Translation Lookaside Buffer Consistency: A Software Approach*

David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill†, and Robert V. Baron
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We discuss the translation lookaside buffer (TLB) consistency problem for multiprocessors, and introduce the Mach shootdown algorithm for maintaining TLB consistency in software. This algorithm has been implemented on several multiprocessors, and is in regular production use. Performance evaluations establish the basic costs of the algorithm and show that it has minimal impact on application performance. As a result, TLB consistency does not pose an insurmountable obstacle to multiprocessors with several hundred processors. We also discuss hardware support options for TLB consistency ranging from a minor interrupt structure modification to complete hardware implementations. Features are identified in current hardware that compound the TLB consistency problem; removal or correction of these features can simplify and/or reduce the overhead of maintaining TLB consistency in software.

1 Introduction

The trend in uniprocessor and multiprocessor operating system design has been toward flexible use of shared memory as an aid to OS performance, application performance and parallelism. SunOS[1] 4.0 is a recent uniprocessor UNIX-based system which relies heavily on memory sharing to support common OS features (e.g. UNIX read/write) [14]. The Mach multiprocessor operating system provides support for copy-on-write and read-write memory sharing between programs, multiple threads of control within a single address space, memory mapped files and user-provided backing storage objects and pagers [21,26,27,30]. These features provide for better use of physical memory, reduced data copy costs, increased parallelism through the availability of lightweight processes scheduled by the kernel on multiple CPUs and user-tailored memory management and sharing which can be critical to the performance of application support services such as database managers [24].

The use of shared memory implies the need to consistently manage several important aspects of multiprocessor state: the instruction and data caches used to funnel information to each CPU and the address translation lookaside buffers (TLBs) which provide page-level virtual-to-physical mappings [5,12,18]. Data cache consistency alone is not sufficient. For example, if one thread of a parallel program remaps a region of virtual memory to contain a mapped file, all other threads which are currently executing need to see a consistent view of this memory region. This implies changing the virtual-to-physical mapping entries of several processors at the same time. Even basic virtual memory management functions such as pagein and pageout will not (in general) work correctly unless the TLBs of all CPUs have the same image of the current state of a physical page.

Although hardware engineers have lavished design time, implementation energy and board space on the problem of maintaining consistent program instruction and data caches in shared memory multiprocessors, they have largely ignored the related problem of TLB consistency [6]. The most compelling reason for this lack of interest has come not from a careful study of the costs of TLB management, but instead from the economic necessities of hardware design which have frequently dictated the use of off-the-shelf microprocessors. Not only do these microprocessors lack hardware support for TLB consistency, they often make hardware consistency management impossible by not providing pinouts for TLB control. The lack of hardware support for TLB consistency implies the need for a software solution; although most multiprocessor operating systems have had to address this problem, there has not yet been a systematic study of the costs of software TLB consistency.

This paper examines the software management of TLB consistency in the Mach operating system. Its extensive use of shared memory of many types both inside the operating system kernel and as a tool for user programming makes the Mach operating system an extreme example of an environment where TLB consistency is of prime importance. This paper describes the design and implementation of Mach virtual memory as it relates to TLB management and examines the costs of TLB management both as a function of system overhead and application execution time.

2 The Mach VM System

Mach is a portable multiprocessor operating system under development at Carnegie Mellon University. Mach's virtual memory system retains compatibility with 4.3BSD UNIX[2] while providing

*This research was sponsored by the Defense Advanced Research Projects Agency (DOD), monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

† Current address is Philips Labs, Briarcliff Manor, NY.

[1] SunOS is a trademark of Sun Microsystems.

[2] UNIX is a trademark of AT&T Bell Laboratories.

significant enhancements to UNIX's memory management capabilities. Specifically, Mach supports

- large, sparse virtual address spaces,
- copy-on-write and read-write memory sharing between tasks,
- read-write memory sharing within tasks,
- memory-mapped files, and
- user-provided backing store objects and pagers.

This new functionality and widespread use of sharing places additional demands on virtual memory hardware that are not present in prior systems.

The programmer's primary view is that Mach's virtual memory system implements address spaces and operations thereon. Each address space is associated with a task that may contain one or more threads of control. All memory within a task's address space is completely shared among its threads; the threads may execute in parallel on multiprocessors. Read-write sharing of portions of address spaces is available between tasks via an inheritance mechanism at task creation. Copy-on-write or virtual copy sharing of memory is aggressively used by many portions of the Mach kernel, including the message passing system and the implementation of the Unix fork operation. Operations supported on Mach address spaces include

- allocation of virtual memory,
- deallocation of previously allocated virtual memory,
- setting protection on virtual memory,
- specification of inheritance of virtual memory, and
- reading or writing memory in some other address space.

These operations may be invoked on arbitrary page-aligned[3] regions of address spaces.

Mach cleanly separates the implementation of virtual memory into machine-independent and machine-dependent portions. The primary implementation of address space operations is in the machine-independent portion of the system. As a result the machine-dependent portion consists of a single module, the physical map or pmap module, that implements a simple interface to the memory management hardware. Machine-independent code associates a pmap with each address space, and makes calls on the pmap module as needed to perform physical map operations. These operations include:

- validate, invalidate, and protection change operations on virtual address ranges,
- invalidate and protection change operations on physical pages, and
- bookkeeping operations that allow the pmap module to keep track of which pmaps are in use on which processors.

Parallel execution of a task with multiple threads results in the same pmap being used simultaneously by multiple processors. More details can be found in [28].

Extensive lazy evaluation is used in determining when to invoke pmap operations. The Mach VM system maintains all memory management information in machine-independent data structures, and does not need to consult the pmap module for address validity or mapping information. As a result pmaps usually do not present a complete view of valid memory for any address space because they are lazily updated as required by page faults. Pmaps can even be destroyed at runtime; they will be reconstructed from scratch as page faults occur.

Operations on Mach address spaces may require actions to maintain TLB consistency when virtual memory is deallocated or its protection is reduced; these actions consist in invalidating inconsistent entries in remote TLBs (i.e. TLBs for processors other than the one executing the address space operation). There are two situations in Mach that require such actions, invoking an operation on the address space of a multi-threaded task that is executing in parallel on more than one processor, and invoking an operation on the address space of another task that is executing on a different processor. For the purposes of TLB consistency the kernel is viewed as a multi-threaded task that is potentially executing on all processors of a multiprocessor.

TLB consistency is among the hardware-specific implementation details that are confined to the pmap module. This can be viewed as an instance of policy-mechanism separation; the machine-independent VM system invokes operations that require TLB consistency actions, but it is up to the pmap module to determine when and how to carry out these actions. The design of the pmap structure itself is also encapsulated within the pmap module to permit support for whatever data structures may be required by the hardware; the machine-independent portion of Mach does not depend on either the internals of the pmap module or the hardware support mechanisms for virtual memory.

3 The TLB Consistency Problem

Maintaining TLB consistency on current multiprocessors is more difficult than it appears at first glance. A naive solution would be for the processor executing an operation that might cause an inconsistency to simply invalidate the entries in the remote TLBs and proceed. This will not work because most multiprocessors do not permit processors to perform invalidate operations on TLBs other than their own. Something else is required: either hardware changes to support remote invalidation of TLB entries, or a software notification mechanism (e.g. interrupts) to cause other processors to perform invalidations. These changes are still insufficient to solve the problem due to two TLB features:

1. Hardware reload mechanisms can reload inconsistent entries after they are flushed.
2. TLBs can write inconsistent entries back to memory in order to set reference and/or modify bits.

Flushing entries before changing the physical map will not work due to the first feature, and the second feature can corrupt physical map changes if flushing is postponed until after the physical map is changed. As a result, it is necessary to stall remote processors during updates of physical maps.

Earlier papers on the Mach VM system [21] discussed three potential techniques for handling TLB consistency on hardware that does not support remote flushing of TLBs (i.e. most current commercial multiprocessors):

1. Notify processors to carry out consistency actions.
2. Delay use of changed mappings until all buffers have been flushed (e.g. by code executed in response to timer interrupts).
3. Allow temporary inconsistency in cases where it does not cause problems (e.g. if protection is being increased).

The Mach kernel implementation relies on the first technique because the additional buffer flushes required by the second technique

[3]This page alignment is with respect to Mach virtual pages which may be a multiple of the actual hardware page size.

can be expensive on some architectures [9], and the third technique is not a complete solution—it is an optimization that can be applied to any TLB consistency technique that correctly handles the decreased protection and invalidated mapping cases.

4 The Shootdown Solution

Mach pmap modules maintain TLB consistency by forcibly interrupting processors to perform TLB consistency actions (i.e. entry or buffer flushes). This forcible interruption is referred to as "shooting" entries out of remote TLBs, and the entire process of causing remote entries to be invalidated is called a "shootdown". The shootdown algorithm is divided into two portions: the code executed by the initiator (send interrupts if an inconsistency might occur as a result of a pmap operation), and the code executed by the responder (receive interrupts and perform the consistency actions).

The shootdown algorithm manipulates a small collection of data structures:

- A set of active processors. This is the set of processors that are actively performing virtual to physical translations on any pmap.
- A set of idle processors.
- For each pmap, a set of processors that is using that pmap.
- For each processor, a flag that indicates that a TLB consistency action is needed, and buffers to hold queued consistency actions.

The shootdown algorithm is invoked when a pmap operation detects that the changes it is about to make to a pmap could cause a TLB inconsistency. Invoking a shootdown guarantees that any inconsistent TLB entries caused by this operation will not be used after the operation completes. The algorithm proceeds in four phases after it is invoked (the initiator holds an exclusive lock on the pmap at invocation):

1 Initiator: The initiator queues consistency action requests for all processors using the pmap and sets their "action needed" flags. It then sends interrupts to the processors and waits for responses.

2 Responders: Each responder receives its interrupt, and removes itself from the set of active processors to acknowledge the interrupt. The responders then spin until the initiator completes its changes to pmap. (This spinning is necessary to ensure that responders neither read nor write the pmap while the update is in progress.)

3 Initiator: The initiator performs its pmap changes after all responders using the pmap are spinning. It unlocks the pmap when it is done.

4 Responders: The responders perform their required TLB invalidations after the pmap is unlocked and dequeue the corresponding actions. They also clear their "action needed" flags and rejoin the set of active processors.

This description omits many details in order to expose the underlying structure. The algorithm itself is considerably more complicated for the following reasons, most of which involve interrupts:

- A responder may cease using the pmap after the shootdown interrupt is requested.
- Concurrent shootdown operations are a source of potential deadlocks.
- Interrupt protection is necessary during a shootdown.
- Inconsistent interrupt protection of locks is another source of potential deadlocks.
- Idle processors should not receive shootdown interrupts for performance reasons.

If a responder ceases using the pmap before receiving its interrupt, there is no need for the initiator to synchronize with this responder because it has flushed all entries for this pmap from its TLB. Hence the initiator can proceed once it notices that the responder is no longer using the pmap. This is implemented by making the initiator's check for a response be a check for the responding processor to either become inactive or stop using the pmap.

The deadlocks mentioned in the second item can be caused by two initiators shooting at each other. Deadlock can then occur with both initiators waiting for the other to continue. This can only happen if the shootdowns are on different pmaps (i.e. the kernel pmap and a user pmap) because the pmap locks prevent concurrent shootdowns on the same pmap. Initiators avoid these deadlocks by disabling shootdown interrupts and removing themselves from the set of active processors. Interrupts are still sent to such initiators, but there is no need to synchronize with their receipt because the interrupts will be acted upon before performing any memory references that may use inconsistent TLB entries.

Responders must disable further shootdown interrupts while servicing one to avoid duplicate add and remove operations on processor sets (most hardware does this by default). Blocking interrupts also enhances response to concurrent shootdown operations because a single instance of the responder's algorithm responds to all shootdowns in progress. Finally, both the initiator and responder should disable all interrupts during a shootdown to avoid delays to the synchronization (a device interrupt at the wrong point could stall the entire machine).

The potential deadlocks in the fourth item result from an interaction of the shootdown algorithm's barrier synchronization at interrupt level with inconsistent interrupt protection of locks. They are avoided by associating a fixed interrupt priority (with respect to the shootdown interrupt) with every lock in the system. Locks are requested at their associated interrupt priority level and can only be held at that level or higher. This requires careful design and coding in modules that can be called by device interrupt routines and in modules that make upcalls. A complete explanation of these deadlocks and their avoidance is beyond the scope of this paper.

The fifth item enhances performance of kernel pmap shootdowns by not sending shootdown interrupts to idle processors, and hence reducing the initiator's synchronization time (initiators still queue actions for idle processors, but do not synchronize with them). The set of idle processors is used to implement this feature; idle processors must check for queued consistency actions and execute them before becoming active.

Incorporating these refinements into the basic description presented earlier yields the full shootdown algorithm shown in Figure 1. English has been liberally substituted for actual code and some details have been omitted for clarity. The variable `mycpu` identifies the current processor. The variables `start` and `end` identify the address range that must be invalidated from the TLBs to prevent an inconsistency. Arrays have been used to implement the `active` and `in_use` processor sets. `disable_interrupts` returns the previous interrupt state; disabling and restoring interrupt state may be done by hardware for the responders. The numbers in comments identify the algorithm phases enumerated above.

```
Initiator:
  s = disable_interrupts();
  active[mycpu] = FALSE;
  lock_pmap(pmap);
  if (inconsistent TLB may result) {
    if (pmap->in_use[mycpu]) {
      invalidate_tlb(pmap,start,end);
    }
    /* Phase 1 */
    if (other cpus using pmap) {
      list_type shoot_list = EMPTY_LIST;

      for (every cpu in system) {
        if (pmap->in_use[cpu] &&
            cpu != mycpu) {
          lock_action_structure(cpu);
          queue_action(cpu,pmap,start,end);
          action_needed[cpu] = TRUE;
          unlock_action_structure(cpu);

          if (idle[cpu] == FALSE) {
            add cpu to shoot_list
          }
        }
      }
      for (every cpu on shoot_list) {
        send_shootdown_interrupt(cpu);
      }
      for (every cpu on shoot_list) {
        while (active[cpu] &&
               pmap->in_use[cpu]) {
          /* spin */ ;
        }
      }
    }
  }
  /* Phase 3 */
  make changes to physical map

  unlock_pmap(pmap);
  active[mycpu] = TRUE;
  restore_interrupt_state(s);

Responders:  /* Phase 2 */
  s = disable_interrupts();
  while (action_needed[mycpu]) {
    active[mycpu] = FALSE;
    while (pmap_is_locked(kernel_pmap) &&
           pmap_is_locked(user_pmap(mycpu)))
      /* spin */ ;

    /* Phase 4 */
    lock_action_structure(mycpu);
    process_queued_actions(mycpu);
    action_needed[mycpu] = FALSE;
    unlock_action_structure(mycpu);
    active[mycpu] = TRUE;
  }
  restore_interrupt_state(s);
```

Figure 1: Mach Shootdown Algorithm Pseudo-Code

Three important details have been omitted from the pseudo-code in Figure 1:

1. The invalidation mechanism for TLBs is not specified. For hardware which supports both a single entry invalidate and an entire buffer flush, the mechanism chooses between individual invalidates and a buffer flush based on the number of entries that must be invalidated (i.e. beyond some threshold it is faster to flush the entire buffer than to do the individual invalidates); this threshold depends on hardware factors (buffer size, speed of invalidates and flushes, etc.).
2. The update queue for each processor is a small buffer. If the initiator detects overflow, it sets a flag that causes the responder to flush its entire TLB. The queue size is set so that this only happens in cases where the responder would flush its entire TLB for efficiency reasons in the absence of update queue overflow.
3. A check is made to see if a shootdown interrupt is already pending for a processor before adding it to the list of processors that will receive shootdown interrupts.

Lazy evaluation of pmaps enhances the effectiveness of the shootdown algorithm by avoiding shootdowns for pages that are never used. This is implemented by having the check for potential inconsistencies determine if the pages in question are mapped in the physical map; if the pages are not mapped, then a shootdown is not necessary because TLBs do not cache invalid mappings. This occurs frequently due to the extensive lazy evaluation of pmap operations by the Mach VM system.

5 An Evaluation of TLB Shootdown

Our evaluation of the Mach TLB shootdown algorithm includes experiments to determine its cost, performance, and overhead impact. The basic costs for TLB shootdown are determined from experiments that involve a simple program for testing TLB consistency. More comprehensive performance and overhead impact results are obtained from experiments with applications that reflect the current use of multiprocessors in a research and production environment. There was no pageout activity during any of our experiments. Pageout does cause shootdowns, but the overhead of actually performing the pageout is much greater than the overhead of the associated shootdown. Our results apply primarily to systems with adequate physical memory for applications.

5.1 Testing TLB Consistency

We developed a simple program that tests whether TLB consistency is being maintained. This program tries to cause a simple TLB inconsistency and then attempts to detect its effects; if consistency is being maintained, there will be no effects. The program employs multiple threads to achieve parallel execution on a multiprocessor, and functions in the following manner:

1. Allocate a page of read-write memory.
2. Start up child threads. Each thread runs in a tight loop, incrementing a separate counter in this page of memory.
3. After threads are all started, the main thread reprotects the page of counters to be read-only and immediately saves a copy of the counters.
4. The spinning child threads all take unrecoverable page faults (write fault on a read-only page).

5. After the page faults, the values of the counters are compared with the saved copy.

A difference in the counter values means that some thread continued to increment its counter after the page containing it became read-only. This can only happen if an inconsistent TLB entry continues to allow write access to the page, and therefore indicates a TLB inconsistency.

We found this program useful not only as a check on correctness, but also as a performance evaluation tool; on an n-processor multiprocessor, running this program with k<n child threads causes exactly one shootdown on its user pmap involving exactly k processors. This can be used to measure the basic overhead of TLB shootdown.

5.2 Evaluation Applications

To evaluate the performance impact of TLB shootdown, we asked several research groups at CMU to provide applications that typify their use of the Multimax. The groups were kind enough to provide us with the following applications:

Mach (operating system): parallel build of the kernel from sources.

Parthenon (parallel theorem prover): Parthenon running 15-way parallel on a difficult standard example [3].

Agora (support base for heterogeneous parallel/distributed systems): Double ended wavefront-based shortest path search program based on the Agora system [2]. Program runs 15-ways parallel.

Camelot (transaction processing): 8-way parallel run of transaction performance analyzer to stress transaction throughput capabilities [23].

Both the Parthenon and Agora applications were run five times in succession to increase the number of shootdown events for which data was collected.

The applications vary in their sophistication of memory use. The Mach kernel build uses multiple processors only for throughput; it does not share memory among user tasks. Parthenon allocates memory as needed to hold the intermediate results of the proof search. Agora uses shared write-once memory for communication among the tasks performing the search. Camelot makes aggressive use of memory sharing and copy-on-write mapping to implement database access and transaction semantics. In addition, many internal components of the Camelot system (e.g. the transaction manager) are multi-threaded for performance reasons. These applications exercise the shootdown algorithm under a variety of memory usage patterns.

6 Measurement Techniques

Our measurements were taken on an instrumented Mach kernel running on a 16 processor NS32332 Encore Multimax. The Multimax is a shared-memory bus-based multiprocessor with write through caches. Its system control card provides a free-running 32-bit microsecond counter from which timestamps can be obtained [13]. The xpr package[4] forms the basis of our instrumentation; it provides a circular buffer of events including data arguments, event identifiers, processor numbers and timestamps. At each event to be monitored, we added code to gather up the data arguments and pass them to the xpr package; it does the rest. The event buffer managed by xpr was sized so that it would never overflow during our test runs.

There are two events of interest in monitoring the shootdown code. For the initiator we save the following items in one event record:

- A flag indicating whether this shootdown is on the kernel pmap or some user pmap.
- Number of Mach VM pages involved in the shootdown.
- Number of processors being shot at.
- Elapsed time from invoking the shootdown algorithm until the initiator can begin making its changes to the pmap.

For the responder we save the elapsed time in the interrupt service routine. This is a slight underestimate because it ignores the interrupt dispatch and return times. To avoid lock contention effects in the xpr package we only record responder events on 5 selected processors. In addition we wrote a number of utility programs to control the instrumentation (e.g. on, off, reset), read the collected data, and perform statistical analysis.

6.1 Measurement Validation

An important issue in instrumenting a hardware or software system is whether the instrumentation affects the behavior it is measuring. For this work, the particular question is whether introduction of kernel instrumentation for TLB shootdowns affects the performance of the applications that we are using to evaluate shootdown behavior. To assess this, we chose the application that is most vulnerable to performance perturbations, Parthenon, and ran it with and without instrumentation to assess this impact. The potential performance impact for these tests was deliberately increased by disabling the lazy evaluation feature of the shootdown algorithm.

Parthenon is highly vulnerable to performance perturbations because it is a search program with an essentially non-deterministic control structure; worker threads remove work from a central workpile and add new work as it is generated. Perturbations in the runtime of these threads change the order of the add and remove operations on the central workpile, and thus the order in which possibilities are searched. This impacts Parthenon's execution time because Parthenon only searches for a single proof rather than exhaustively searching for all proofs at a given depth, and hence the order in which possibilities are considered can greatly affect the time it takes to find a proof. Parthenon has obtained super-linear speedups on some problems from this effect—a parallel search will consider some possibilities much earlier in its execution than a serial search; if one of these possibilities quickly leads to the proof, the potential savings of unproductive effort over the serial case is enormous.

The results from five runs of Parthenon indicated a perturbation in runtime of about 1.5% (4 seconds out of 253). This result is not statistically significant, and the effect is swamped by other effects (e.g. timer interrupts) that produce perturbations of 8–10% in Parthenon's runtime. We conclude that our kernel instrumentation does not significantly perturb the behavior of the applications we are measuring, and that our results are therefore representative of shootdown behavior on uninstrumented kernels.

[4] Implemented in the Mach kernel by Steve Stone.

7 Results

Our evaluation of the shootdown algorithm has three major goals

- Determine the basic costs of shootdown.
- Measure the effects of lazy evaluation.
- Measure overhead and performance impact for typical applications.

The experiments to achieve the first goal involve measuring the performance of the shootdown algorithm while running our test program. The remaining two goals are achieved by experiments that involve collecting data while running the evaluation applications provided by research projects at Carnegie Mellon.

7.1 Basic Costs of Shootdown

The ability of our shootdown test program to cause a single shootdown involving a predetermined number of processors was used to measure the basic costs of TLB shootdown. Varying the number of child threads used by the tester from 1 to 15 causes the number of processors involved in the shootdown to also vary from 1 to 15. The tester was run ten times for each case; means and standard deviations were calculated for the resulting data. Figure 2 plots the resulting data means with error bars of plus or minus the standard deviation. The data exhibits a pronounced change between 12 and 13 processors; the points depart significantly from the trend line established by the data for smaller numbers of processors, and the standard deviation doubles. This leads us to believe that some unexpected/unrelated effect is coming into play when more than 12 processors are involved (the shootdown algorithm does not change when more processors are involved). Bus contention and congestion effects are likely candidates; previous experiments have shown that these effects become significant on the Multimax when 12 or more processors are actively using the bus (e.g. for block copy of data) [21]; both the saving of state (i.e. registers) in response to the interrupts and the access to shootdown state by the responders can be expected to miss in cache. Excluding the data for 13–15 processors, we obtain the trend line shown in the figure by a least-squares fit. From its equation we calculate the basic cost of a shootdown as 430 microseconds for the first processor plus 55 microseconds for every additional processor involved.

This cost measures the time it takes the initiator from starting the shootdown until it can proceed with its pmap changes. Measuring responder synchronization (i.e. spin) times would not be meaningful for this experiment because these times strongly depend on the duration of the pmap operation that invokes the shootdown; the shootdown tester only exercises shootdowns from one operation (pmap_protect) of short duration (only 1 page involved). In addition the shootdown algorithm makes it very difficult to distinguish between responses to shootdowns on user pmaps and responses to shootdowns on kernel pmaps (the algorithm can handle shootdowns on both pmaps in one response).

7.2 Effectiveness of Lazy Evaluation

We also performed experiments to assess the contribution of lazy evaluation to the performance of the shootdown algorithm. We removed most of the lazy evaluation for shootdowns by disabling the check for valid mappings in the check for potential inconsistencies. The remaining lazy evaluation comes from internal pmap module knowledge about the structure of Multimax page tables. The Multimax uses the NS32382 MMU which employs two-level page tables, and the pmap module organizes the second level tables

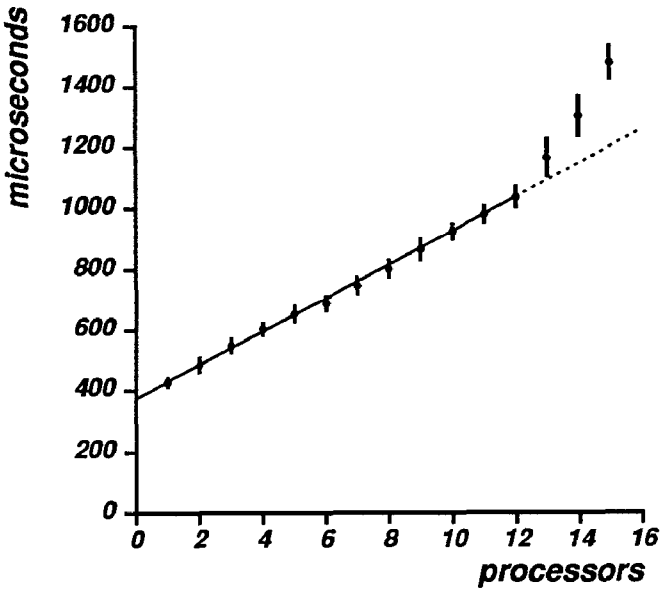

Figure 2: Basic Costs of TLB Shootdown

Application	Mach		Parthenon	
Lazy	No	Yes	No	Yes
Kernel Events	8091	3827	107	4
Avg. Time	1185	1020	1379	1395
User Events	0	0	70	0
Avg. Time			867	

Table 1: Effect of Lazy Evaluation on Shootdowns

into page-sized chunks. Therefore if the pmap module ever finds a missing second level page table entry, it knows that an entire page of second level entries is missing and skips the corresponding address range. Removing this code would significantly increase the cost of some important pmap operations (e.g. destroying a pmap) and impact our evaluation applications.

To assess the effects of lazy evaluation we ran both the Mach and Parthenon applications with lazy evaluation on and off. These were run on a kernel in which the Unix compatibility code has not been parallelized, so the Mach kernel build rapidly saturates the processor that executes the Unix code, thus limiting speedup. The results for initiators are reported in Table 1. All times are in microseconds. These results show the pronounced effect of lazy evaluation; it reduces the total overhead (number of events times average time per event) of shootdowns by almost 60% for the Mach kernel build and all but eliminates it (reduction of over 97%) for Parthenon.

The elimination of user pmap shootdowns for Parthenon is a good example of the benefits of lazy evaluation. These shootdowns are caused by code in the cthreads library [10] that sets up stacks for new threads. This code allocates a large aligned stack region, reserves the first page in the region for private data, and reprotects the second page to no access in order to detect stack overflows. Without lazy evaluation the reprotection operation causes a shootdown if more than one thread is running. The lazy evaluation check notices that the stack page in question is not mapped (because it has not been accessed) and therefore does not require a shootdown. The net effect is to remove an average four-fifths of a millisecond from the startup time for new threads. This and other savings from lazy evaluation are well worth the time spent in the valid mapping check (approximately 2 instructions per check).

Application	Mach	Parthenon	Agora	Camelot
Events	7494	4	88	68
Processors	6.3±9.3	1.0±0.0	4.0±4.9	9.3±2.0
Pages	2.3±1.4	1.0±0.0	1.0±0.5	1.3±0.8
Time	1109±1272	1395±1431	1425±1911	1641±1994
Median	939	645	NM	1171
10^{th}%ile	784	NM	NM	1031
90^{th}%ile	1488	NM	NM	3466

Table 2: Kernel Pmap Shootdown Results: Initiator

Application	Camelot
Events	5407
Processors	3.9±0.2
Pages	1.3±0.7
Mean Time	588±591
Median	573
10^{th}%ile	545
90^{th}%ile	662

Table 3: User Pmap Shootdown Results: Initiator

Application	Mach	Parthenon	Agora	Camelot
Events	15316	0	85	7547
Mean Time	548±796		565±944	325±377
Median	425		NM	318
10^{th}%ile	237		NM	181
90^{th}%ile	817		NM	440

Table 4: Responder Results

7.3 Shootdown Overhead and Performance

The major results from our evaluation applications are reported in Tables 2, 3, and 4. All times are in microseconds. Results are reported as mean±standard deviation. The Mach build results here were recorded on a kernel with most of the Unix code parallelized to obtain better speedups. NM indicates that the number is not meaningful due to either insufficient data or an unusual distribution (in the statistical sense). Table 3 contains results solely from Camelot because the other three applications did not cause any user shootdowns. Any comparisons of the raw number of events should take into account the different runtimes of the evaluation programs: about 7.5 minutes for Agora, 20 minutes for the Mach build and Parthenon, and one hour for Camelot. In addition, the responder data represent approximately one-third of the actual responder events because data was only collected on 5 out of 16 processors.

Most of the time distributions are not normal in the statistical sense. The distributions are skewed towards high frequencies at low values; this is demonstrated by the greater difference between the 90^{th} percentile and the median than between the 10^{th} percentile and the median. The median is a better indicator of typical or expected values for such data than the mean, but the mean is still useful for calculating the total overhead. The Camelot responder times are an exception; their distribution is nearly symmetric as evidenced by the near agreement between mean and median.

Medians and percentiles are not meaningful for the Agora data due to the bimodal nature of its shootdown events. Agora causes shootdowns involving large numbers of processors only during its setup phase; once it has allocated the memory internally, the 15-way parallel shortest path program can be run again and again without causing any large shootdowns. As a result the Agora kernel shootdown data splits into two groups; the shootdown events during setup (11 to 15 processors involved, median time of 1367 μsec), and the remaining events (1 to 4 processors involved, median time of 779 μsec). Medians and percentiles are not meaningful for the resulting bimodal distribution.

8 Performance Analysis

Our results show that shootdowns impose greater costs on initiators than responders. There are two causes for this:

1. The typical pmap operation that is executed during a shootdown is short due to the small number of pages involved (usually 1).
2. The average responder only waits for half of the total responders, whereas any initiator must wait for all responders.

This combination reduces the responders' spin times to less than the initiators' setup and synchronization times. This was a surprise to us, as we had expected to find responders spinning for extended periods of time while pmap operations were performed.

The results also indicate that shootdowns on the kernel pmap behave differently from shootdowns on user pmaps. Kernel pmap shootdowns take longer, as can be seen by comparing the data in Table 2 with the basic cost data for user pmap shootdowns reported in Section 7.1. In addition, the kernel times exhibit a greater skew than the user times. The major causes of both effects are that the kernel disables shootdown (and other) interrupts in many places, and that kernel pmap shootdowns are far more likely to find at least one responder in the kernel with interrupts disabled than user pmap shootdowns are. A responder with interrupts disabled extends the shootdown times because the initiator cannot proceed with its pmap changes until all responders respond to their interrupts. The additional skew in the kernel time distributions is caused by the varying intervals for which interrupts are disabled; there are many short intervals, but few long ones. A secondary contribution to longer kernel pmap shootdown times is that kernel pmap shootdowns must queue action requests for all processors in the system including idle processors, while user pmap shootdowns only involve processors that are actually executing the user task.

The most important conclusion to be drawn from these results is that the overhead of maintaining TLB consistency in software is almost negligible on current machines. After scaling the overheads upward to represent shootdowns across the entire machine, the largest overheads are still in the neighborhood of 1% for kernel pmap shootdowns (Mach) and less than 0.2% for user pmap shootdowns (Camelot); due to the pessimistic scaling, these numbers overstate the actual performance impact.

Performance impact on future machines can be extrapolated from this data; the fact that shootdown overhead scales linearly with the number of processors is a warning that shootdown overhead may pose problems for larger machines. Extrapolation of our results predicts that user pmap shootdowns will not present performance problems on machines with a few hundred processors, but that kernel pmap shootdowns might (the 1% overhead figure for kernel pmap shootdowns for the Mach build could reach 10% or more on a machine of that size). Operating systems for such machines may have to restructure their use of memory to limit shootdowns; similar changes may be required in any case because physical constraints put uniform memory access designs at a disadvantage for machines in this class. Virtually all current and proposed shared-memory machines in this class (e.g. RP3 [19] and the Butterfly [11]) utilize a non-uniform memory access structure. One possible restructuring is to divide both the processors and the kernel virtual address space into pools that mirror the non-uniform memory structure. One can then identify memory within the kernel that may require shootdowns (due to pageability or internal use of copy-on-write) and

restrict sharing of it between pools. This results in most kernel pmap shootdowns occurring within pools of processors instead of across the entire machine.

The Agora and Parthenon evaluation applications are parallel programs that make extensive use of shared memory. Agora causes no machine wide shootdowns after its setup phase, and Parthenon causes almost no shootdowns whatsoever. This suggests that the overhead of maintaining TLB consistency has essentially no impact on conventional parallel programs because consistency actions are required on an extremely infrequent basis. These results have been confirmed by the RP3 group at IBM Hawthorne; they have found that for typical parallel applications on a 4-way RP3 prototype "the number of cross-machine TLB invalidations is so small that the time spent doing them has no impact on performance" [22]. It should be emphasized that these results hold for conventional parallel programs that make static use of shared memory (allocate it once and use it).

9 Hardware Design Implications

The most aggressive hardware support for TLB consistency involves a complete hardware implementation; this is analogous to the complete hardware implementations of cache consistency found on many current multiprocessors. There are two obvious forms that such an implementation could take:

1. For hardware-reloaded TLBs, take advantage of the cache consistency protocol so that a TLB entry is invalidated when the memory location it was loaded from is written.
2. For software-reloaded TLBs, use a bus-based invalidation protocol. One possibility would be to base the invalidations on physical instead of virtual addresses (i.e. dedicate a range of physical addresses on the bus to be used for invalidating TLB entries by encoding the physical page number in the low order bits[5]). This has the advantage of enhanced performance for operations that require invalidation of all entries for a physical page (e.g. pageout), but suffers from the corresponding disadvantage of being too aggressive when not all of the TLB entries for a physical page need to be invalidated.

Another alternative is to use virtual address caches. This completely eliminates the TLB consistency problem by eliminating the TLBs. Unfortunately it substitutes a mapping consistency problem that is more difficult to solve; invalidating a page mapping can require that the page be flushed from all virtual caches. The designers of VMP (a proposed multiprocessor with virtual caches) have chosen to implement this flush by "an exhaustive search of the cache directory for [entries] in the specified range, with a few optimizations" in software on every processor that has the page mapped [8]. They also claim that this and similar functionality is prohibitively difficult to implement in hardware [7] with current technology. The resulting increase in invalidation overhead should be considered by multiprocessor designers when choosing between virtual and physical cache designs.

The low overhead of maintaining TLB consistency in software on current machines may not justify a complete hardware implementation. Our work suggests a number of hardware features that can reduce the overhead of maintaining consistency by providing partial hardware support. Detailed performance and cost/benefit evaluations of these proposed features are topics for future research.

The first feature on our list is a high-priority software interrupt. Operating systems need to mask device interrupts to prevent deadlocks and delays caused by interrupt routines (e.g. interrupt routine tries to grab lock held by interrupted code), but shootdown interrupts are not device interrupts and cannot cause deadlocks. A software interrupt with priority above that of device interrupts would allow us to disable device interrupts without blocking shootdowns; this would reduce the time for kernel shootdowns to more closely match user shootdowns, and eliminate the skew caused by long periods of interrupt disablement. Aside from the shootdown code (which must disable shootdown interrupts), we would disable shootdown interrupts in only a few other places in the kernel (places where critical locks are held), as opposed to the widespread disabling of device interrupts. Even a non-maskable interrupt could be used by implementing the masking in software (i.e. set a flag and immediately dismiss the interrupt when it is "masked", then check the flag when "unmasking" the interrupt).

Hardware support for multicast interrupts would also help. The shootdown algorithm does not specify the implementation of the list of processors to which interrupts will be sent. This list can easily be maintained as a bit vector which is then loaded into the hardware to cause the interrupts. This replaces a for loop in the initiator's code with a small number of instructions. Even a simple interrupt that is broadcast to all other processors would be helpful; beyond some number of processors it is faster to use a broadcast interrupt (and interrupt too many processors) than it is to iterate down the list interrupting one processor at a time.

Redesigning TLBs for multiprocessors can eliminate most of the responder overhead. In Section 3 we showed that the combination of hardware reload with asynchronous writeback of reference and/or modify bits requires stalling the responders. The following techniques (among others) can eliminate these stalls and the barrier synchronizations that are required to implement them:

- Substituting software reload for hardware reload allows the responder to return immediately instead of stalling; software can check whether the pmap is being modified when a reload is needed and only stall in that case. The MIPS microprocessor [16] uses this technique.
- Eliminating reference and modify bit writeback allows the shootdown interrupts to be postponed until after the pmap change is completed. The reference and modify bits themselves can be completely eliminated at the cost of using page faults to detect page modifications. RP3 [4] uses this technique.
- Interlocking MMU access to the reference and modify bits also allows postponing shootdown interrupts because it eliminates the potential page table corruption by accesses that set these bits. Accesses to set the bits should be interlocked read-modify-write accesses, and the read data must be checked in all cases for mapping validity[6]. Software for such TLBs must invalidate the old mapping in the page table and all TLBs before entering a new valid mapping in the page table. The MC88200 uses this technique[17]. The 80386 attempts to use this technique, but it is not clear whether mapping validity is correctly checked in all cases [15].

TLBs that support remote invalidation of entries can eliminate shootdown interrupts entirely if the reference/modify bit writeback problem is successfully addressed. The initiator can shoot the entries directly out of the responders' TLBs without involving the responders. This eliminates virtually all of the responder overhead

[5]VMP[8] will use a similar mechanism for a different purpose.

[6]The critical case involves setting the modify bit for a mapping that is cached in the TLB and already has the reference bit set. If the page table entry read from memory does not indicate a valid mapping, then a page fault must occur.

because responders are not involved in the invalidation of their TLBs. Minor overhead may result from responders taking page faults on invalid page table entries that are in the process of being updated; this should be a rare event. In addition, initiator overhead is greatly reduced because it is no longer necessary to synchronize with the responders. The Motorola MC88200 Cache/Memory Management Unit employs this technique [17].

If the barrier synchronization cannot be eliminated, then barrier synchronization hardware would also be useful if processors can opt out of the barrier. This would replace the final for loop in the initiator's algorithm with a barrier check. Processors must be able to opt out of the barrier because a responder may cease to use a pmap before its shootdown interrupt is delivered, and therefore does not need to take part in the shootdown barrier synchronization. By itself, TLB shootdown probably cannot justify the implementation of barrier synchronization hardware, but it can take advantage of such hardware if it has been implemented for other uses (see [20] for an example of such hardware and its potential uses).

10 Related Work

The increasing sophistication of multiprocessor applications has resulted in a corresponding increase in the complexity of the TLB consistency problem. The initial use of multiprocessors for increased throughput confined the consistency problem to the operating system because applications could not share memory. Subsequent additions of shared memory functionality to these systems have been limited to avoid causing major consistency problems (e.g. System 5 shared memory does not support operations on remote address spaces or parallelism within address spaces [1]). Relatively straightforward techniques suffice to handle TLB consistency in these cases (e.g. postpone releasing pages freed by pageout until buffers can be flushed system-wide) [29].

Thompson et al. [29] describes the implementation of TLB consistency on a multiprocessor based on the MIPS microprocessor. Their TLB consistency problem is considerably simpler than the one Mach faces because they worked within System 5.3 Unix which does not support either parallel execution within an address space or operations on remote address spaces. The MIPS microprocessor does present an additional feature not found on the multiprocessors we have worked with; the TLB is not flushed automatically on context switch. Instead entries are tagged with an address space identifier to allow entries from different address spaces to coexist in the same buffer. The Mach shootdown algorithm can be extended to handle such buffers by ignoring the bookkeeping call that informs the pmap module that a pmap is no longer in use on a processor; the pmap is considered in use until its entries are explicitly flushed from that processor's TLB. We would also experiment with changing the responder's invalidate code to completely flush entries for any address space that requires an invalidation even though it is not currently being used by a thread on that processor.

Teller et al. [25] proposes algorithms for TLB consistency on large-scale multiprocessors. Three algorithms for maintaining TLB consistency are presented, one of which could be retrofitted to current systems with software reload of TLBs; the other two require extensive hardware modifications. There are a number of drawbacks to using these algorithms in a practical system:

- The authors fail to solve the problem in full generality by assuming that reductions in page protection are always caused by the use of copy-on-write. This is not the case for Mach. This assumption also reduces their problem to that found in a parallel System 5 Unix for which simpler techniques (e.g. those in [29]) suffice.
- Their algorithm that can be retrofitted to existing systems "must forgo the copy-on-write optimization for any page that is resident in a TLB." This is a poor decision given the large performance benefits of copy-on-write [28], and the small costs of the Mach shootdown algorithm. Performance of a Unix-like fork operation will suffer greatly.
- Their second algorithm requires tagging all memory references with a generation counter. This increases processor-memory traffic by an unacceptable amount (25% or more); the authors acknowledge this drawback.
- Their third algorithm performs address translation at memory, and therefore requires that the memory cluster holding a page be determined directly from the virtual address after extension via a segment register. This makes copy-on-write impossible among memory modules (a bad decision), and severely reduces the flexibility of shared memory by requiring that it be implemented as shared segments (where the segment is determined by using the high-order bits of the virtual address to index into a segment table). Such hardware cannot fully support the shared memory functionality provided by Mach.

In summary, these algorithms do not solve the translation buffer consistency problem in its full generality, and impose large costs for small benefits. We do not believe this to be a productive design approach.

11 Conclusions

The Mach shootdown algorithm and its implementations demonstrate that translation buffer consistency can be implemented in software. The algorithm works reliably and is in production use on many multiprocessors at CMU and elsewhere. We were pleasantly surprised by the low overhead on current machines. The algorithm as presented here will scale badly to larger machines (e.g. 6ms basic shootdown time for 100 processors), but appropriate hardware support and system structures that match the hardware structures on such machines should be able to reduce this to acceptable levels. The two most desirable hardware support features for TLB consistency are a high-priority software interrupt and an MMU design that allows us to avoid stalling remote processors while pmaps are being updated. We conclude that translation buffer consistency overhead is not an obstacle to building multiprocessors with hundreds of processors.

References

[1] AT&T. *UNIX System V/386 Programmer's Reference Manual.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[2] R. Bisiani and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Trans. Comput.*, 37(8):930–945, August 1988.

[3] S. Bose, E. Clarke, D. Long, and S. Michaylov. *Parthenon: A Parallel Theorem Prover for Non-Horn Clauses.* Technical Report CMU-CS-88-137, Computer Science Department, Carnegie Mellon University, Pittsburgh,PA, 1988.

[4] W. Brantley, K. McAuliffe, and J. Weiss. RP3 Procesor-Memory Element. In *Proceedings of the International Conference on Parallel Processing*, pages 782–789, IEEE Computer Society, 1985.

[5] R. Case and A. Padegs. *Architecture of the IBM System/370*, chapter 51, pages 830–855. McGraw-Hill Book Company, New York, 1982.

[6] S. Chatterjee. Multiprocessor Cache Consistency, an annotated bibliography. To Appear.

[7] D. Cheriton, P. Boyle, and G. Slavenburg. Comments on 'Coherency for Multiprocessor Virtual Addressed Caches' by James R. Goodman in ASPLOS II, October 1987. *Computer Architecture News*, 16(3):3–6, June 1988.

[8] D. Cheriton, A. Gupta, P. Boyle, and H. Goosen. The VMP Multiprocessor: Initial Experience, Refinements, and Performance Evaluation. In *Conference Proceedings, The 15th Annual International Symposium on Computer Architecture*, pages 410–421, ACM-SIGARCH/IEEE Computer Society, Honolulu, HI, May/June 1988.

[9] D. Clark and J. Emer. Performance of the VAX 11/780 Translation Buffer: Simulation and Measurement. *ACM Transactions on Computer Systems*, 3(1):31–62, February 1985.

[10] E. Cooper and R. Draves. *C Threads*. Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1988. Programmer's manual for the Cthreads library.

[11] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance Measurements on a 128-node Butterfly Parallel Processor. In *Proceedings of the International Conference on Parallel Processing*, pages 531–540, IEEE Computer Society, 1985.

[12] *VAX Hardware Handbook*. Digital Equipment Corporation, Maynard, MA, 1982.

[13] Encore Computer Corporation. Multimax 320 Multiprocessor System. Data Sheet.

[14] R. Gingell, J. Moran, and W. Shannon. Virtual Memory Architecture in SunOS. In *Proceedings of the Summer 1987 USENIX Conference*, pages 81–94, USENIX Association, Phoenix, AZ, June 1987.

[15] *80386 Programmer's Reference Manual*. Intel Corporation, Santa Clara, CA, 1986.

[16] G. Kane. *MIPS R2000 RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[17] *MC88200 Users Manual*. Motorola, Inc., Austin, TX, 1988.

[18] *Series 32000 Databook*. National Semiconductor Corporation, Santa Clara, CA, 1986.

[19] G. Pfister, et. al. The IBM Research Parallel Processor Prototype: Introduction and Architecture. In *Proceedings of the International Conference on Parallel Processing*, pages 764–771, IEEE Computer Society, 1985.

[20] C. Polychronopoulos. Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design. *IEEE Trans. Comput.*, 37(8):991–1004, August 1988.

[21] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Trans. Comput.*, 37(8):896–908, August 1988.

[22] B. Rosenburg. Personal Communication. Member of the RP3 Group, IBM T. J. Watson Research Center.

[23] A. Spector, R. Pausch, and G. Bruell. Camelot: A Flexible Distributed Transaction Processing System. In *Proceedings of Spring Compcon 88*, pages 432–437, IEEE, San Francisco, CA, February/March 1988.

[24] A. Spector and K. Swedlow, eds. *Guide to the Camelot Distributed Transaction Facility*. Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 0.98(51)[aleph] edition, 1988.

[25] P. Teller, R. Kenner, and M. Snir. TLB Consistency on Highly Parallel Shared Memory Multiprocessors. In *Proceedings, 21st Annual Hawaii International Conference on System Sciences*, pages 184–192, IEEE Computer Society, Honolulu, HI, 1988.

[26] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young. Mach Threads and the UNIX Kernel: The Battle for Control. In *Proceedings of the Summer 1987 USENIX Conference*, pages 185–197, USENIX Association, Phoenix, AZ, June 1987.

[27] A. Tevanian, R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi. A UNIX Interface for Shared Memory and Mapped Files under Mach. In *Proceedings of the Summer 1987 USENIX Conference*, pages 53–68, USENIX Association, Phoenix, AZ, June 1987.

[28] A. Tevanian, Jr. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1987.

[29] M. Thompson, J. Barton, T. Jermoluk, and J. Wagner. Translation Lookaside Buffer Synchronization in a Multiprocessor System. In *Conference Proceedings, Winter 1988, USENIX Technical Conference*, pages 297–302, USENIX Association, Dallas, TX, February 1988.

[30] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 63–76, ACM-SIGOPS, Austin, TX, November 1987.