TLB Consistency on Highly-Parallel Shared-Memory Multiprocessors

by

Patricia J. Teller
Richard Kenner
Marc Snir†

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
715 Broadway, 10th Floor
New York, NY 10003

Ultracomputer Note #129
October, 1987

The first two authors were supported in part by the Applied Mathematical Sciences Program of the U.S. Department of Energy under contract DE-AC02-76ER03077, in part by the National Science Foundation under grant DCR-8413359, and in part by I.B.M. under joint study agreement N00039-84-R-0605(Q).

† **Current Address:** IBM T. J Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

ABSTRACT

Multiprocessors that store the same shared data in different private caches must ensure these caches have consistent copies. Almost all known solutions to this *cache consistency* problem are only suitable for architectures with a few tens of processors (PEs). Efficient solutions to the *TLB (translation lookaside buffer) consistency problem*, a special case of the cache consistency problem, can be found for *highly-parallel, shared-memory multiprocessors* (HPSMMs) with many hundreds of PEs for the following reasons: the number of references to address translation information per modification is very large; the cache for storing translation information can be present anywhere on the path from the PEs to memory; when the memory mapping needs to be modified, one can often select which translation information to change; and obsolete mapping information can be used until permanent changes must be made. We present three general methods that exploit these features and can be used on HPSMMs to maintain TLB consistency. Tradeoffs are discussed and are related to overall system performance. Some interesting issues inherent to the TLB consistency problem and the support of a demand-paged virtual memory system on a HPSMM are also presented. Our methods demonstrate the use of alternative implementations of the classical readers-writers algorithm.

1. Introduction

To match the relatively high speed of the CPU with the relatively low speed of a large, modest-cost memory, most computer systems use a *hierarchical memory*, where higher levels are faster but smaller. By exploiting the *principle of locality* [4], the average delay for memory access is little more than that of the fastest level. Most parallel programs exhibit locality of reference to shared code and many exhibit significant locality of reference to shared data. Therefore, a hierarchical memory organization is attractive for the shared memory of multiprocessors.

Many applications, on multiprocessors as well as uniprocessors, require a virtual address space much larger than physical memory. To support these applications, an efficient implementation of demand paging is needed. Such an implementation conventionally uses a TLB *(translation lookaside buffer)*, a cache for address translation information, to speed up virtual memory access.

Demand paging is already supported on large supercomputers such as the CDC Cyber 205 [3], the Fujitsu VP-200 and the Hitachi S810/20 [13], and on commercially available, bus-connected multiprocessor systems such as the Elxsi System 6400 [18] and Sequent's Balance 8000 and 21000 [19]. We are interested in issues relevant to the support of a demand-paged virtual memory on highly-parallel, shared-memory multiprocessor (HPSMM) systems with potentially hundreds of processors [24], e.g., the NYU Ultracomputer [5], the IBM RP3 [14], and the BBN Butterfly [17]. One of these issues is the design and management of TLBs. On such systems, cache consistency problems, including the special case of TLB consistency, are much more difficult to solve than on machines with fewer processors.

In contrast to the TLB consistency problem, the design and performance of general-purpose caches in multiprocessors have received considerable attention. It seems intrinsically difficult to solve the general problem of ensuring the consistency of multiple copies of the same shared data in different private

caches (i.e., maintaining *cache consistency* or *cache coherence*) without imposing severe restrictions on the caching of shared data. Known hardware solutions ([1], [21], [26], and references therein) require intercache communication rates that grow sufficiently fast with the number of processors, making them only suitable for architectures with a few tens of processors. The efficiency of several of these solutions is examined in [1], where, based on simulation results, the authors indicate to what extent each may limit the number of processors in the system.

In this paper we show that efficient solutions to the TLB consistency problem on HPSMM systems exist. After first defining the assumed computing environment, we explore the problem of TLB consistency, contrast it to the general cache consistency problem, show why efficient solutions to the TLB consistency problem are possible, and present metrics that can be used to evaluate them. Three general TLB consistency techniques are then described and tradeoffs between them are discussed and related to overall system performance.

2. The Problem

We first present the models used for the HPSMM, including those for the physical and virtual memory and page-fault handling. With these models defined, we explore the general problem of maintaining the consistency of translation information on such an architecture and demonstrate why a TLB is even more important than on a uniprocessor.

2.1. The Physical Model

As shown in Figure 1, a *highly-parallel, shared-memory multiprocessor* (HPSMM) is composed of many *processing elements* (PEs), typically hundreds, that access a global main memory via an interconnection network. Main memory is partitioned into *memory modules* (MMs), usually one per PE, that are grouped into *memory clusters* (MCs). A *cluster controller* (CC), associated with each MC, has access to all the MMs in its cluster. It is used as an I/O controller and in some cases is capable of executing code and communicating with all PEs in the system.

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 MC and is usually interleaved across all the MMs in the cluster. (On the RP3, some frames, called ‘‘local’’ frames, are contained entirely within a single MM.) Consecutive frames are interleaved across the MCs. These interleavings map a *physical address*, composed of a frame number and displacement, to a memory location. The frame number specifies an MC and a unique frame within the MC; the displacement determines a particular location within that frame. Since each frame is interleaved identically across an MC, the displacement uniquely defines the MM containing a specific location. This mapping is depicted in the lower part of Figure 2; the upper part of the figure illustrates the logical-to-physical address translation which is described below.

In order to reduce the cost of context switching between related processes, a global (i.e., system-wide) *segmented virtual address space* is desirable on a HPSMM. 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

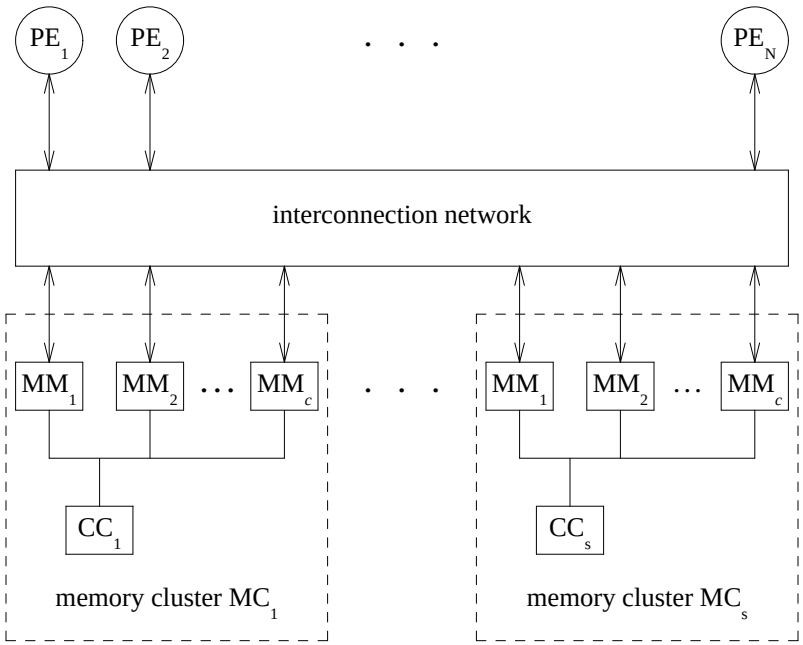

Figure 1: HPSMM Architecture

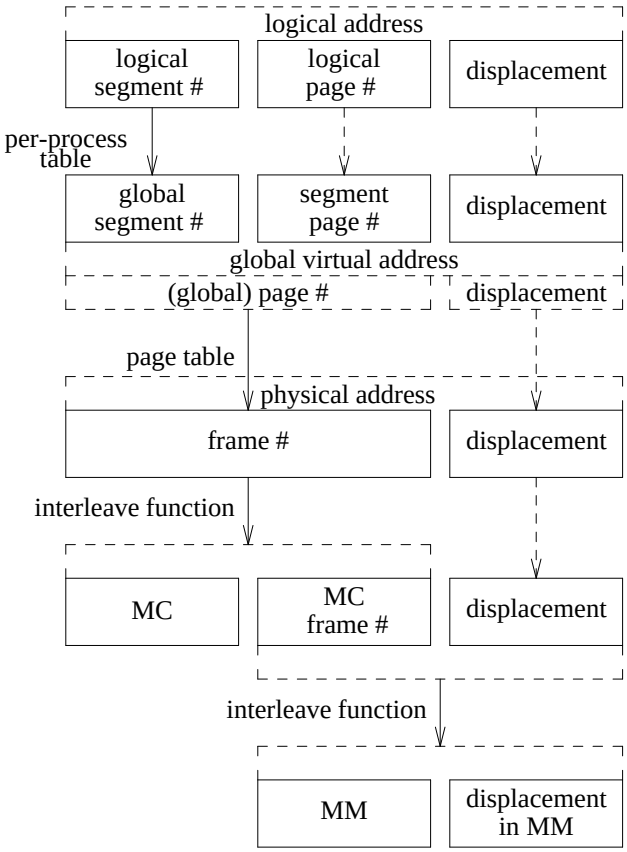

Figure 2: Address Translation

address space is composed of global segments. Each *global segment* refers to a unique object in the system[1] 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*, *segment page number*, and *displacement*. Each process can access a subset of the global segments in the system.

Having a global virtual address space means that no two processes running concurrently may use the same virtual address to refer to different objects. This implies that the global virtual address of any object

[1]However, two pages in different segments (and hence having different virtual addresses—''aliases'') may be mapped to the same frame in order to efficiently implement ''copy-on-write''. When this is done, the pages are marked read-only and thus do not create cache consistency problems.

cannot be known until run-time. To permit the 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*, *logical 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 PE 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 paper, we ignore the fact that the global 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 that page.

A page can have a frame in memory allocated to it (a *resident* page) or only be stored on disk (a *nonresident* page). Pages can be dynamically relocated during program execution. The location of pages is contained in a *page table*, an array indexed by page number. Each *page table entry* (PTE) contains the frame number of a resident page or disk address of a nonresident page, a dirty bit, the page status, and protection information. If a page is not resident, a memory reference to the page causes a *page fault* which results in the referenced page being made available (see Section 2.2).

A *page frame table* (PFT) 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. In two of the TLB consistency techniques presented in this paper, the PFT is distributed so that parts of each entry are replicated at multiple MMs.

We define a *paging arena* as a region of memory to which the paging algorithm is independently applied. Each MC can be made a separate paging arena by restricting which frames can be used to store a specific page, with the page number uniquely determining the MC in which it can be stored. This, along with the global nature of memory interleaving, allows each virtual address to uniquely specify the MM in which it must reside. Therefore, page replacement can be done locally within an MC, possibly by the CC, and the page table can be fragmented, with each cluster only storing entries for pages that can be contained in it. The technique of Section 3.5 requires separate paging arenas.

Our HPSMM model also provides *fetch-and-*ϕ operations [7], atomic read-modify-write operations, referred to individually as fetch-and-ϕ(**m**, **v**), where **m** is a memory location and **v** is some value. They return the original contents of **m** to the PE and simultaneously store 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 can be performed as a fetch-and-add of zero. If the previous value is returned when executing a store, one obtains a *fetch-and-store* operation which swaps the old value in memory with the new value sent by the PE. The following fetch-and-ϕ operations are defined by various HPSMM 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.

2.2. Page-Fault Handling

In this section, we present a model of classical uniprocessor demand-paging implementations and contrast it to multiprocessor implementations.

2.2.1. General Model

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 available. 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 disk 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 to during its 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.

In most of the discussions below, 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 that page, it becomes the *pager* for that page and takes whatever actions are necessary to make that page available. This may include evicting a resident page, writing a dirty page to disk, and reading the requested page. If the faulted page is already being obtained by 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.

2.2.2. Multiprocessor Issues

On 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 PEs 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, much of the serialization otherwise required by these locks can be eliminated [23].

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 looking for 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. 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 [2], an approximation to the LRU policy, can be modified using the fetch-and-add operation so that each concurrent evicter is given a unique page to evict [24]. When there is more than one distinct paging arena, it may be best to have one page daemon per arena, each using an optimal eviction choice strategy.

2.3. Consistency of Translation Information

If multiple copies of translation information exist, we must ensure that they are kept consistent. In this section we discuss why these multiple copies arise, how they are protected, and when address translation modifications are made.

2.3.1. Address Translation Caches

The standard method of speeding-up virtual memory access is to cache address translation information. If a physical-address cache is used by the PE, a special-purpose device (called a *translation look-aside buffer*, TLB) is used to cache translation information [20]. If a virtual-address cache is used, translation information can be stored in it [11]; we consider the part of this cache used to store translation information as a separate TLB. By taking advantage of the principle of locality, these caches substantially reduce the average number of memory accesses required to satisfy a virtual memory request.

With a TLB, only one memory access per virtual memory request is required when the translation for the referenced page is present in the TLB (a *TLB hit*). When there is no entry in the TLB for the referenced page, a *TLB miss* occurs, and the translation information for the page is copied from the PTE into the TLB (a *TLB reload*).

2.3.2. Copies of PTEs

In the absence of an address translation cache and assuming a single-level page table in main memory, a uniprocessor requires two memory accesses to satisfy a virtual memory request for uncached data—one to access the address translation information and the other to access the data itself; a HPSMM incurs a much higher cost. Not only is access to shared main memory usually the performance bottleneck on a HPSMM, but we will show that extra memory accesses are required to obtain up-to-date address translation information. Moreover, if the translation information for a shared page is not replicated, it may become a ‘‘hot spot’’ of memory activity [15] unless the ‘‘combining’’ network of [9] is used.

Shared address translation information is always writable even if the mapped page is read-only, since it is modified whenever a page is moved. Therefore, we must ensure that if multiple copies of address translation information exist (e.g., in TLBs at multiple PEs), they are kept consistent with the

page table. This is the issue of *TLB consistency*.

The general problem of maintaining the consistency of address translation information is not unique to systems with multiple processors, TLBs, or demand paging since a copy of a PTE always exists, either explicitly or implicitly. On a uniprocessor with a TLB, the software must ensure that the TLB copy is kept consistent with the PTE by removing entries from the TLB (*invalidating TLB entries*) that contain translation information of modified PTEs.

On a multiprocessor system without TLBs, between the time a PE reads the address translation information and the time it accesses the data in the referenced page, there is an implicit copy of the PTE in the PE. Due to the presence of the processor-to-memory interconnection network, we must make the very pessimistic assumption that an arbitrary number of memory accesses by other PEs may occur between these two accesses. Therefore, another PE may evict the referenced page and reclaim the page's frame for use by another process and write into it during this time interval.[2]

2.3.3. Locking Translation Information

This is 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 moved, no reader is allowed to observe an intermediate state where the page has moved but its translation information has not changed (or vice versa). One way this might happen is if a reader uses an obsolete copy of a PTE.

A classical solution to these kinds of problems is the use of *readers-writers locks*. When a process needs to obtain a consistent view of a data structure but will not modify it, the process requests a *read lock*. An arbitrary number of read locks may be granted as long as no process is concurrently modifying the data. When a process desires to modify the data, it requests a *write lock*. Only one write lock is granted at a time and only when no read locks are active.

When these locks are used, a reference to resident virtual memory formally consists of four steps:

- obtain a read lock on the PTE;
- read the address translation information from the PTE;

[2] On an NYU Ultracomputer or an IBM RP3, for example, typical access time to shared memory is on the order of one microsecond, but worst-case access time can be in the millisecond range. Although situations such as the one mentioned are highly improbable, they must be considered to ensure the correctness of our algorithms.

- access the requested memory; and
- release the read lock on the PTE.

(To modify a PTE, a PE must hold a write lock on it.)

On a uniprocessor, these locks are implicit: no actions can occur between an access to the translation information and the subsequent access to the page itself.[3] On a shared-memory multiprocessor these locks must be explicit. In the absence of a TLB, four memory operations are required to satisfy a virtual memory request. (The first two can be merged into one with the use of special hardware [10]). This contrasts with two on a uniprocessor, making a TLB even more desirable on a multiprocessor system.

Unfortunately, on a multiprocessor, one must do more than merely add a TLB at each PE. Locks on a PTE must enforce consistency of the multiple PTE copies resident in the TLBs. A PTE is only guaranteed not to change while a read lock is in force. If a read lock on a PTE is not maintained during its entire TLB residency, but rather is only held from the start of a TLB reload until the reference that caused the reload has been satisfied, that TLB entry must be assumed to be stale for the next reference. This requires a reload of the TLB for each virtual memory request and, thus, clearly defeats the purpose of having a TLB.

2.3.4. Safe vs. Unsafe Changes

Modifications to a PTE fall into two categories: safe and unsafe. Safe changes are those where, in the natural course of execution, obsolete translation information will be detected and corrected. For example, if the status of a page is changed from nonresident to resident, use of information indicating that the page is nonresident results in a page fault. The page-fault handler can consult the page table and recognize that the page is actually resident and simply reload the TLB. A similar process occurs when the protection information of a page is changed from read-only to read-write. In contrast, the remaining possible modifications to a PTE are unsafe: altering a page's status from resident to nonresident and changing a page's protection from read-write to read-only. In the absence of locking, unsafe changes can result in erroneous program execution. The techniques presented in Section 3 can be used to ensure TLB consistency when unsafe changes are required.

The status of a page can change from resident to nonresident when the page is selected for eviction or when the virtual memory containing the page is deallocated. Evictions are by far the most frequent case and are dealt with explicitly by each of our TLB consistency techniques. We can sidestep the problem when deallocating virtual memory by doing so solely on a segment basis and by only reusing a global segment number when no TLB contains an entry for a page in that segment, as discussed in Section 3.3.

---

[3] There is, however, a pathological possibility of the opposite happening. Namely, on a system with a TLB, it might be possible to access a page in the process of updating the translation for that page. This does not happen in practice because context switching is normally disabled during execution of paging code and the paging code is locked in memory.

The only situation we consider where a page's protection can change from read-write to read-only is when implementing the *copy-on-write* optimization. When copies of a segment must be made, e.g., to implement the *fork* system call, the copy-on-write optimization defers duplicating each page until it is written. This is done by creating a new segment, mapping all of its pages to the frames used in the original segment, setting the protection for these pages to read-only in both segments, and incrementing a reference count. When a process writes to a page whose reference count is greater than one, a private, read-write copy of the page is made. In the common case, where most of the pages of the segment are never written, this saves unnecessary duplication.

3. The Solutions

Unlike the general problem of cache consistency, the consistency problem with respect to shared address translation information is sufficiently specialized so that it admits efficient solutions. This is because:

(1) The number of references to address translation information per modification is very large; a reference occurs for each virtual memory access, while a modification only occurs when a page's status or protection is changed.

(2) Address translation information can be present anywhere on the path from a PE to an MM.

(3) Whenever 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'' [22] techniques can use obsolete address mapping information until permanent changes must be made.

After presenting two data-locking mechanisms, we present a list of metrics and a discussion of general design and performance issues that are useful in evaluating the proposed consistency techniques. We then show how the above features can be exploited to help us approach the performance advantages of TLBs enjoyed by uniprocessors—one main memory access per virtual memory request on a TLB hit.

The approach discussed in Section 3.3 maintains a read lock on a PTE during the entire time that its data is present in a TLB and restricts modifications to translation information of pages whose PTEs are read-locked (e.g., a page that has an entry in any TLB is not considered a candidate for eviction). In Section 3.4, we present a method that capitalizes on ''optimistic recovery''—it avoids the use of explicit locks by allowing TLB data to become stale and being able to detect the use of stale data. The third method, described in Section 3.5, eliminates the need for a TLB at each PE by sending virtual, rather than physical, addresses through the network and performing address translation at the memory.

3.1. Data-Locking Mechanisms

Classical implementations of readers-writers locks suffer from the fact that readers can busy-wait even in the absence of writers. To avoid this, we present two algorithms that employ the fetch-and-add

operation (see [8] for details) and are used in the implementation of our TLB consistency techniques.

3.1.1. 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 [8]. This counter, **C**, is equal to $\mathbf{r} - (\mathbf{n} \times \mathbf{w})$, 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;
```

Note that, in the absence of writers, no serialization occurs in these algorithms. One might think the first test in these algorithms is ‘‘redundant’’, however, as explained in [8] it is necessary to prevent deadlock. (Priority-based and starvation-free variants of these algorithms are also discussed in [8].)

3.1.2. Two-Counter Algorithm

Instead of implementing the readers-writers algorithm, this technique enables a reader 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 := C1;
    make a private copy of the data;
    MyC2 := C2;
 until MyC1 = MyC2;
 use the private copy of the data;
end;
```

A prospective writer does the following:

```
begin
  fetch-and-add(C2, 1);
  update the data;
  fetch-and-add(C1, 1);
end;
```

Because a writer increments C_2 first and increments C_1 last and a reader reads C_1 first and reads C_2 last, 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.)

3.2. Design and Performance Issues

In order to assess the performance of our TLB consistency techniques, we must consider the following metrics, each of which affects system performance:

- the time to access information mapped by an entry in the TLB;
- the probability of a TLB hit;
- the delay due to a TLB miss (TLB reload time);
- the page-fault rate;
- the time to service a page fault; and
- the memory required for address translation information.

Each of these metrics represents a parameter relevant to the overall performance of the computer system. However, the relative importance of each of these parameters is not always clear and sometimes depends on both the specific values of the parameters and detailed engineering issues. Our TLB consistency techniques occasionally improve one parameter while worsening another. In the description of each method, we discuss its performance as measured by the above metrics. Usually only a subset of the

metrics are significant. For example, the service time for a page fault is not greatly affected by any of these methods. On the other hand, the TLB reload time differs among the various techniques but cannot be quantitatively compared without detailed engineering information.

As discussed in Section 2.1, we sometimes impose the constraint of always mapping a page to the same MC. This will not always significantly reduce performance. 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 MC, which will run into the thousands. If hashing is used to associate pages to MCs, the distribution of active pages over the MCs can be expected to be very even. However, in drawing this analogy the following differences should be noted:

(1) 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.

(2) The shared memory acts as a ‘‘page cache’’ for all PEs; 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 PEs.

3.3. Maintaining the Read Lock

The most straightforward way of ensuring the consistency of TLB data with the shared page table is for each PE to maintain a read lock on a PTE during the entire time the translation information is present in its TLB. This makes the number of active read locks for a PTE equal to the number of TLBs containing that PTE. A PE only releases its read lock when the entry is removed from its TLB. Since a write lock must be held while modifying a PTE, unsafe modifications are only allowed to PTEs that are not in any TLB.

3.3.1. Basic Technique

To perform a TLB reload, a read lock is obtained on the PTE and the translation information is copied into a TLB entry. Once the translation information is in the TLB, references to the page result in TLB hits and, as on a uniprocessor, we attain one main memory access per virtual memory request. If the referenced page is not resident, a TLB reload results in a page fault and the page-fault handling described in Section 2.2 is performed. When the page is available, the TLB reload is retried. Prior to the invalidation of a TLB entry, the TLB dirty bit must be examined and, if set, the dirty bit of the PTE must be set; then the read lock is released.

To implement the required locks without unnecessary serialization, we use the one-counter algorithm presented in Section 3.1.1 to protect each PTE. The counter used by that algorithm indicates the number of PEs that currently hold a read lock on that PTE and, hence, the number of TLBs containing a

copy of that PTE. We call this counter the *replication count* (RC).[4] For pages that are to be permanently locked in memory (system pages), a read lock that is never released can be obtained at boot time.

3.3.2. Avoiding Write Locks

To avoid trying to obtain a write lock on a PTE with active read locks, we must eliminate the need to make unsafe modifications to TLB-resident PTEs. As discussed in Section 2.3.4, unsafe changes are caused by deallocating virtual memory, changing a page's protection from read-write to read-only (while performing the copy-on-write optimization), and evicting a page.

A TLB can contain an entry for a page in a deallocated segment as long as we can ensure that it will not be used. We do this by not reusing the segment number of a deallocated segment while any TLB contains an entry for a page in that segment. This is done by associating a counter (distinct from the counter used by the operating system to denote the number of processes sharing the segment) with each global segment number; this counter is incremented on every associated TLB reload and decremented on each invalidation.

The two remaining unsafe modifications occur when performing the copy-on-write optimization and evicting a page. We must forgo the copy-on-write optimization for any page that is resident in a TLB since otherwise we would need to set the page's protection to read-only. In the case of eviction, the page replacement policy must only consider frames containing pages with an RC of zero as candidates for eviction.

3.3.3. Deadlock Avoidance

We must guarantee that a page is always available for eviction (i.e., not in any TLB). This is only ensured if the number of frames is greater than the total number of TLB entries at all PEs. Otherwise, a deadlock can occur. If each MC defines a separate paging arena, a deadlock can occur if the number of frames in each MC is less than the total number of TLB entries.

Provided that this condition will seldom occur, we can drop the guarantee of there always being a page that is not in any TLB. When deadlock does occur, an inefficient, global mechanism must be used: after selecting a page for removal, a message is broadcast to all PEs to invalidate entries for the chosen page.[5] Upon receipt of this message, each PE waits for the completion of its outstanding references to the page and then invalidates any TLB entries it has for that page. The page selected for eviction cannot be removed from memory until all the outstanding read locks on its PTE have been released. Alternatively,

[4] In addition to representing the read-write lock state of a PTE, 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 can be evicted and evict it if so (see [23] for details).

[5] This is essentially one of the methods used in the MACH operating system [16]. Rather than restricting unsafe modifications, they rely solely on a global mechanism. The MACH operating system sometimes postpones the use of a changed mapping until all PEs have had a chance to flush their TLBs (e.g., at a timer interrupt) and when acceptable, e.g., for safe changes, they allow temporary inconsistency. This is inefficient for systems with many PEs.

if the TLBs implement an LRU replacement policy, one might use an election scheme: a ‘‘deadlocked’’ message can be broadcast to all PEs which then elect a page to evict by identifying the oldest entry in their TLBs.

3.3.4. Evaluation

We feel that our restriction on which pages to evict is appropriate. Page replacement policies base their choice of a page to evict on recent activity; a page that has not been accessed recently is a good candidate for eviction. The presence of copies of the PTE for a page in TLBs is one measure of activity for a page—if no TLB contains an entry for the page, the page is a good candidate for eviction. However, care must be taken to ensure that the TLBs of idle PEs get flushed periodically.

No additional memory is required for the address translation structures used to implement this method; the RC field of the PTE would be required by any other locking mechanism. Also, this method keeps the network load low because only the minimum required information (the physical address) is transmitted from the PEs to the MMs.

On the other hand, there are some disadvantages to this method. It relates the size of the TLBs to the size of main memory. If an MC is considered a separate paging arena, in order to avoid deadlock we may be forced to use smaller TLBs than are otherwise desirable; this will increase the TLB miss rate. Also, since the copy-on-write optimization cannot be fully utilized, a cost of this method is the possible unnecessary duplication of pages when segments are copied.

3.4. Validation

Rather than ensuring that TLB information is always valid, we can allow TLB entries to become stale and detect accesses that use stale TLB entries. Similar to the approaches used in [6] and [21], we associate a *generation count* (GC) with each frame of physical memory; the current GC assigned to a frame acts as a version number or time stamp. We use readers-writers locks to ensure that the translation information and its GC are updated atomically. Unlike the locks of the previous section, these locks need only be held momentarily, and not during the entire TLB residency of the PTE data.

A PE augments each memory request with the GC of the related entry in its TLB. Using this GC, the ‘‘validity’’ of the translation is atomically verified by the MM. When a stale entry is detected, the request fails and a signal is sent to initiate the refreshing of the stale TLB information.

3.4.1. Basic Technique

Located at each MM is a *generation count table* (GCT), accessible and modifiable by the CC associated with the MM as well as by the PEs. It contains an entry for each frame in the cluster. Each entry consists of a valid bit, a GC, protection information, and a dirty bit. On writes, the MM sets the GCT entry (GCTE) dirty bit. When a page is to be evicted, the dirty bits of all GCTs in the cluster are consulted.

A TLB reload is done by obtaining a read lock on the PTE, copying the translation information from it, reading the GC from the GCTE, and releasing the read lock. A process must obtain a write lock on a PTE when it desires to make an unsafe modification to the PTE. Before releasing the lock, it updates the GC in all GCTs that have an entry for the frame containing the page.

3.4.2. Locking the Frame

Although we associate a read-write lock with each page, it is actually more efficient to associate it with each frame. Less storage space is required because there are more PTEs than PFT entries (PFTEs). Also, in the presence of aliases, there may be more than one PTE per frame; if a read-write lock is associated with each page, a PE would have to obtain multiple write locks in order to modify the mapping of a frame.

It is possible to associate the read-write lock with each frame by using the following method: A TLB reload first consults the PTE for a frame number and obtains a read lock on that frame. The frame number is read again from the PTE, this time while the read lock is in force, and is compared with the previously read frame number. If they differ, the read lock is released and the process is repeated. Otherwise, the GC is read from the GCTE, the read lock is released, and the TLB reload is completed. Write locks are obtained similarly. This method of obtaining locks on a frame, rather than a page, can also be used to implement the technique of Section 3.3.

Using the two-counter algorithm of Section 3.1.2, 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 GCTEs and the GC in the related PFTE (a field added to each PFT entry). The GC in the GCTE, corresponding to C_2 in the algorithm, is used, as before, by the MM to validate incoming requests. However, a TLB reload copies the GC from the PFTE, corresponding to C_1, instead of the GCTE. The GC of the PFTE and all the GCTEs agree except while translation information is being modified.

A PE that desires to make unsafe modifications to a PTE executes the ‘‘writer’’ portion of the algorithm of Section 3.1.2 by first updating the GCs in the GCTEs (C_2), then modifying the PTE, and finally updating the GC in the PFTE (C_1).

A TLB reload obtains a trial frame number from the PTE, the GC from the PFTE (C_1), and then the actual frame number is read again from the PTE. 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 of the GCTE (C_2). This completes the ‘‘reader’’ portion of the algorithm of Section 3.1.2.

3.4.3. GC Overflow

This scheme only works correctly if ‘‘regenerations’’ are prevented—each TLB entry has to be reloaded with current translation information (refreshed) before a GC has cycled through all its values; thus long GC fields seem better. On the other hand, the GC is transmitted from the PEs to the MMs. According to [12] asymptotic network delay grows quadratically with the length of the message.

Therefore, keeping the GC short is required to avoid a significant effect on network traffic. The optimal GC length is determined by a compromise between these two competing goals.

The refresh rate is also affected by the manner in which the GC is updated. A straightforward implementation is to simply increment a GC at each PTE unsafe update. On overflow, we broadcast a message to all PEs that causes them to refresh their TLBs. The probability of a GC increment requiring a TLB refresh is the reciprocal of the number of possible GC values. It may be possible to reduce the frequency of broadcasts by determining the average time between two overflows of a GC and periodically refreshing TLBs more frequently than this. If the GC is long and successive updates of a GC are relatively infrequent, the overhead of TLB refreshes will be reduced to tolerable levels.

A method of supporting short GCs is to maintain a set of counters (each log **N** bits long, where **N** is the number of PEs in the system) for each frame, one per allowable GC value. Each counter denotes the number of TLBs that contain the corresponding GC for that frame. A counter is incremented on a TLB reload and decremented when a TLB entry is invalidated. When a GC is to be updated, a search is made for a GC that is not in use; i.e., for a counter with a value of zero. A refresh is only needed when each allowable value is present in at least one TLB. Periodically refreshing TLBs of idle PEs reduces the likelihood of this occurring.

3.4.4. Evaluation

The validation approach allows us to attain one main memory reference per virtual memory request on a TLB hit. However, network traffic is increased since each memory request is augmented by a GC. Also, the need for each MM to perform additional operations to process each memory request has the potential of decreasing the memory's service unless the operations are pipelined. This, in turn, increases network congestion.

Another disadvantage of this technique is that additional memory is required at each MM for the GCTs. If the counter scheme is used, yet more memory is needed.

3.5. Translation at Memory

Here, we require each memory cluster to be a distinct paging arena. In this case, there is no need to translate the virtual address at the PE—the virtual address determines the MM, where address translation can be performed. TLBs at the PEs become superfluous.

3.5.1. Basic Technique

Each MM has a copy of the PTEs of pages resident in its cluster. This copy can be accessed associatively by page number or directly by frame number and is called a *resident page table* (RPT); each RPT entry (RPTE) has a dirty bit.

A memory request contains its virtual, rather than physical, address. It is routed to the appropriate MM where the RPT is searched for an entry for the referenced page. If the search is successful, the access is performed and a reply is returned to the PE; otherwise, an exception occurs and the PE executing the

faulting process is interrupted. The page exception routine is executed by either the PE or the CC. In this scheme, read locks are implicit in the atomicity of the translation and data access that are done by the MMs.

3.5.2. The RPT

This method requires each RPT to be an associative memory with a number of entries equal to the number of frames in a cluster. Since there may be several thousand such frames, the construction of fast associative memories of this size may be beyond what current technology can achieve at a reasonable cost. However, there are several ways to minimize this problem.

We can reduce the number of frames that can contain a page, beyond the restriction of associating a page with a specific paging arena, so that the associative search for a page involves significantly fewer entries. This will reduce the degree of associativity (set-size) required. With a pseudo-random allocation of pages to frames it might be possible to achieve reasonable performance with a small set size, albeit a larger set size than the two to four that gives good performance in general-purpose caches.

Another alternative is to use a slower and cheaper mechanism to store the RPT, such as the memory itself, and provide a high-speed cache that stores the most recently used entries. This is effectively a TLB at the memory. If the TLBs at the MMs have a high hit rate, one can dispense with the RPTs altogether—when a TLB miss occurs the relevant PTE is loaded from the cluster page table. For shared frames that are interleaved across all MMs of a cluster, it is reasonable to expect a high correlation between accesses to the same frame at distinct MMs. This justifies the use of identical TLB entries at all MMs for these frames (the value of the dirty bit of an entry may differ at each RPT). In such a setting, the TLB replacement decisions can be the responsibility of the CC—the same entry is invalidated in all the TLBs of the cluster; the dirty bits are ‘‘ORed’’ and the result stored in the cluster page table; and the new entry is stored in all the TLBs.

3.5.3. Evaluation

With address translation at the memory, an additional delay in memory access time may be experienced. This additional delay need not occur if a technique, due to Sun Microsystems, Inc. [25], which takes advantage of the characteristics of *dynamic random access memories* (DRAMs), is implemented. A DRAM is accessed by first selecting a row with the low-order bits of an address and then by selecting a column with the high-order bits. If a successful address translation can be performed while the memory is using the displacement as the row address, the DRAM can be presented with the frame number as the column address. This overlaps translation with memory access.

Performing translation at the MMs requires larger message sizes since the virtual, rather than the physical, address is transmitted with each memory request. This implies either wider paths or longer messages between the PEs and shared memory. However, there are no significant differences in hardware complexity when compared with performing address translation at the PE. In addition, the reload time for TLBs at memory is shorter than the reload time for TLBs located at the PEs and no network traffic is

required to perform the reload.

As discussed in Section 3.2, dividing memory into distinct paging arenas may have an adverse impact on the page-fault rate. This will be even more pronounced if further division of paging arenas is done to reduce the associativity requirements of the RPT. Also, as in the validation scheme, additional memory is required at each MM.

Assuming a fixed-sized TLB, we compare the relative TLB hit rate achieved with TLBs at memory to that of TLBs at the PEs: Let a page be concurrently accessed by p processors and be interleaved across the c memory modules in a cluster. How many TLB entries are necessary to guarantee that no TLB miss occurs for this page? If the TLBs are located at the PEs, this requires p TLB entries; if the TLBs are located at the MMs, c entries are needed. The hit rates of both organizations are equal when p is equal to c. However, other, more basic, considerations of efficiency affect the balance of p and c: having c much lower than p creates memory contention; on the other hand, choosing too large a c reduces available I/O bandwidth. These considerations thus influence the optimal TLB location.

4. Conclusions

In this paper, we have presented three different mechanisms that can be used to maintain the consistency of TLBs in highly-parallel, shared-memory multiprocessors. Since the use of global communications to enforce TLB consistency is only likely to be a reasonable option for machines with a few tens of processors, designers of highly-parallel architectures with many hundreds of PEs must choose among techniques similar to those described above.

The qualitative arguments presented with each method are not intended to settle the issue; extensive simulations, with different types of codes and varying assumptions on the parameters of the computer architecture, are needed in order to make an intelligent choice. These simulations are currently being performed and their results will be published in the near future. It is likely that the ideal approach will be some combination of the techniques presented in this paper.

We also have shown that standard replacement policies may not be efficient in the presence of multiple pagers (or page daemons). The choice of a good page replacement policy is another topic requiring further research.

Acknowledgements

We would like to thank Allan Gottlieb for his helpful comments.

References

[1] James Archibald and Jean-Loup Baer, ‘‘Cache Coherence Protocols: Evaluation Using a

Multiprocessor Simulation Model'', *ACM Transactions on Computer Systems*, Vol. 4, No. 4, Nov. 1986, pp. 273-298.

[2] L. A. Belady, ''A Study of Replacement Algorithms for a Virtual Storage Computer'', *IBM Systems Journal*, 5, 2, 1966, pp. 78-101.

[3] *CDC Cyber 200 Model 205 Computer System Hardware Reference Manual*, Revision 01, A, 1980, 1981.

[4] Peter J. Denning, ''On modeling program behavior'', *Proceedings of the Spring Joint Computer Conference*, 1972, pp. 937-944.

[5] J. Edler, A. Gottlieb, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. J. Teller, and J. Wilson, ''Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach'', *The 12th Annual International Symposium on Computer Architecture*, June 1985, pp. 126-135.

[6] David R. Emberson, ''A Cache Coherence Management Technique for Hypercube Multiprocessors'', *Proceedings of the 1987 International Conference on Parallel Processing*, Aug. 1987, pp. 262-265.

[7] Allan Gottlieb and Clyde P. Kruskal, ''Coordinating Parallel Processors: A Partial Unification'', Ultracomputer Note No. 34, Courant Institute, NYU, Sept. 1981.

[8] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph, ''Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors'', *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 2, Apr. 1983, pp. 164-189.

[9] Allan Gottlieb, ''An Overview of the NYU Ultracomputer Project'', Ultracomputer Note No. 100, Courant Institute, NYU, July 1986.

[10] Malcolm C. Harrison, ''The Add-and-Lambda Operation: An Extension of F&A'', Ultracomputer Note No. 104, Courant Institute, NYU, July 1986.

[11] Mark Hill et al., ''Design Decisions in SPUR'', *Computer*, Vol. 19, No. 11, Nov. 1986, pp. 8-22.

[12] Clyde Kruskal and Marc Snir, ''The Performance of Multistage Interconnection Networks for Multiprocessors'', *IEEE Transactions on Computers*, Vol. C-32, No. 12, Dec. 1983, pp. 1091-1098.

[13] Olaf Lubeck, James Moore, and Raul Mendez, ''A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2'', *Computer*, Dec. 1985, Vol. 18, No. 12, pp. 10-24.

[14] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, ‘‘The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture’’, *Proceedings of the 1985 International Conference on Parallel Processing*, Aug. 1985, pp. 764-771.

[15] G. F. Pfister and V. A. Norton, ‘‘‘Hot Spot’ Contention and Combining in Multistage Interconnection Networks,’’ *IEEE Transactions on Computers*, Vol. C-34, No. 10, Oct. 1985, pp. 943-948.

[16] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky, and Jonathan 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 (ASPLOS-II)*, Oct. 1987.

[17] Randall Rettberg and Robert Thomas, ‘‘Contention is No Obstacle to Shared-Memory Multiprocessing’’, *Communications of the ACM*, Vol. 29, No. 12, Dec. 1986, pp. 1202-1212.

[18] John Sanguinetti, ‘‘Performance of a Message-Based Multiprocessor’’, *Computer*, Vol. 19, No. 9, Sept. 1986, pp. 47-55.

[19] *Balance Technical Summary*, MAN-0110-00, Sequent Computer Systems, Inc., Nov. 1986, pp. B-1—B-9.

[20] Alan Jay Smith, ‘‘Cache Memories’’, *Computing Surveys*, Vol. 14, No. 3, Sept. 1982, pp. 473-530.

[21] Alan Jay Smith, ‘‘CPU Cache Consistency with Software Support and Using ‘One Time Identifiers’’’, *Proceedings of the Pacific Computer Communications Symposium*, Oct. 1985, pp. 142-150.

[22] Robert E. Strom and Shaula Yemini, ‘‘Optimistic Recovery in Distributed Systems’’, *ACM Transactions on Computer Systems*, Vol. 3, No. 3, Apr. 1985, pp. 204-226.

[23] Patricia J. Teller, ‘‘Algorithms for Demand Paging on a Highly-Parallel Shared-Memory Multiprocessor’’, Ultracomputer Note, Courant Institute, NYU, to appear.

[24] Patricia J. Teller, ‘‘*TLB Consistency and Demand Paging on a Highly-Parallel Shared-Memory Multiprocessor*’’, Ph.D. thesis, Courant Institute, NYU, in preparation.

[25] Shreekant S. Thakkar and Alan E. Knowles, ‘‘A High-Performance Memory Management Scheme’’, *Computer*, Vol. 19, No. 5, May 1986, pp. 8-20.

[26] Wei C. Yen, David W. L. Yen, and King-Sun Fu, ‘‘Data Coherence in a Multicache System’’, *IEEE Transactions on Computers*, Vol. C-34, No. 1, Jan. 1985, pp. 56-65.