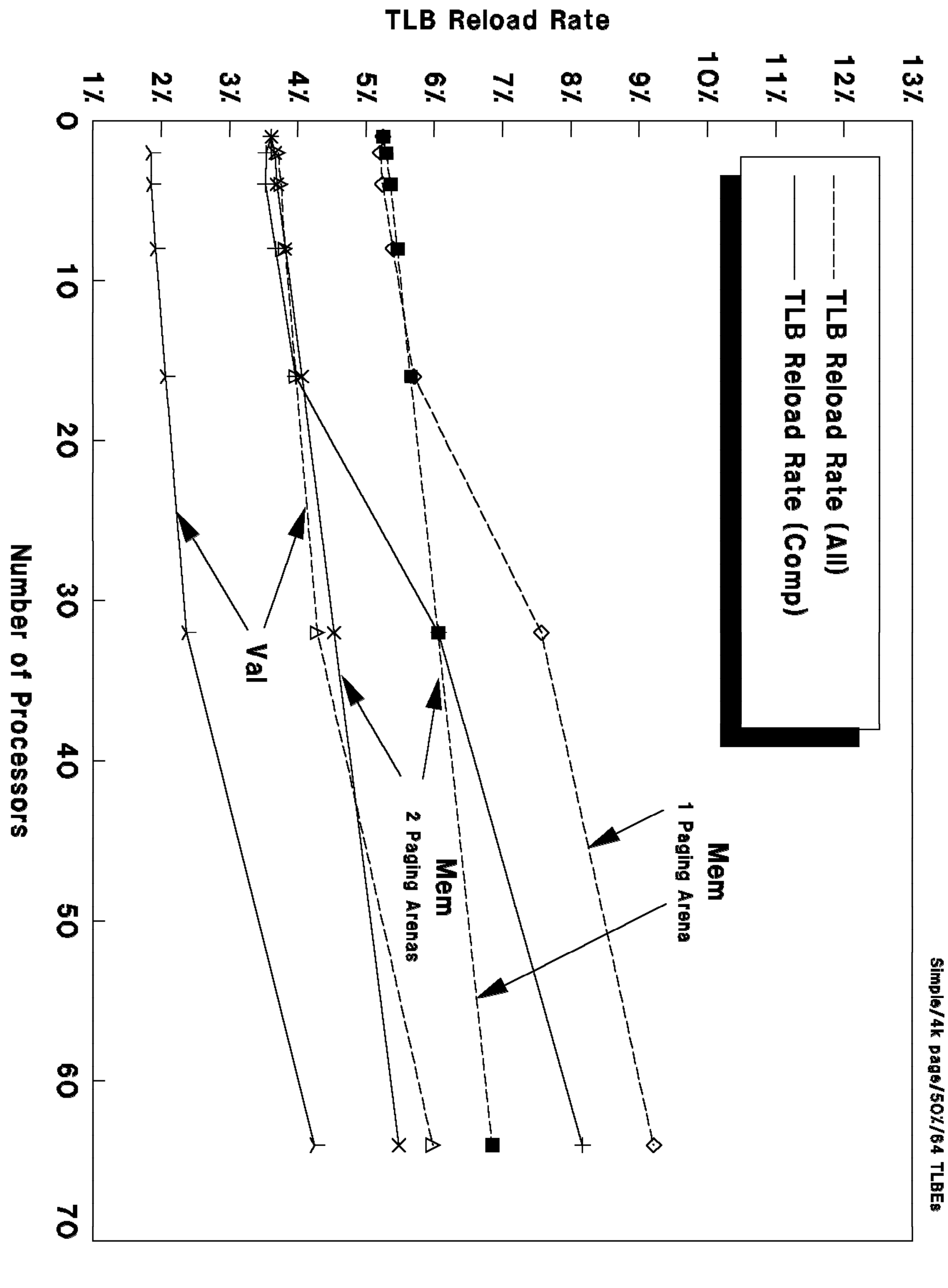
Graph 62. TLB Miss Rates vs. *N*–Peskin/33%/64 TLBEs



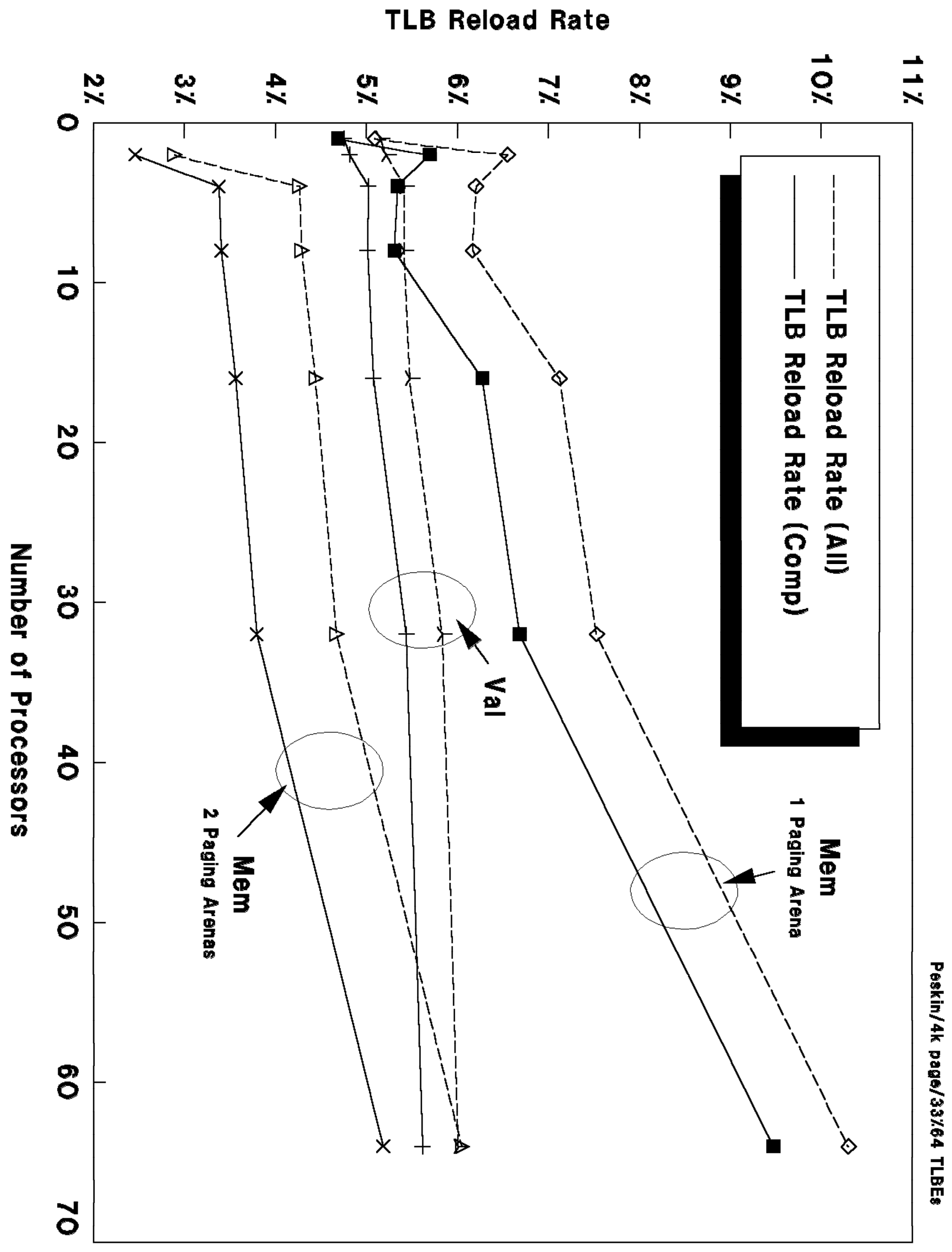
Graph 63. TLB Miss Rates vs. *N*–Peskin 50%/64 TLBEs/64 TLBEs



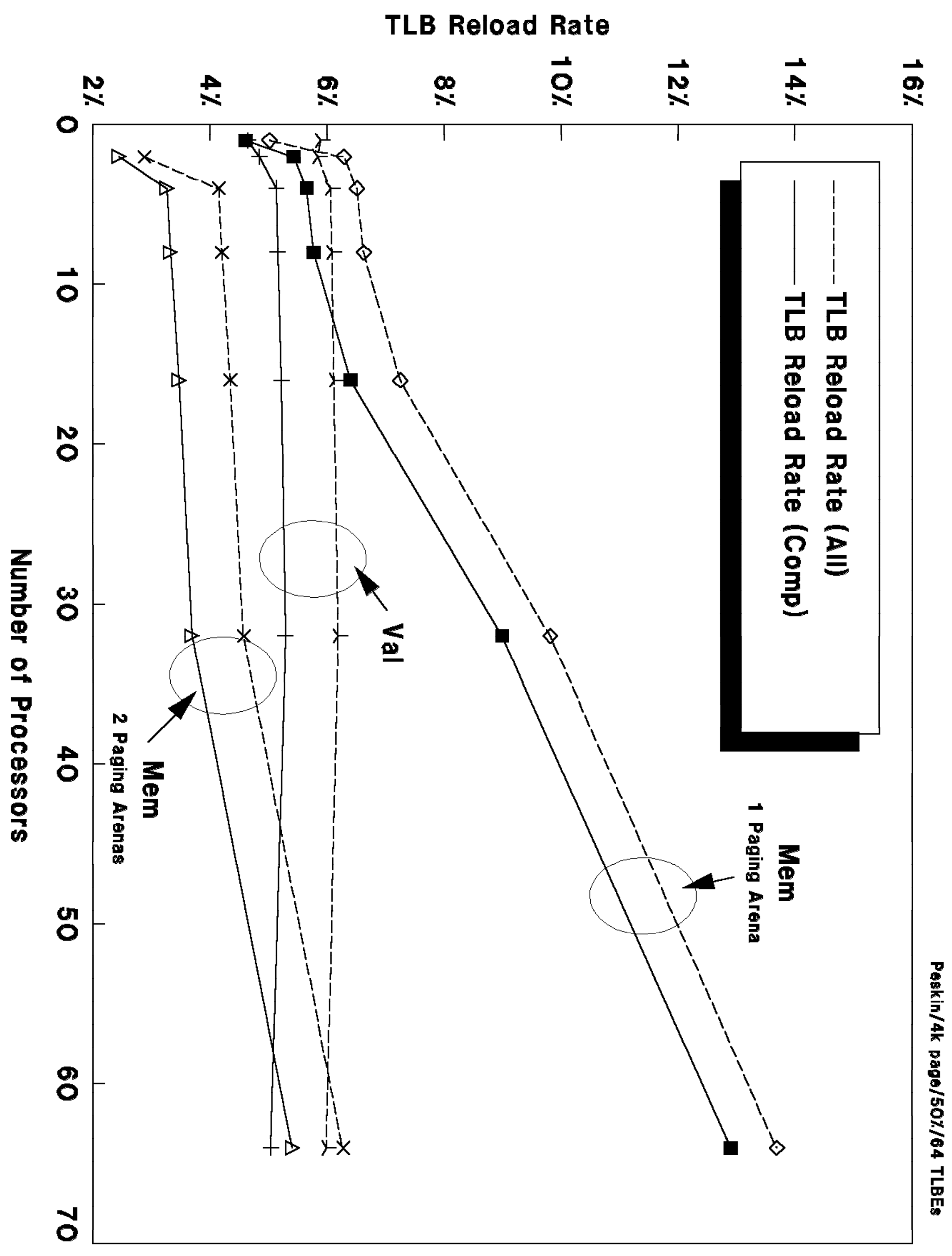
Graph 64. TLB Reload Rates vs. *N*–Simple/33%/64 TLBEs



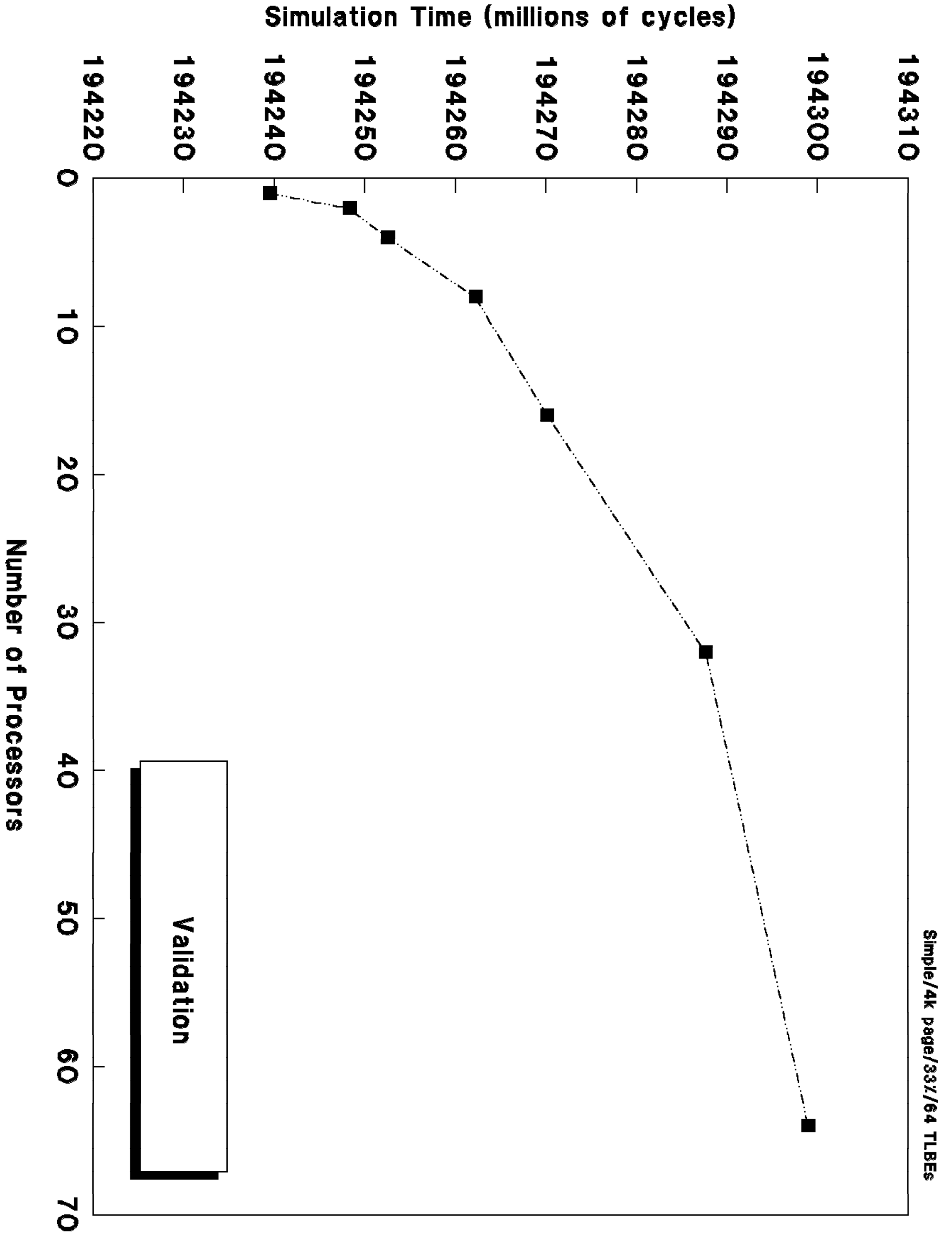
Graph 65. TLB Reload Rates vs. *N*–Simple/50%/64 TLBEs



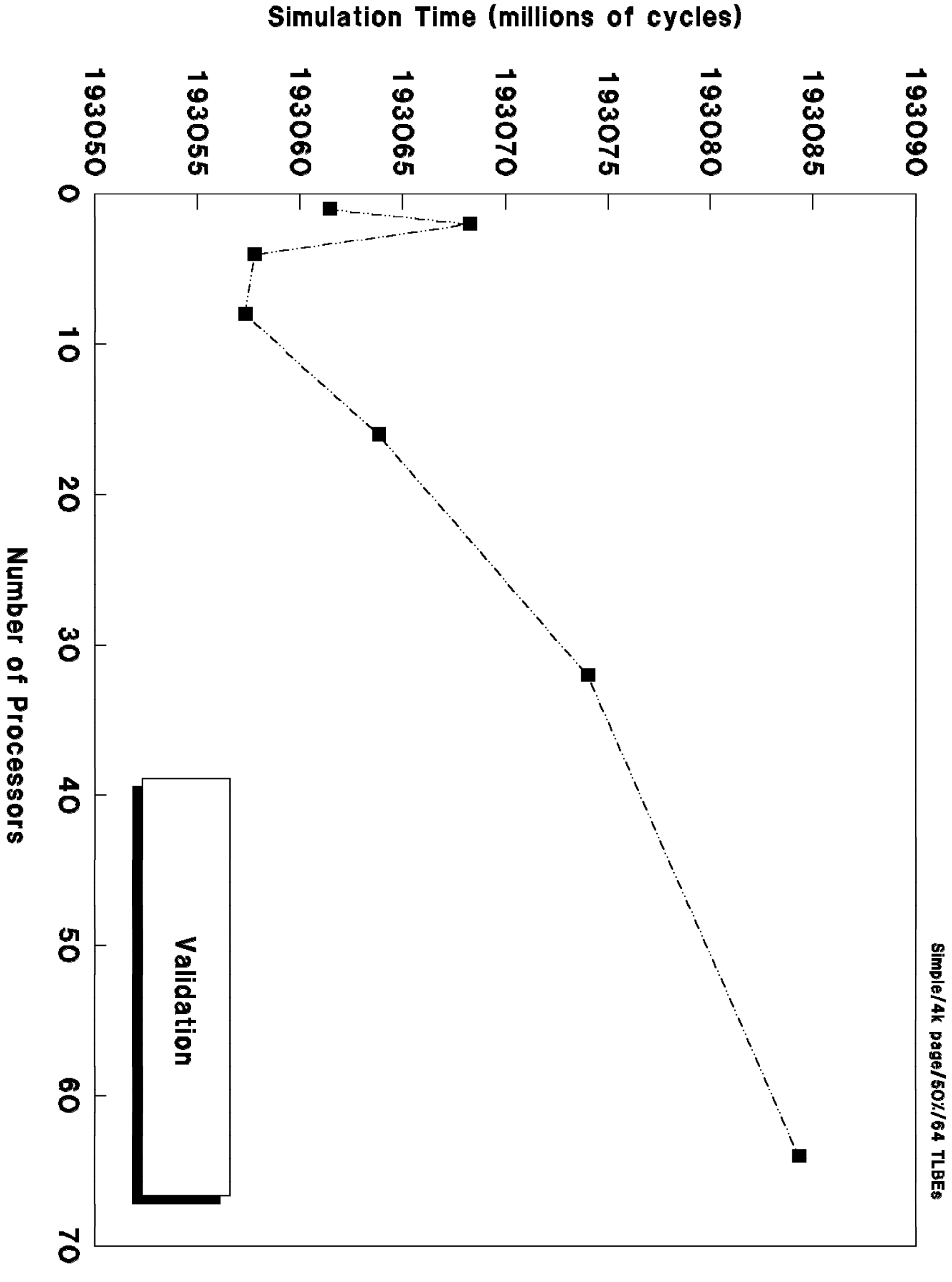
Graph 66. TLB Reload Rates vs. *N*–Peskin/33%/64 TLBEs



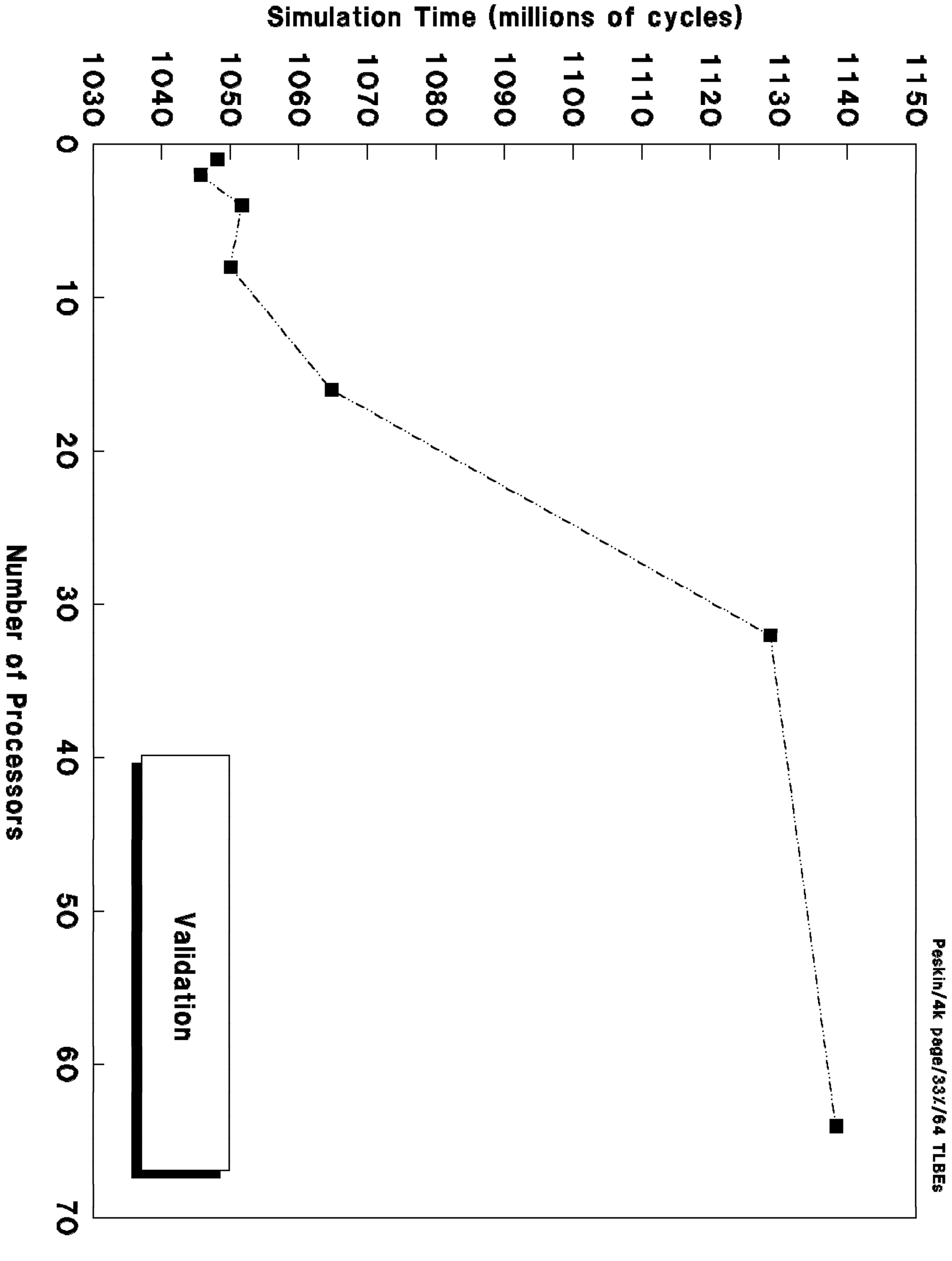
Graph 67. TLB Reload Rates vs. *N*–Peskin 50%/64 TLBEs



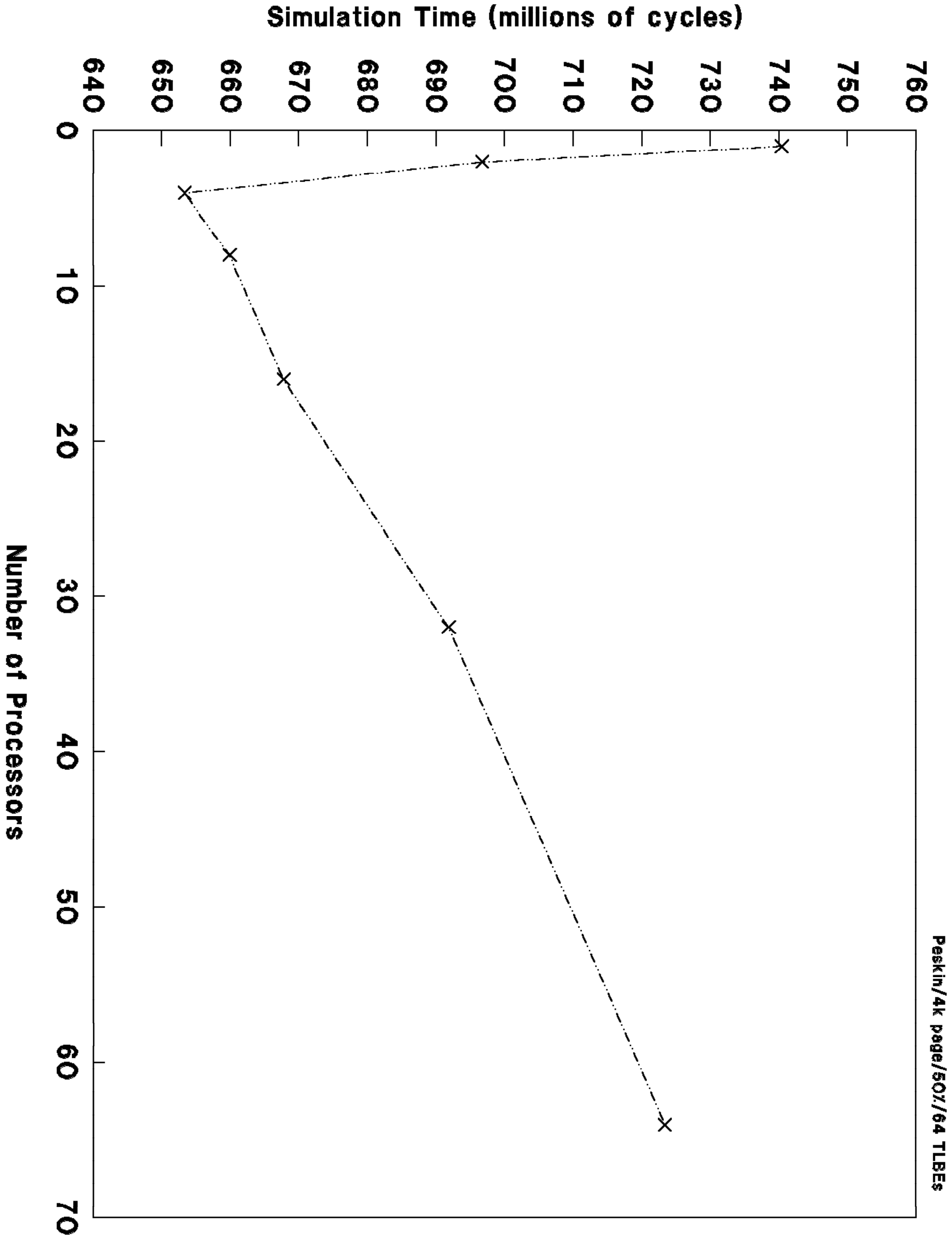
Graph 68. Simulation Time vs. *N*–Simple/33%/64 TLBEs



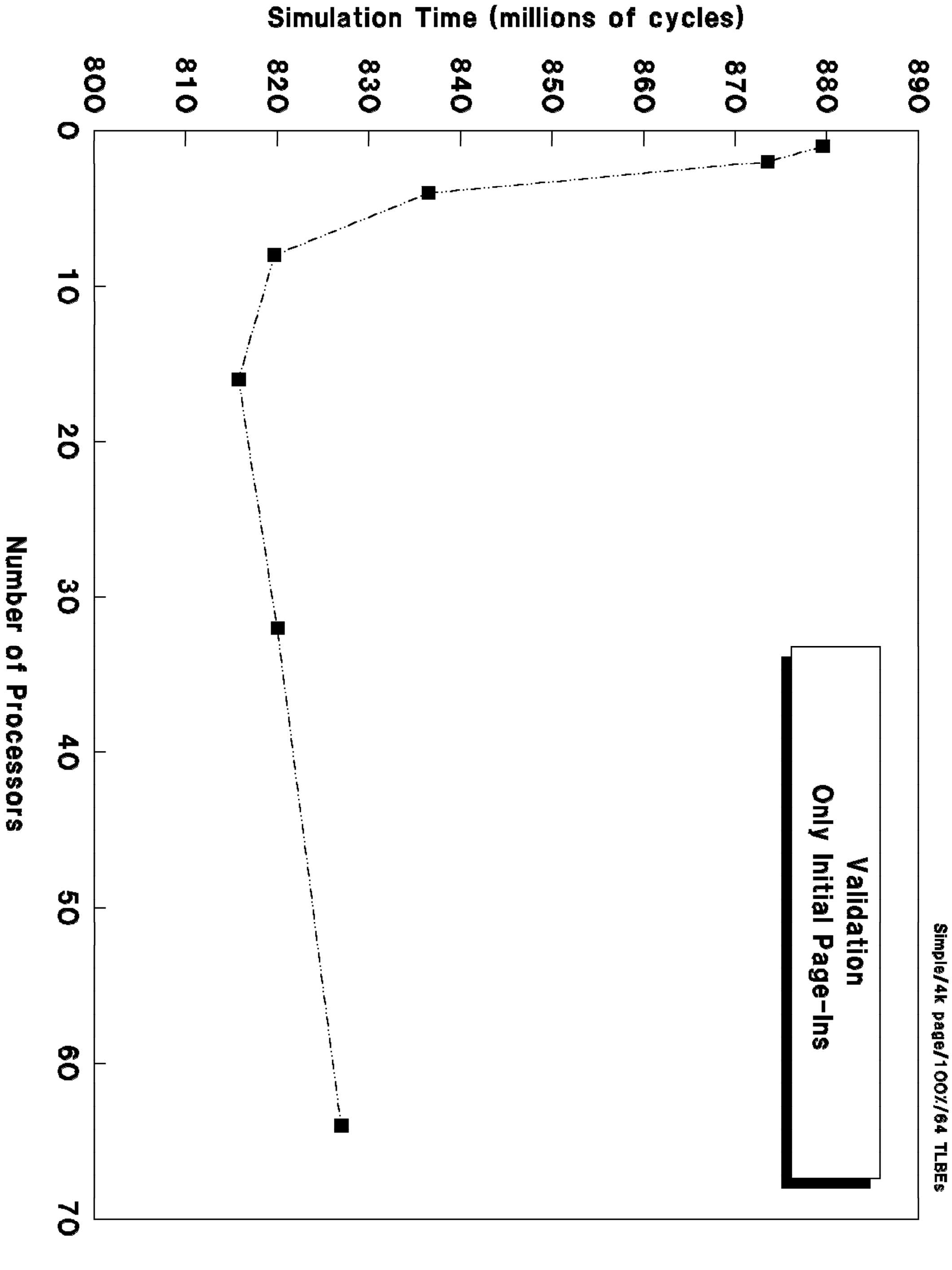
Graph 69. Simulation Time vs. *N*–Simple/50%/64 TLBEs



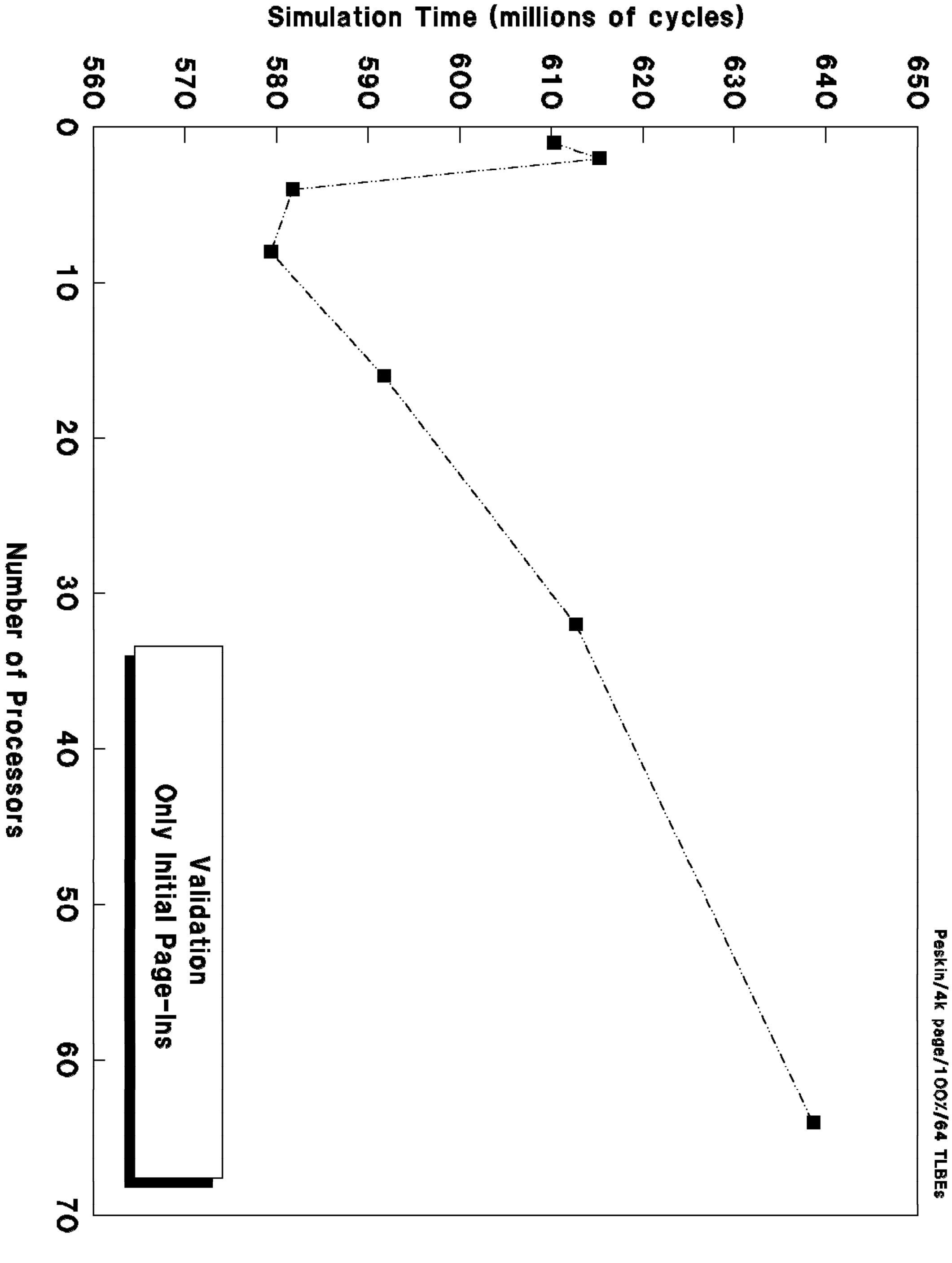
Graph 70. Simulation Time vs. *N*–Peskin/33%/64 TLBEs



Graph 71. Simulation Time vs. *N*–Peskin/50%/64 TLBEs



Graph 72. Simulation Time vs. *N*–Simple/100%64 TLBEs



Graph 73. Simulation Time vs. *N*–Peskin/100%/64 TLBEs

Part IV: Conclusions and Future Research

This part of the thesis includes:

- a discussion of the conclusions drawn from this research and
- an outline of research directions suggested by this thesis.

Chapter 9. Conclusions and Future Research

This chapter ends the thesis by presenting conclusions we have drawn from our research and discussing directions for possible future research that are suggested by this thesis.

9.1. Conclusions

Our research gives evidence that:

1. TLB consistency solutions proposed by other researchers fall into two classes. Those in the first class are suitable only for systems with efficient broadcast capability, *e.g.*, multiprocessors with a shared bus interconnecting processors and memory. Those in the second class also can be implemented in multiprocessors with a more general network interconnecting processors and memory. Further research is needed to quantify the efficiency of the resulting implementations.
2. In large-scale systems, our three new TLB-consistency solutions, which incorporate the cost of maintaining TLB consistency in the cost of TLB reloads, appear to have a smaller computational overhead than other solutions.
3. Program speedup and, thus, system performance is likely to be degraded when parallel programming constructs cause R, the number of TLB reloads, to grow linearly with N, the number of processors in the system. In this case, the TLB-reload overhead increases at least linearly with N. In systems with processor-based TLBs, the reload cost grows as $\log N$ and, thus, the TLB-reload overhead grows as $R \log N$.
4. System performance may improve if TLBs are moved from their conventional location in processors and are located at memory instead, provided that memory is organized in clusters, called paging arenas, the number of which grows with N. (Each page is mapped to a specific paging arena in this model.)
5. To understand the performance impact of algorithms in a scalable multiprocessor architecture, it is not sufficient to study performance in a single multiprocessor system, where the cost of network transit is relatively constant and the sizes of system components are fixed. In order to characterize performance in a scalable architecture, performance must be studied in a simulation system like ours, which increases the cost of network transit with the system size and allows the sizes of system components to be varied.

We briefly address each of these points below.

9.1.1. Other Solutions

Solutions of other researchers that can be implemented in the targeted multiprocessor architecture are based on the TLB Shootdown algorithm [Black, *et al.*, 1989]. TLB Shootdown is used in many existing multiprocessors. Part of the reason for this is that TLB Shootdown solves the most general formulation of the TLB consistency problem, requires no special hardware support, and has, thus far, performed adequately for the small-scale systems (*i.e.*, with small N) to which it has been ported. We do not believe that TLB Shootdown will be efficient in larger systems because it interrupts the execution of all processors that have access to the page table being modified, requires communication and synchronization among these processors, and may introduce serialization.

9.1.2. *Our Solutions*

As an alternative to TLB Shootdown, this thesis proposes three new solutions to the TLB consistency problem that are targeted for highly-parallel systems with multi-stage interconnection networks:

1. Validation,
2. Read-locked TLBs, and
3. Memory-based TLBs.

The first two assume TLBs located at the processors, and the third assumes TLBs located at memory. In order to compare these solutions with those based on TLB Shootdown, the next three subsections discuss the generality of our TLB consistency solutions, the hardware support they require, and their computational overhead.

9.1.2.1. Generality

We believe that future operating systems will provide very flexible use of shared memory. Consider the Mach operating system [Rashid, *et al.*, 1987] and the Symunix operating system [Edler, Lipkis, and Schonberg, 1988a and 1988b]. In addition, page-protection techniques are being used to implement concurrent garbage collection and concurrent checkpointing algorithms [Appel and Li, 1991]. Thus, the generality of a solution is important.

Validation, like TLB Shootdown, solves the most general formulation of the TLB consistency problem.

For systems with one paging arena, so does Memory-based TLBs. For systems with multiple paging arenas, Memory-based TLBs does not because the mapping of pages to sets of frames is restricted and, thus, pages sharing a frame are limited to the pages that map to the cluster in which the frame resides. These imposed restrictions on page sharing must be weighed against the reduction in TLB-reload overhead that may be realized by employing memory-based, rather than processor-based, TLBs. If the parallel programming environmemt is designed so that pages intended to map to the same frame are mapped to the same cluster, then our solution for memory-based TLBs is a viable one.

Read-locked TLBs also does not solve the most general formulation of the problem because in some cases page-table modifications may be postponed indefinitely. Therefore, for many systems it may not be a practical solution by itself. However, if used in concert with another solution it may reduce the overhead associated with ensuring TLB consistency.

9.1.2.2. Hardware Support

Since hardware costs are one-time costs and performance costs are continuously being incurred, we believe that the performance cost of a solution should be considered carefully before dismissing a solution because of its inherent hardware cost.

Read-locked TLBs, like the original TLB Shootdown algorithm, does not require any special hardware, however, our other two solutions do. For example, Validation requires that TLB entries have an extra field to store a generation count and each memory module have a Validation Table that can be accessed quickly. Memory-based TLBs substitutes a TLB located at each memory module for one at each processor and adds a bus that interconnects the TLBs and the page table associated with a cluster of memory modules.

9.1.2.3. Computational Overhead

The computational cost of our solutions is manifested in the cost of TLB reloads and, thus, increases with R, the number of TLB reloads. For processor-based TLBs, this overhead is $O(R \log N)$. If R increases linearly with N, then it is $O(N \log N)$. Even though R may be larger for memory-based TLBs, the TLB-reload cost is potentially much smaller because a TLB reload does not require network transit, which is $O(\log N)$. Accordingly, the overhead for memory-based TLBs potentially is smaller.

In contrast, the computational overhead of solutions based on TLB Shootdown is $O(\beta P \log N)$, where β is the number of page-table modifications that may cause TLB inconsistencies and P is the number of processors that share a page table. It is not known how β and P increase with N. If they grow linearly with N then the computational overhead of TLB Shootdown is $O(N^2 \log N)$.

9.1.3. R Increases with N

Whether the employed TLB-reload routine is consistency ensuring or not, R's growth may determine program speedup and, thus, system performance. This is because as N increases, the percentage of TLB reloads that must be executed by all N processors may increase. In a scalable shared-memory multiprocessor architecture, program speedup depends on the distribution of, among other things:

- the computational work performed by a one-processor system in order to execute the program,
- the number of TLB reloads generated during the program's execution, and
- the computational overhead associated with the employed TLB consistency solution.

Generally, the best speedup is achieved if the total work is distributed evenly among all N processors in a system.

In order to illustrate how R's growth may determine program speedup and effect system performance, consider a system that employs Validation. Assume that all work is evenly distributed among all N processors in the system and that $R = aN + b$. In this case, an even distribution of the R TLB reloads means that each processor generates $a + (b/N)$ TLB reloads. Thus, a determines program speedup: The percentage of TLB reloads that must be executed by each processor, $a/(a + (b/N))$, increases with N. In this thesis we show that certain parallel programming constructs can cause R to grow linearly with N, therefore, this example is not hypothetical. However, if N is restrained so that $b/N \neg a$, then program speedup is not significantly effected by TLB-reload overhead.

9.1.4. Memory-based TLBs vs. Processor-based TLBs

Comparing the computational overhead of our two solutions for processor-based TLBs, our one solution for memory-based TLBs, and TLB Shootdown, we find that memory-based TLBs outperform processor-based TLBs if the number of paging arenas increases with N. Although the location of TLBs at memory is very unconventional, these results may persuade architects to consider this alternative.

General-purpose caches managed by a write-back policy introduce problems for a system that employs Validation or Memory-based TLBs. In a Validation system, a write-back of a cache entry is accompanied by a generation count. In this case, a virtual-address cache does not pose a problem since the virtual page number is stored in the cache and translation information is required to perform a write whether the cache is managed by a write-back or write-through policy. A conventional physical-address cache poses a problem because a write-back must be accompanied by a generation count. Generation counts can be stored in cache entries so that they can be transmitted with write-backs, but what if a cached count is no longer associated with the referenced frame? Then the referenced virtual page number is needed. Thus, there either must be a history of the virtual page numbers and generation counts associated with a frame or, like Goodman's Multi-tagged Caches [1987], virtual-address tags can be stored in cache entries. In this case, if generation counts are not also stored in cache entries, then translation information may have to be accessed in order to perform write backs. In addition, it must be ensured that a generation count update does not associate with the frame a count that is stored in associated cache entries. This can be ensured if generation counts updates are performed by unit increments and when wrap-around occurs associated entries are flushed from TLBs and caches. To avoid the problems associated with write-back physical-address caches in a Validation system, if consistency-ensuring TLB management is only necessary for shared data, caches could be partitioned so that cached private data is managed by a write-back policy and shared data is managed by a write-through policy.

In a memory-based TLB system with write-back virtual-address general-purpose caches, illegal in-cache writes to a page may be executed. This can occur when a page's protection is increased, *e.g.*, when its status has changed from read-write to read-only. The associated protection violation is discovered only when the entry is written back to memory. In this case, on a protection increase the cache-consistency solution can be designed to force the write-back and subsequent invalidation of associated cache entries. If consistency-ensuring TLB management is only necessary for shared data, then in a memory-based TLB system, memory-based TLBs can be employed for only shared data, while processor-based TLBs are employed for private data. In addition, the caches for shared data can have a write-through policy, while the caches for private data have a write-back policy.

9.1.5. Performance Analysis

The performance of general-purpose caches and TLBs becomes more important as the system size increases. This is because the cost of network transit increases with system size and the performance of a scalable multiprocessor is dependent on the amount of network traffic that is generated.

Performance studies, such as those of Black, *et al.* [1989] and Rosenburg [1989], which study the performance of their TLB consistency solutions in real systems, have their limitations. They are adequate for studying performance in the particular systems used, but their results cannot be used with confidence to predict performance in larger systems in which the cost of network transit increases. In addition, these methods cannot predict the effect of architectural changes.

In contrast, our simulation methodology allows us to study performance of multiprocessors spanning a range of architectures and sizes.

9.2. Future Research

Our simulation methodology has proved to be an effective way to evaluate the performance of methods for ensuring TLB consistency and examine the behavior of demand paging in scalable multiprocessor systems. Since we now have the simulation system in place, we intend to study other parallel workloads in order to increase the validity of our results. We are already in the process of simulating a weather forecasting benchmark that was provided to New York University's Courant Institute of Mathematical Sciences by the NASA Space

Flight Center, Greenbelt, Maryland (*cf.*, [Kalnay-Rivas and Hoitsma, 1979; Dorn and Rushfield, 1981]).

We believe that this simulation methodology will be useful in studying related issues as well, some of which are discussed below. In addition, the following subsections discuss other directions for future research suggested by this thesis.

9.2.1. TLB-reload Overhead

Our simulation results show that the number of TLB reloads increase with the number of processors cooperating in the execution of a program. In systems with processor-based TLBs this is due to page sharing among processors and changes in TLB context. In systems with memory-based TLBs this is due to replication of translation information at multiple TLBs and TLB-entry thrashing.

It would be interesting to study the change in the number of TLB reloads that occur during a program's execution as the page size is varied and when multiprogramming is supported. As the page size increases, the amount of page sharing among cooperating processors is likely to increase. For processor-based TLBs, an increase in page sharing is likely to increase the number of reloads associated with sharing processors. However, a larger page size probably will decrease the number of TLB reloads per processor due to the greater region of memory covered by the (constant size) TLB. For memory-based TLBs, an increase in page sharing is likely to decrease the number of reloads associated with pages shared among processors.

It also would be interesting to see what affect the scheduling of multiple tasks of a parallel section of code to one processor, *i.e.*, *chunking*, would have on the number of TLB reloads generated during program execution and, thus, a solution's TLB-reload overhead.

We found that context switching actually reduced the number of generated processor-based TLB reloads, in comparison to not context switching. We believe that this is because the granularity of tasks are reduced so that tasks in the same concurrent task group are able to reuse TLB-resident translation information.

A related area to investigate is whether the scheduling of tasks, in general, could be guided in order to minimize the number of TLB reloads generated. This would include investigating affinity scheduling, where processes do not migrate among processors. The issues we refer to are related to optimizing data locality to increase cache performance [Wolfe, 1989].

Our simulation study held the problem size, p, fixed as the number of processors, N, and, thus, the size of the interconnection network, was increased. It would be interesting to conduct a similar study increasing p with N. This would require substantially more (and larger) program traces. In addition to increasing p with N, we would like to measure the growth of R and the related TLB-reload overhead with respect to all user and system pages.

Along the same lines, we would like to investigate ways of generating traces for larger size problems from the trace of a fixed-size problem.

9.2.2. TLB Shootdown

In comparing the computational cost of our solutions to that of TLB Shootdown, we used data from our simulations to estimate β, the number of page-table modifications that may cause TLB inconsistencies. As explained in Chapter 8, this estimate includes modifications of entries for only user-shared pages. We would like to measure β in a real system, where modifications are made to entries for all user and system pages. In addition, we would like to measure β as the problem size increases.

In estimating the computational cost of TLB Shootdown for the programs used in our study, we assumed $P = N$, *i.e.*, all N processors in the system participate in the algorithm. We would like to measure P in terms of user and system page tables, and as N and the problem size increase.

Our solutions assume that in order to modify a page-table entry, it must first be locked. In contrast, TLB Shootdown assumes that the entire page table be locked. As a result, TLB Shootdown restricts parallelism in a manner that our solutions do not. Employing TLB Shootdown, modifications to a page table are serialized and modifications to page tables that are shared by the same processor(s) are serialized. We would like to measure the frequency that serialization is introduced.

9.2.3. Paging Activity

The programs used in our study proved to be poor candidates for a demand paging environment. It would be enlightening to study the performance of other parallel programs. Demand paging may impair the performance of a parallel program run on multiple processors, even when its serial counterpart run on a uniprocessor performs well in a demand paging environment. This can occur when all processors page fault at the same time, either on the same shared (critical) page or on different (critical) pages. Such behavior can interrupt program execution when there are no other tasks to execute. (Consider a serial task that page faults). If this proves to be a problem, then it may help performance to determine potentially critical pages, and prepage them into the system. With such information, translation information for such pages could be broadcast to all or a subset of the processors. This might reduce the number of common TLB misses such that the total number of misses would not increase as $O(N)$.

When one page is referenced by many concurrent tasks, the translation information for that page can be thought of as a *hot spot* [Pfister and Norton, 1985] and the related page can be thought of as a *hot page*. The fully productive page-ins generated by the Simple trace, described in Section 8.3.1.1, identify such pages. Our preprocessor removes all EPEX-generated code and our traces do not include operating systems references. If such references were included, we expect that the number of fully productive page-ins would increase. Using full traces of parallel programs, a simulation study could better determine the effect of such hot-spots and hot-pages.

An optimal page-replacement algorithm for a multiprocessor system has not yet been defined. This is certainly a worthwhile project. As suggested by Read-locked TLBs, it may be reasonable to lock in memory pages that have their translation information resident in TLBs. Also, if we consider paging both private and shared pages, it is an interesting question as to whether private pages or shared pages should have preference for memory residence. We have begun work on this topic and are considering replacement algorithms that take into consideration the number of processes that reference a page.

The need for serialization-free algorithms to implement demand paging should be evident. We are in the process of designing such algorithms based on event-processing techniques discussed in [Wilson, 1988] and the use of fetch-and-ϕ primitives. Our simulator is designed so that we can easily evaluate the performance of alternative algorithms.

9.2.4. Miscellaneous

The simulation study used in this thesis to evaluate TLB-reload overhead is for multi-processor with a fixed architecture, except for the number of processors and memory modules, the TLB size, the page size, and the number of paging arenas for systems with memory-based TLBs. We would like to vary some of the architectural features of the studied multiprocessor and evaluate their effect on performance. For example, as the processor speed increases, critical pages will have a more significant effect on performance.

TLB-entry thrashing may impair the performance of multiprocessor architectures like MIT's Alewife, which plans to employ a multithreaded processor called APRIL [Agarwal, *et al.*, 1990]. APRIL permits fast context switches on cache misses, in order to compensate for the latency inherent in satisfying a cache miss. Instead of one process accessing a TLB at a processor during a certain time interval, analogous to memory-based TLBs, several processes will access a TLB during the same time interval. This is likely to stress the TLB and increase the TLB miss rate. The difference in the situations is that in the case of memory-based TLBs the number of processes that may access a TLB is equal to N, whereas the number of processes in a processor like April will be limited to a small number. In any case, it would be interesting to investigate what effect the April approach has on the TLB miss rate.

This thesis has helped quantify the belief that TLB consistency does not come for free. Perhaps this will encourage designers of operating systems to carefully consider what unsafe changes are really necessary. It will be interesting to see what happens in the future.

In any case, we feel that our simulation results indicate interesting and important trends that may be helpful in designing consistency-ensuring TLB management for highly-parallel multiprocessor systems. The solutions that we proposed may prove to be efficient solutions in multiprocessor systems, or may provide a basis on which to design other solutions to the TLB consistency problem.

Bibliography

Agarwal, A., and M. Cherian, “Adaptive Backoff Synchronization Techniques,” *Proceedings of the 16th International Symposium on Computer Architecture*, IEEE Catalog No. 89CH2705-2, pp. 396-406, May/June 1989.

Adve, S. and M. Hill, “Weak ordering – a new definition,” *Proceedings of the 17th International Symposium on Computer Architecture*, IEEE Catalog No. 90CH2877-8, pp. 2-14, May 1990.

Agarwal, A., and A. Gupta, “Memory-reference characteristics of multiprocessor applications under MACH,” *Proceedings of the 1988 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 215-225, May 1988.

Agarwal, A., B-H. Lim, D. Kranz, and J. Kubiatowicz, “APRIL: a processor architecture for multiprocessing,” *Proceedings of the 17th International Symposium on Computer Architecture*, IEEE Catalog No. 90CH2877-8, pp. 104-114, May 1990.

Agarwal, A., R. Simoni, J. Hennessy, and M. Horowitz, “An evaluation of directory schemes for cache coherence,” *Proceedings of the 15th International Symposium on Computer Architecture*, IEEE Catalog No. 88CH2545-2, pp. 280-289, June 1988.

Alexander, C., W. Keshlear, F. Cooper, and F. Briggs, “Cache memory performance in a UNIX environment,” *Computer Architecture News*, **14**, 3, pp. 41-70, June 1986.

Almasi, G., and A. Gottlieb, *Highly Parallel Processing*, Redwood City, CA: The Benjamin/Cummings Publishing Co., Inc., 1989.

AMD, *Am29000 Streamlined Instruction Processor User's Manual*, 1987.

Anderson, D., S-Y. Tzou, G. Graham, “The DASH virtual memory system,” Technical Report No. UCB/CSD 88/461, U. C. Berkeley, Computer Science Division, November 1988.

Appel, A., and K. Li., “Virtual memory primitives for user programs,” *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Order No. 556910, pp. 96-107, April 1991.

Archibald, J., and J-L. Baer, “Cache coherence protocols: evaluation using a multiprocessor simulation model,” *ACM Transactions on Computer Systems*, **4**, 4, pp. 273-298, November 1986.

Archibald, J., and J-L. Baer, “An economical solution to the cache coherence problem,” *Proceedings of the 11th International Symposium on Computer Architecture*, IEEE Catalog No. 84CH2051-1, pp. 355-362, June 1984.

Axelrod, T., P. Dubois, and P. Eltgroth, “A simulator for MIMD performance prediction–application to the S-1 MkIIa multiprocessor,” *Proceedings of the 1983 International Conference on Parallel Processing*, IEEE Catalog No. 83CH1922-4, pp. 350-358, August 1983.

Bach, M., *The Design of the UNIX Operating System*, Englewood Cliffs, NJ: Prentice-Hall, 1986.

Baer, J-L., *Computer Systems Architecture*, Rockville, MD: Computer Science Press, Inc., 1980.

BBN, *Butterfly Parallel Processor Overview*, Cambridge, MA, June 1985.

BBN, *Inside the Butterfly GP1000*, Cambridge, MA, October 1988.

BBN, *Inside the BBN TC2000*, Cambridge, MA, February 1990.

Beck, B., Private communication, February 1988 and July 1990.

Beck, B., and D. Olien, "A parallel programming process model", *Proceedings Winter USENIX Conference '87*, pp. 83-102, January 1987.

Belady, L., "A study of replacement algorithms for a virtual storage computer," *IBM Systems Journal*, **5**, 2, pp. 78-101, 1966.

Bensoussan, A., C. Clingen, and R. Daley, "The Multics virtual memory: concepts and design," *Communications of the ACM*, **15**, 5, pp. 308-318, May 1972.

Berke, W., "ParFOR–a structured environment for parallel FORTRAN," Ultracomputer Note No. 137, N.Y.U., Courant Institute, April 1988.

Black, D., R. Rashid, D. Golub, C. Hill, and R. Baron, "Translation lookaside buffer consistency: a software approach," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE Catalog No. 89CH2710-2, pp. 113-122, April 1989.

Brantley, W., K. McAuliffe, and A. Weiss, "RP3 processor-memory element," *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE Catalog No. 85CH2140-2, pp. 782-789, August 1985.

Brorsson, M. "A distributed virtual memory scheme evaluated on an emulated MIMD multiprocessor," Technical Report, Department of Computer Engineering, University of Lund, Sweden, April 1988.

Censier, L., and P. Feautrier, "A new solution to coherency problems in multicache systems," *IEEE Transactions on Computers*, **C-25**, 12, pp. 1112-1118, December 1978.

Chaiken, D., C. Fields, K. Kurihara, and A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors," *Computer*, **23**, 6, pp. 49-58, June 1990.

Cheong, H., and A. Veidenbaum, "Compiler-directed cache management in multiprocessors," *Computer*, **23**, 6, pp. 39-47, June 1990.

Cheong, H., and A. Veidenbaum "A cache coherence scheme with fast selective invalidation," *Proceedings of the 15th International Symposium on Computer Architecture*, IEEE Catalog No. 88CH2545-2, pp. 299-307, May 1988.

Cheriton, D., A. Gupta, P. Boyle, and H. Goosen, "The VMP multiprocessor: initial experience, refinements and performance evaluation," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 88CH2545-2, pp. 410-421, May 1988.

Cheriton, D., G. Slavenburg, and P. Boyle, "Software-controlled caches in the VMP multiprocessor," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 86CH2291-3, pp. 366-374, June 1986.

Chow, P., and M. Horowitz, "Architectural tradeoffs in the design of MIPS-X," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 87CH2420-8, pp. 300-308, June 1987.

Clark, D., and J. Emer, "Performance of the VAX-11/780 translation buffer: simulation and measurement," *ACM Transactions on Computer Systems*, **3**, 1, pp. 31-62, February 1985.

Colwell, R., C. Hitchcock III, E. Jensen, H. Brinkley Sprunt, and C. Kollar, "Instruction sets and beyond: computers, complexity, and controversy," *Computer*, **18**, 9, pp. 8-19, September 1985.

Control Data Corporation, *CDC Cyber 200 Model 205 Computer System Hardware Reference Manual*, Revision 01, A, 1980, 1981.

Crowley, W., C. Hendrickson, and T. Rudy, "The Simple code," Lawrence Livermore Laboratory, Livermore, CA, February 1978.

Cytron, R., S. Karlovsky, and K. McAuliffe, "Automatic management of programmable caches," *Proceedings of the 1988 International Conference on Parallel Processing*, IEEE Catalog No. 88CH2625-2, pp. 229-238, August 1988.

Darema, F., A. Karp, and P. Teller (Contributors), "Applications survey reports–I," IBM Research Report No. RC12743, Edited by F. Darema, May 1987.

Darema-Rogers, F., G. Pfister, and K. So, "Memory access patterns of parallel scientific programs," *Proceedings of the 1987 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 46-58, May 1987.

Darema-Rogers, F., D. George, V. Norton, G. Pfister, "A single-program-multiple-data computational model for EPEX/FORTRAN," IBM Research Report No. RC11552, November 1986.

Darema-Rogers, F., D. George, V. Norton, and G. Pfister, "VM/EPEX–a VM environment for parallel execution," IBM Research Report No. RC11225, January 1985.

Davidson, J., and R. Vaughan, "The effect of instruction set complexity on program size and memory performance," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE Catalog No. 87CH2440-6, pp. 60-64, October 1987.

Denning, P., "On modeling program behavior," *Proceedings of the Spring Joint Computer Conference*, pp. 937-944, May 1972.

Denning, P., "Virtual memory," *Computing Surveys*, **2**, 3, pp. 153-189, September 1970.

Dickey, S., A. Gottlieb, R. Kenner, and Y-S. Liu, “Designing VLSI network nodes to reduce traffic in a shared memory parallel computer,” *Circuits, Systems, and Signal Processing*, **6**, 2, pp. 217-238, 1987.

Dickey, S., R. Kenner, M. Snir, and J. Solworth, “A VLSI combining network for the NYU Ultracomputer,” *Proceedings of the IEEE International Conference on Computer Design*, IEEE Catalog No. 85CH2223-6, pp. 110-113, October 1985.

Digital Equipment Corporation, *VAX-11/780 Architecture Handbook*, 1977.

Dubois, M., D. Scheurich, and F. Briggs, “Synchronization, coherence, and event ordering in multiprocesors,” *Computer*, **21**, 2, pp. 9-21, February 1988.

Edler, J., “NYU Ultracomputer UNIX programmer's manual,” Ultracomputer Documentation Note No. 1, NYU, Courant Institute, March 1988.

Edler, J., “Upgrading a read lock to a write lock,” Ultracomputer System Software Note No. 54, N.Y.U., Courant Institute, September 1984.

Edler, J., A. Gottlieb, C. P. Kruskal, K. McAuliffe, L. Rudolph, M. Snir, P. Teller, and J. Wilson, “Issues related to MIMD shared-memory computers: the NYU Ultracomputer approach,” *Proceedings of the 12th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 85CH2144-4, pp. 126-135, June 1985.

Edler, J., J. Lipkis, and E. Schonberg, “Process management for highly-parallel Unix systems,” *Proceedings of the 1988 USENIX Workshop on Unix and Supercomputers*, pp. 1-18, September 1988a.

Edler, J., J. Lipkis, and E. Schonberg, “A design for large-scale shared-memory multiprocessors,” *Proceedings of the 1988 USENIX Workshop on Unix and Supercomputers*, pp. 151-168, September 1988b.

Eggers, S., and R. Katz, “A Characterization of sharing in parallel programs and its application to coherency protocol evaluation,” *Proceedings of the 15th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 88CH2545-2, pp. 373-382, June 1988.

Eickemeyer, R., and J. Patel, “Performance evaluation of multiple register sets,” *Proceedings of the 14th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 87CH2420-8, pp. 264-271, June 1987.

Emberson, D., “A cache coherence management technique for hypercube multiprocessors,” *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 262-265, August 1987.

Emer, J., and D. Clark, “A characterization of processor performance in the VAX-11/780,” *Proceedings of the 11th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 84CH2051-1, pp. 301-309, June 1984.

Flynn, M., Private communication, 1989.

Flynn, M., C. Mitchell, and J. Mulder, “And now a case for more complex instruction sets,” *Computer*, **20**, 9, pp. 71-83, September 1987.

Flynn, M., “Some computer organizations and their effectiveness,” *IEEE Transactions on Computers*, **C-21**, pp. 948-960, September 1972.

Frank, S., “Tightly coupled multiprocessor system speeds memory access times,” *Electronics*, **57**, 1, pp. 164-169, January 1984.

Freudenthal, E., and A. Gottlieb, "Process coordination with fetch-and-increment," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Order No. 556910, pp. 260-268, April 1991.

Fuhrt, B., and V. Milutinovic, "A survey of microprocessor architectures for memory management," *Computer*, **20**, 3, pp. 48-67, March 1987.

Gajski, D., D. Kuck, D. Lawrie, and A. Sameh, "CEDAR–a large scale multiprocessor," *Proceedings of the 1983 International Conference on Parallel Processing*, IEEE Catalog No. 83CH1922-4, pp. 524-529, August 1983.

Gimarc, C., and V. Milutinovic, "A Survey of RISC processors and computers of the mid-1980s," *Computer*, **20**, 9, pp. 59-69, September 1987.

Goodman, J. "Coherency for multiprocessor virtual address caches," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE Catalog No. 87CH2440-6, pp. 72-81, October 1987.

Goodman, J. "Using cache memory to reduce processor-memory traffic," *Proceedings of the 10th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 83CH1889-5, pp. 124-131, June 1983.

Gottlieb, A., "An overview of the NYU Ultracomputer project," in J. Dongarra, editor, *Experimental Parallel Computing Architectures*, Amsterdam, The Netherlands: North-Holland, pp. 25-95, 1987. (formerly Ultracomputer Note No. 100, N.Y.U., Courant Institute, July 1986.)

Gottlieb, A., B. Lubachevsky, and L. Rudolph, "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors," *ACM Transactions on Programming Languages and Systems*, **5**, 2, pp. 164-189, April 1983.

Gottlieb, A., R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer – designing an MIMD shared memory parallel computer," *IEEE Transactions on Computers*, **C-32**, 2, pp. 175-189, February 1983.

Gottlieb, A., and C. Kruskal, "Coordinating parallel processors: a partial unification," Ultracomputer Note No. 34, N.Y.U., Courant Institute, September 1981.

Gottlieb, A., "WASHCLOTH 81," Ultracomputer Note No. 21, N.Y.U., Courant Institute, January 1981.

Gottlieb, A., "WASHCLOTH–the logical successor to Soapsuds," Ultracomputer Note No. 12, N.Y.U., Courant Institute, December 1980.

Gross, T., J. Hennessy, S. Przybylski, and C. Rowen, "Measurement and evaluation of the MIPS architecture and processor," *ACM Transactions on Computer Systems*, **6**, 3, pp. 229-257, August 1988.

Gustafson, J., "Reevaluating Amdahl's law," *Communications of the ACM*, **31**, 5, pp. 532-533, May 1988.

Harrison, M., "The add-and-lambda operation: an extension of F&A," Ultracomputer Note No. 104, N.Y.U., Courant Institute, July 1986.

Heath, M., and P. Worley, "Once again, Amdahl's law," *Communications of the ACM*, **32**, 2, pp. 262-264, February 1989.

Heidelberger, P., and S. Lavenberg, "Computer performance evaluation methodology," *IEEE Transactions on Computers*, **C-33**, 12, pp. 1195-1220, December 1984.

Hennessy, J., "VLSI processor architecture," *IEEE Transactions on Computers*, **33**, 12, pp. 1221-1246, December 1984.

Hester, P., R. Simpson, and A. Chang, "The IBM RT PC ROMP and memory management unit architecture," *IBM RT Personal Computer Technology*, Form No. SA23-1057, pp. 48-56, 1986.

Hill, M., *et al.*, "Design decisions in SPUR," *Computer*, **19**, 11, pp. 8-22, November 1986.

Hiromoto, R., "Some issues in parallel processing as encountered on the Denelcor HEP," *Parallel Computing*, **3**, 2, pp. 111-127, May 1986.

Hitchcock III, C., and H. Brinkley Sprunt, "Analyzing multiple register sets," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 85CH2144-4, pp. 55-63, June 1985.

Holliday, M., "Reference history, page size, and migration daemons in local/remote architectures," Technical Report CS-1988-22, Duke University Dept. of Computer Science, July 1988.

Hoshino, T., *Pax Computer*, Reading, MA: Addison-Wesley Publishing Co., 1989.

IBM Corp., *IBM Virtual Machine/System Product CMS User's Guide*, Release 6, July 1988.

James, D., A. Laundrie, S. Gjessing, and G. Sohi, "Distributed-directory scheme: Scalable Coherent Interface," *Computer*, **23**, 6, pp. 74-77, June 1990.

Kalnay-Rivas, E., and D. Hoitsma (Sigma Data Services Corp.), "Documentation of the fourth order band model," NASA Modeling and Simulation Facility Laboratory for Atmospheric Sciences, NASA/Goddard Space Flight Center, Greenbelt, MD, November 1979.

Katz, R., S. Eggers, D. Wood, C. Perkins, and R. Sheldon, "Implementing a cache consistency protocol," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 85CH2144-4, pp. 276-283, June 1985.

Kenner, R., R. Bianchini, S. Dickey, and P. Teller, "A compact design for a highly-parallel shared-memory MIMD computer," *Proceedings of Compcon Spring 1989*, IEEE Catalog No. 89CH2686-4, pp. 264-269, February 1989.

Kenner, R., S. Dickey, and P. Teller, "The design of processing elements on a multiprocessor system with a high-bandwidth, high-latency interconnection network," *Proceedings of the 22nd Hawaii International Conference on System Sciences*, IEEE Catalog No. 89TH0242-8, pp. 319-328, January 1989.

Kim, M., "Synchronized disk interleaving," *IEEE Transactions on Computers*, **C-35**, 11, pp. 978-988, November 1986.

Knapp, V., *Virtually Addressed Caches for Multiprogramming and Multiprocessing Environments*, Doctoral Dissertation, University of Washington, June 1985.

Korn, D., and N. Rushfield, "Washcloth simulation of three-dimensional weather forecasting code," Ultracomputer Note No. 55, N.Y.U., Courant Institute, January 1981.

Kroft, D., "Lockup-free instruction fetch/prefetch cache organization," *Proceedings of the 8th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 81CH1593-3, pp. 81-88, June 1981.

Kruskal, C., L. Rudolph, and M. Snir, "Efficient synchronization on multiprocessors with shared memory," *ACM TOPLAS*, **10**, 4, pp. 579-601, October 1988.

Kruskal, C., M. Snir, and A. Weiss "The distribution of waiting times in clocked multistage interconnection networks," *Proceedings of the 1986 International Conference on Parallel Processing*, IEEE Catalog No. 86CH2355-6, pp. 12-19, August 1986, and *IEEE Transactions on Computers*, **37**, 11, pp. 1337-1352, November 1988.

Kruskal, C., M. Snir, and A. Weiss, "On the distribution of delays in buffered multistage networks for uniform and nonuniform traffic (extended abstract)," *Proceedings of the 1984 International Conference on Parallel Processing*, IEEE Catalog No. 84CH2045-3, pp. 215-219, August 1984.

Kruskal, C., and M. Snir, "The performance of multistage interconnection networks for multiprocessors," *IEEE Transactions on Computers*, **C-32**, 12, pp. 1091-1098, December 1983.

Kuck, D., E. Davidson, D. Lawrie, and A. Sameh, "Parallel supercomputing today and the Cedar approach," *Science*, 231, pp. 967-974, February 1986.

Kuck, D., *The Structure of Computers and Computations, Volume One*, New York, NY: John Wiley & Sons, Inc., 1978.

Kumar, M. and K. So., "Trace driven simulation for studying MIMD parallel computers," *Proceedings of the 1989 International Conference on Parallel Processing*, pp. 168-172, August 1989.

Lamport, L., "How to make a multiprocessor computer that correctly executes multriprocess programs," *IEEE Transactions on Computers*, **C-28**, 9, pp. 690-691, September 1979.

Laha, S., J. Patel, and R. Iyer, "Accurate low-cost methods for performance evaluation of cache memory systems," *IEEE Transactions on Computers*, **37**, 11, pp. 1325-1336, November 1988.

LaRowe, Jr., R., and C. Ellis, "Experimental comparison of memory management policies for NUMA multiprocessors," Technical Report CS-1990-10, Duke University Dept. of Computer Science, April 1990.

Lee, F., "Study of &rprime.look aside′ memory," *IEEE Transactions on Computers*, **C-18**, 11, pp. 1062-1064, November 1969.

Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, *Proceedings of the 17th International Symposium on Computer Architecture*, IEEE Catalog No. 90CH2877-8, pp. 148-159, May 1990.

Levy, H., and P. Lipman, "Virtual memory management in the VAX/VMS operating system," *Computer*, **15**, 3, pp. 35-41, March 1982.

Li, K., and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, **7**, 4, pp. 321-359, November 1989.

Li, K., "IVY: a shared virtual memory system for parallel computing," *Proceedings of the 1988 International Conference on Parallel Processing*, IEEE Catalog No. 88CH2625-2, pp. 94-101, August 1988.

Liles, A., and G. Paul, SEMUL was developed at IBM Research to trace execution of scientific programs.

Liu, Y-S., *Architecture and Performance of Processor-Memory Interconnection Networks for MIMD Shared Memory Parallel Processing Systems*, Doctoral Dissertation, N.Y.U., Courant Institute, September 1990.

Lovett, T., and S. Thakkar, “The Symmetry multiprocessor system,” *Proceedings of the 1988 International Conference on Parallel Processing*, IEEE Catalog No. 88CH2625-2, pp. 303-310, August 1988.

Lubachevsky, B., “Parallelization of the Simple code for an Ultracomputer,” to appear.

Lubachevsky, B., “Parallelizing an algorithm of Charles S. Peskin for describing incompressible flow of a fluid coupled to an elastic membrane,” Ultracomputer Note No. 42, N.Y.U., Courant Institute, January 1981.

Lubeck, O., J. Moore, and R. Mendez, “A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2,” *Computer*, **18**, 12, pp. 10-24, December 1985.

McCreight, E., “The Dragon computer system: an early overview,” NATO Advanced Study Institute on Microarchitecture of VLSI Computers, Urbino, Italy, July 1984.

Motorola Semiconductor Products, Inc., *MC68010/MC68012 16-/32-bit virtual memory microprocessors*, May 1985.

National Semiconductor Co., *NS16082 Memory Management Unit (MMU) Data Sheet*, 1982.

Nelson, M., and J. Ousterhout, “Copy-on-write for Sprite,” *Proceedings of Summer USENIX '88*, pp. 187-199, June 1988.

Norton, A., and G. Pfister, “A methodology for predicting multiprocessor performance,” *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE Catalog No. 85CH2140-2, pp. 772-781, August 1985.

Papamarcos, M., and J. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” *Proceedings of the 11th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 84CH2051-1, pp. 348-354, June 1984.

Patterson, D., “Reduced instruction set computers,” *Communications of the ACM*, **28**, 1, pp. 8-21, January 1985.

Patterson, D., and C. Sequin, “RISC I: a reduced instruction set VLSI computer,” *Proceedings of the 8th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 81CH1593-3, pp. 443-457, May 1981.

Peskin, C., D. McQueen, and S. Greenberg, “Three-dimensional fluid dynamics in a two-dimensional amount of central memory,” *Wave Motion: Theory, Modeling and Computation, Proceedings of the 60th Birthday of Peter D. Lax.*, N.Y.C., NY: Springer-Verlag, 1987.

Peskin, C., “Ultracomputer implementation of odd-even reduction,” Ultracomputer Note No. 19, N.Y.U., Courant Institute, January 1981a.

Peskin, C., “Remarks on efficient numerical algorithms for Ultracomputers,” Ultracomputer Note No. 18, N.Y.U., Courant Institute, January 1981b.

Peskin, C., “Numerical analysis of blood flow in the heart,” *Journal of Computation Physics*, 25, pp. 220-252, 1977.

Pfister, G., and V. Norton, “&rprime.Hot spot′ contention and combining in multistage interconnection networks,” *IEEE Transactions on Computers*, **C-34**, 10, pp. 943-948, October 1985.

Pfister, G., W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss, “The IBM research parallel processor prototype (RP3): introduction and architecture," *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE Catalog No. 85CH2140-2, pp. 764-771, August 1985.

Pohm, A., and O. Agrawal, *High-Speed Memory Systems*, Reston, VA: Reston Publishing Co., Inc., 1983.

Rashid, R., A. Tevanian, M. Young, D. Young, R. Baron, D. Black, W. Bolosky, and J. Chew, "Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE Catalog No. 87CH2440-6, pp. 31-39, October 1987.

Reed, D., and R. Kanodia, "Synchronization with eventcounts and sequencers," *Communications of the ACM*, **22**, 2, pp. 115-123, February 1979.

Rettberg, R., and R. Thomas, "Contention is no obstacle to shared-memory multiprocessing," *Communications of the ACM*, **29**, 12, pp. 1202-1212, December 1986.

Ritchie, S., "TLB for free: in-cache address translation for a multiprocessor workstation," Technical Report No. UCB/CSD 85/233, U. C. Berkeley, Computer Science Division, May 1985.

Rosenburg, B., Private communication, July 1990.

Rosenburg, B., "Low-synchronization translation lookaside buffer consistency in large-scale shared-memory multiprocessors," *Proceedings of the 13th Symposium on Operation Systems Principles*, pp. 137-146, September 1989.

Rudolph, L., *Software Structures for Ultraparallel Computing*, Doctoral Dissertation, N.Y.U., Courant Institute, February 1982.

Rudolph, L., and Z. Segall, "Dynamic decentralized cache schemes for MIMD parallel processors," *Proceedings of the 11th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 84CH2051-1, pp. 340-347, June 1984.

Sanguinetti, J., "Performance of a message-based multiprocessor," *Computer*, **19**, 9, pp. 47-55, September 1986.

Satyanarayanan, M., and D. Bhandarkar, "Design trade-offs in VAX-11 translation buffer organization," *Computer*, **14**, 12, pp. 103-111, December 1981.

Scheurich, C., and M. Dubois, "Dynamic page migration in multiprocessors with distributed global memory," *IEEE Transactions on Computers*, **38**, 8, pp. 1154-1163, August 1989.

Sequent Computer Systems, Inc., *Balance Technical Summary*, MAN-0110-00, pp. B-1–B-9, November 1986.

Shaw, A., *The Logical Design of Operating Systems*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1974.

Smith, A., "CPU cache consistency with software support and using &rprime.one time identifiers′," *Proceedings of the Pacific Computer Communications Symposium*, pp. 142-150, October 1985.

Smith, A., "Disk cache–miss ratio analysis and design considerations," *ACM Transactions on Computer Systems*, **3**, 3, pp. 161-203, August 1985.

Smith, A., "Cache evaluation and the impact of workload choice," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 85CH2144-4, pp. 64-73, June 1985.

Smith, A., "Cache memories," *Computing Surveys*, **14**, 3, pp. 473-530, September 1982.

Smith, B., Private communication, 1989.

Smith, B., "A pipelined, shared resource MIMD computer," *Proceedings of the 1978 International Conference on Parallel Processing*, IEEE Catalog No. 78CH1321-9C, pp. 6-8, 1978.

So., K., A. Bolmarcich, F. Darema, and V. Norton, "A speedup analyzer for parallel programs," *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 653-662, August 1987.

So, K., F. Darema-Rogers, D. George, V. Norton and G. Pfister, "PSIMUL–a system for parallel simulation of the execution of parallel programs," IBM Research Report No. RC11674, January 1986.

Stanford University, "MIPS-X: a high performance computer," Systems Laboratory Technical Progress Report, pp. 7-12, March 1985.

Stone, H., *High-Performance Computer Architecture*, Reading, MA: Addison-Wesley Publishing Co., 1987.

Stone, J., F. Darema-Rogers, V. Norton, and G. Pfister, "The VM/EPEX FORTRAN preprocessor reference," IBM Research Report No. RC11408, September 1985.

Stenström, P., "A Survey of cache coherence schemes for multiprocessors," *Computer*, **23**, 6, pp. 12-24, June 1990.

Strom, R., and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems*, **3**, 3, pp. 204-226, August 1985.

Sweazey, P., and A. Smith, "A class of compatible cache consistency protocols and their support by the IEEE Futurebus," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 86CH2291-3, pp. 414-423, June 1986.

Tang, C., "Cache system design in the tightly coupled multiprocessor system," *Proceedings of NCC*, pp. 749-753, 1976.

Teller, P., "The cost of TLB consistency," *Cache and Interconnect Architectures in Multiprocessors*, Norwell, MA: Kluwer Academic Publishers, pp. 1-14, 1990.

Teller, P., "Translation-lookaside buffer consistency," *Computer*, **23**, 6, pp. 26-36, June 1990.

Teller, P., R. Kenner, and M. Snir, "TLB consistency on highly-parallel shared-memory multiprocessors," *Proceedings of the 21st Hawaii International Conference on System Sciences*, IEEE Catalog No. 85TH0209-7, pp. 184-193, January 1988.

Thakkar, S., M. Dubois, A. Laundrie, G. Sohi, D. James, S. Gjessing, M. Thapar, B. Delagi, M. Carlton, and A. Despain, "New directions in scalable shared-memory multiprocessor architectures," *Computer*, **23**, 6, pp. 71-83, June 1990.

Thakkar, S., P. Gifford, and G. Fielland, "The Balance multiprocessor system," *IEEE Micro*, pp. 57-69, February 1988.

Thakkar, S., and A. Knowles, "A high-performance memory management scheme," *Computer*, **19**, 5, pp. 8-20, May 1986.

Thompson, M., Private communication, January 1988.

Thompson, M., T. Jermoluk, and J. Wagner, "Translation-lookaside buffer synchronization in a multiprocessor system," *Proceedings of the Winter 1988 USENIX Technical Conference*, pp. 297-302, February 1988.

Tzou, S-Y, D. Anderson, and G. Graham, "Efficient local data movement in shared-memory multiprocessor systems," Technical Report No. UCB/CSD 87/385, U. C. Berkeley, Computer Science Division, December 1987.

Vernon, M., E. Lazowska, and J. Zahorjan, "An accurate and efficient performance analysis technique for multiprocessor snooping cache-consistency protocols," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 88CH2545-2, pp. 308-315, May 1988.

Wilson, J., *Operating System Data Structures for Shared-Memory MIMD Machines with Fetch-and-Add*, Doctoral Dissertation, N.Y.U., Courant Institute, June 1988.

Wolfe, M., "More iteration space tiling", *Supercomputing '89*, IEEE Catalog No. 89CH2802-7, pp. 655-674, November 1989.

Wood, D., *The Design and Evaluation of In-cache Address Translation*, Technical Report No. UCB/CSD 90/565, U. C. Berkeley, Computer Science Division, March 1990.

Wood, D., *et al.*, "An in-cache address translation mechanism," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, IEEE Catalog No. 86CH2291-3, pp. 358-365, June 1986.

Yen, W., D. Yen, and K-S. Fu, "Data coherence in a multicache system," *IEEE Transactions on Computers*, **C-34**, 1, pp. 56-65, January 1985.

Zhou, X., "Bridging the gap between Amdahl's law and Sandia Laboratory's result," *Communications of the ACM*, **32**, 8, pp. 1014-1016, August 1989.