# Translation-Lookaside Buffer Consistency

Patricia J. Teller
IBM Thomas J. Watson Research Center

**Shared-memory multiprocessors with multiple translation-lookaside buffers must deal with a cache consistency problem. This article describes nine solutions.**

A *translation-lookaside buffer* is a special-purpose, virtual-address cache required to implement a paged virtual memory efficiently. Shared-memory multiprocessors with multiple TLBs, also known as translation buffers or directory-lookaside tables, give rise to a special case of the cache consistency problem, which can occur when multiple images of data can reside in multiple distinct caches, as well as in main memory. If one of these images is modified, then the others become inconsistent with the modified data image and no longer represent a valid image of the data.

A processor accesses a TLB entry to determine the memory location and accessibility of referenced data. TLB entries store this information in data structures called *page tables*. Since multiple processors can read and write page tables, they can make corresponding information in TLBs stale, which in turn can cause erroneous memory accesses and incorrect program behavior.

We can solve this problem, called the TLB consistency problem, either by ensuring consistency between information in page tables and TLBs or by preventing the use of inconsistent TLB entries. In this article, I discuss nine solutions proposed in the literature. Three of these require virtual-address, general-purpose caches kept consistent by special-purpose hardware. Although I describe the general idea behind these solutions, I concentrate on the others, since I am particularly interested in identifying solutions for scalable architectures without hardware cache consistency, especially multiprocessors with a multistage network interconnecting processors and memory. In scalable multiprocessor architectures, the number of processors and memory modules can increase with the dimensions of the network, so a solution to TLB inconsistency should meet the needs of tens, hundreds, or thousands of processors.

One of a multiprocessor's main goals is to increase the execution speed of application programs by distributing the computational load among multiple processors. Therefore, it is important that the performance overhead for a solution to TLB inconsistency does not reduce the possible speedups in these computing environments. The overhead includes processor execution and idle time attributable to maintaining TLB consistency and the implicit side effects of a particular solution (such as serialized page-table modifications, increased page-ins and TLB misses, and the inability to use time-saving optimizations). In scalable multiprocessor architectures, the average time to satisfy a memory request grows with system size. Accordingly, the importance of caches, TLBs, and efficient solutions to the cache and TLB consistency problems also grows with system size.

With these points in mind, I discuss the six solutions that do not require virtual-address caches kept consistent by hardware (see Table 1). For each solution, I describe the algorithm, the hardware, the limitations with respect to the memory-sharing patterns it supports, and the cost in

0018-9162/90/0600-0026$01.00 © 1990 IEEE

**Table 1. Summary of solutions to the translation-lookaside buffer consistency problem.**

| Solution (Strategy) | Required Hardware | Memory Sharing Limitations | Performance Overhead |
|---|---|---|---|
| TLB shootdown (invalidate) | Interprocessor interrupt and architected TLB entry invalidation | None | Processors that might be using a page table that is being modified are forcibly interrupted and idled until the unsafe change has been made. Participation of the modifying processor depends on the number of other participating processors. Overhead also includes interprocessor communication and synchronization, serialization of unsafe changes to a page table, and serialization of page-table modification and use. |
| Modified TLB shootdown (invalidate) | High-priority interprocessor interrupt, architected TLB entry invalidation, and hardware support for atomic operations | None | Overhead is the same as for TLB shootdown with two exceptions: Interrupted processors are not idled, and their participation is small and possibly constant. Page-table modification and use are not serialized. |
| Lazy devaluation (delay & invalidate) | Extra TLB field, interprocessor interrupt, and architected TLB entry invalidation | No parallel execution in same address space or remote address-space modification. | In some cases, the modifying processor independently updates the TLBID or invalidates TLB entries. In other cases, overhead is the same as for TLB shootdown. |
| Read-locked TLBs (avoid & postpone) | Interprocessor interrupt and architected TLB entry invalidation | No parallel execution in same address space or remote address-space modification. Copy-on-write sometimes forfeited. | The counter is updated on each TLB reload, invalidation, and replacement. When an unsafe change cannot be postponed, overhead is the same as for TLB shootdown. |
| Memory-based TLBs (invalidate) | TLBs with bus monitors interconnected by bus, virtual-address caches, and sufficient network bandwidth | Address-space identifier or single, global address space. Given multiple memory clusters, pages in different clusters cannot map to same page and virtual-memory management is restricted. | The TLB notifies the processor of a page fault, protection exception, and virtual-memory deallocation. Also, memory requests contain virtual addresses, and there might be contention for a cluster bus and TLBs. |
| Validation (detect & correct) | Validation tables, extra TLB field, comparators, and sufficient network bandwidth | None | Memory requests contain a generation count. The modifying processor updates the generation count. An extra network trip is needed when a stale TLB entry is used. Overhead also includes a solution to the generation-count wraparound problem. |

processor execution time and multiprocessor performance. It is difficult to determine which solution is most suitable for a scalable architecture. Since small-scale multiprocessors and prototypes of large-scale multiprocessors have been built, the performance of some solutions can be evaluated in these systems. However, appropriate large-scale multiprocessors are not available. Also, factors that could determine a solution's efficiency have not yet been measured (such as the frequency of page-table modifications, the rate at which TLB inconsistencies occur, and the degree of memory sharing among processes cooperating in program execution). These factors are of particular interest in scalable architectures, since a solution's efficiency becomes even more important if these measurements increase with system size.

Since I do not have data to quantitatively evaluate these solutions, I compare their potential effects on multiprocessor performance and the limitations they impose on memory sharing. When implemented in a multiprocessor with a shared bus interconnecting processors and memory, the overhead is very low for the three solutions that require virtual-address caches kept consistent by hardware, since only the processor modifying a page table participates in the algorithm, and this processor's related execution time is small and constant.

Although solutions for multiprocessors with more general interconnnects do not have such small overheads, two solutions — memory-based TLBs and validation — come close to achieving this goal. Such a small overhead might not be necessary, however, if TLB inconsistencies are rare events and memory sharing among processes is limited. In this case, a solution called TLB shootdown, which requires essentially no hardware support, might be adequate, even though interprocessor communication and synchronization are inherent in its algorithm.

The number of ways in which TLB inconsistencies can arise is a function of the generality of memory-sharing patterns among processes. Of the six solutions, TLB shootdown, modified TLB shootdown, and validation solve the TLB consistency problem for all memory-sharing patterns. If the targeted multiprocessor has only one cluster of memory modules, then memory-based TLBs joins this group. Otherwise, memory-based TLBs, as well as lazy devaluation and read-locked TLBs, solve the problem for only a limited set of memory-sharing patterns.

I first define the function of a TLB and describe how TLB inconsistency arises both in uniprocessor and multiprocessor architectures. Then, after explaining how the problem of TLB consistency is solved in a uniprocessor and in multiprocessors with a shared bus, virtual-address caches, and hardware cache consistency, I describe solutions that can be implemented in multiprocessors with more general interconnection networks and without hardware cache consistency.

## Virtual memory and TLBs

Without a TLB, two or three memory accesses might be needed to satisfy a data request. To understand why this is so, consider the organization and management of a paged virtual-memory system and the function of a TLB.

A hierarchical memory system that supports paged virtual memory includes both main and auxiliary memories. Main memory is divided into frames, each containing an equal number of memory locations. Virtual memory comprises a set of virtual pages, the size of each being a multiple of the corresponding frame size. An image of a virtual page can reside in both main and auxiliary memory, but only the image in main memory can be read or written. Therefore, these two images are often inconsistent.

While executing a program, a processor references data by virtual address. The high-order bits of a data item's virtual address identify the page where the data is stored, and the low-order bits indicate the displacement from the beginning of the page to the data's location. Although data is referenced by virtual address, data often can be accessed only when its virtual address is translated to a physical address, where the high-order bits identify a frame and the low-order bits identify a displacement. Address translation is not necessary if a virtual-address, general-purpose cache is associated with each processor and the referenced data is cacheable and cache resident.

Virtual-to-physical address translation information is stored in a page table, which has an entry for each page. The information includes

- the location of the page in physical memory (a frame number);
- a protection field indicating how the page can be accessed, for example, read/write or read-only;
- a valid bit (sometimes called a resident bit) indicating if the frame number is valid; and
- a modify bit (sometimes called a dirty bit) indicating if the page was modified since it became main-memory resident, that is, if an exact image of the page resides in auxiliary memory.

TLBs give processors fast access to translation information for recently referenced pages, so the processors need not access a page table whenever address translation is necessary. Since programs generally exhibit locality of reference, a process usually accesses the same page a number of times during a certain time interval. Thus, a TLB often eliminates the need for a processor to access translation information from main memory. Measurements on the VAX-11/780 reveal that translation information can be accessed from a TLB rather than from a page table more than 97 times out of 100; that is, the TLB miss rate is approximately 3 percent.[1] Without a TLB, a data request might require one or two additional memory accesses. Two accesses might be needed when virtual memory is organized in segments, each containing a set of pages. In this case, the processor might also have to access a data structure called a segment table to get the location of the page table containing the referenced page's translation information.

Figure 1 illustrates the data paths of a typical physical-address cache and TLB design, where the page and frame sizes are assumed to be equal. In this case, cache access and address translation can overlap to some extent.[2] A processor uses the high-order bits of the virtual address (the page number) to get the referenced page's translation information from the TLB. A *TLB hit* occurs if a valid TLB entry exists for the page. The number of the frame containing the page is then concatenated with the low-order bits of the virtual address, yielding the referenced data's physical address, which can be used to access either the cache or main memory. Otherwise, a *TLB miss* occurs, initiating a *TLB reload*. Using the virtual page number to form the address of the page's page-table entry, a TLB reload accesses and loads in the TLB a copy of the translation information stored in the page table. In general, one entry must be removed from the TLB to make room for another.

When the referenced page is not in main memory, a *page fault* occurs. If the page is not already becoming resident, such a process, called a *page-in*, begins. A page-in allocates a frame to the page and, if necessary, copies the page's image in auxiliary memory to the allocated frame. If all physical memory has been allocated, a page-in evicts another page from main memory. Before a page is evicted, however, auxiliary memory must contain an exact image of it; this could require writing the evicted page to auxiliary memory before overwriting it with the newly referenced page.

Multiple processes can access, or share, a page by either a one-to-one or a many-to-one mapping of pages to frames. General memory-sharing patterns are produced when

- processes sharing a page via a many-to-one mapping can execute concurrently on different processors;
- multiple processes can execute concurrently in the same address space (the set of virtual addresses a process can reference); and
- a process can modify the translation

information of a page in the address space of another process, which might be executing concurrently on a different processor.

When pages are mapped to frames on a many-to-one basis and process identifiers are not affixed to virtual page numbers, multiple processes can refer to the same page with different virtual page numbers. Therefore, switching processor execution from one process to another, called a *context switch*, requires invalidating all TLB entries. This invalidation is called a *TLB flush*. If pages are mapped to frames on a one-to-one basis or process identifiers are affixed to virtual page numbers, a page is always referenced with a distinct page number. Therefore, a context switch in a uniprocessor does not require a TLB flush. This is also the case in a multiprocessor if TLB consistency is guaranteed when processes migrate among processors.

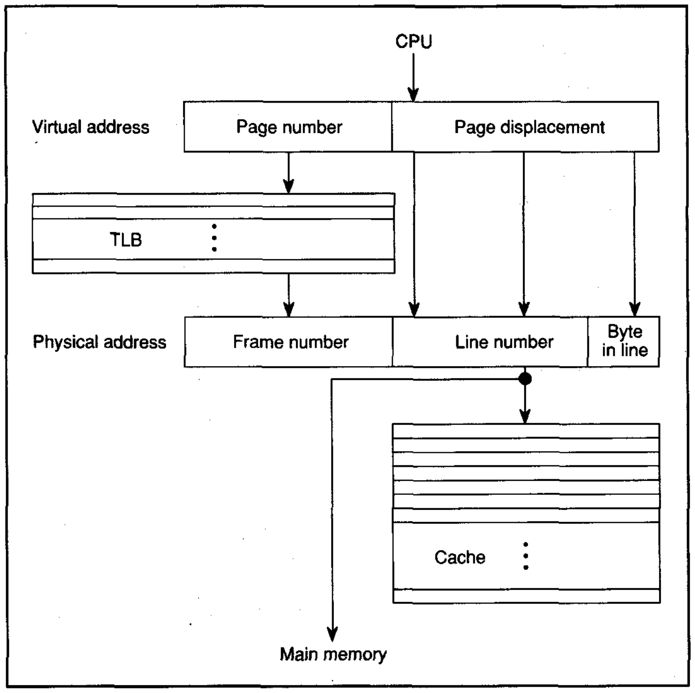

**Figure 1. Data paths of a typical physical-address cache and TLB design.**

## The problem of TLB consistency

More than one image of a page's translation information can exist: one is stored in the page table and others can be stored in TLBs. Processes can have access to multiple images, using TLB images for virtual-to-physical address translation and accessing the page-table image to perform TLB reloads or modify translation information. Therefore, page-table modifications can make TLB entries inconsistent with the page-table entries they are supposed to mirror. Since inconsistent TLB entries can generate erroneous memory accesses, *this TLB consistency problem* must be solved by ensuring consistency or preventing the use of inconsistent entries.

TLB inconsistencies resulting from updates of page-reference history information (such as bits that record if a page has been modified or referenced during some time interval) are harmless if precautions are taken when updating page tables. However, inconsistencies resulting from other page-table modifications, categorized as safe and unsafe changes, can be harmful.

*Page-reference history information* need not be consistent among TLBs and page tables. However, it is crucial that the reference history information stored in page tables reflect states that will not cause erroneous program behavior. Information can be lost if a modify bit that indicates whether a page has been written is inaccurate, since the only accurate representation of a page (stored in main memory) can be overwritten if it does not also reside in auxiliary memory. On the other hand, a bit that indicates whether a page has been referenced need not be accurate, since it is generally used as a heuristic to select a page to replace in main memory; replacing one page instead of another does not cause errors, although it can have performance consequences. In any case, if processors can concurrently modify a page-table entry (for example, if one can change a page's mapping while another sets the modify bit), then a page's reference history must be updated in such a way that it does not corrupt current page-table information.

A *safe change* results from

- a reduction in page protection, such as the modification of access rights from read-only to read/write, or
- a page's becoming main-memory resident.

We can avoid using a TLB entry that is inconsistent due to a safe change by designing the hardware so that using the entry generates an exception, which invokes an operating-system trap routine that invalidates or corrects the TLB entry. For example, suppose the translation information for page *a*, stored in a valid TLB entry accessed by processor *X*, defines the page's protection as read-only. Also, suppose *a*'s page-table entry is subsequently modified by a process executing on processor *Y* to reflect a change in protection from read-only to read/write. When a process executing on *X* attempts to write *a*, the operating system intervenes, since the TLB entry for *a* that *X* accessed defines the protection as read-only. Checking *a*'s page-table entry, the operating system finds that the protection has been increased and invalidates the stale TLB entry. After *X* reloads a consistent TLB entry for *a* and resumes execution (assuming *X* hasn't started executing a different process), the modification of *a* will be successfully and correctly executed.

In contrast, *unsafe changes* cause TLB inconsistencies that cannot be detected during program execution. This class of

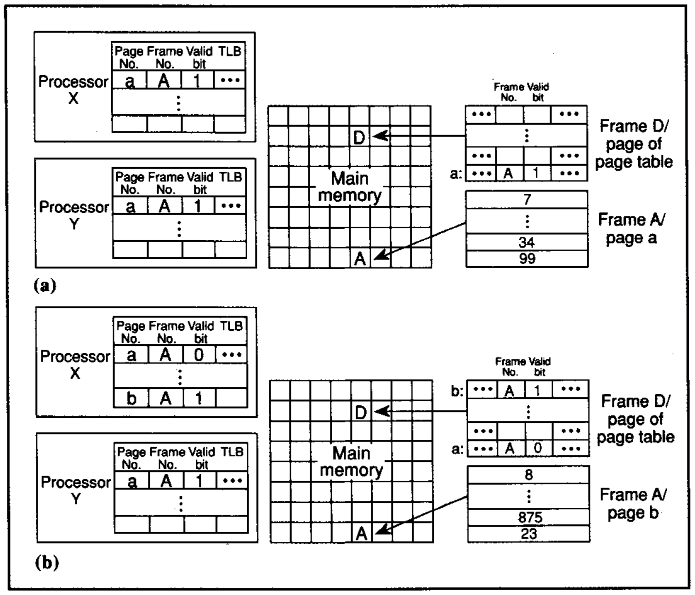

**Figure 2. TLB inconsistency in a multiprocessor: (a) before processor X evicts page a; (b) after processor X replaces page a with page b.**

page-table modifications raises the need for TLB management that ensures consistency. An unsafe change results from

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

The virtual-to-physical memory mapping of a page is invalidated when virtual memory is deallocated or when a page is evicted from main memory. Virtual memory is deallocated when virtual pages are removed from the address space of a process. A shared page's protection can be increased to implement the copy-on-write optimization and interprocess communication. Copy-on-write saves copying overhead by letting multiple processes share a page as a read-only page until one of the processes attempts to modify it.

Unsafe changes can cause TLB inconsistency in both uniprocessor and multiprocessor systems, as illustrated in Figure 2 and described below. Consider a multiprocessor system, and suppose that valid entries map page *a* to frame *A* in both the TLB accessed by processor *X* and the TLB accessed by processor *Y*. Now suppose that a process executing on *X* maps page *b* to frame *A* and evicts page *a*, updating *X*'s TLB accordingly. Unless *Y* is prevented from using *a*'s inconsistent TLB entry, it could erroneously access *b* when attempting to access *a*. Now consider *X* by itself. Whether in a uniprocessor or multiprocessor system, *X* faces the same problem as *Y* if it does not update its own TLB after modifying the mapping for page *a* and before issuing any other memory requests.

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

## Solutions to the TLB consistency problem

The TLB consistency problem is easy to solve in a uniprocessor, but it is decidedly more difficult in a multiprocessor. In a multiprocessor architecture with multiple TLBs, in addition to a consistency problem between a page table and a TLB, a consistency problem exists among the system's multiple TLBs and page tables. Figure 3 shows a system with *N* processors and *N* memory modules, where one TLB is associated with each processor. The memory modules are collected into *m* clusters of size *c*, where $a \le m \le N$, and $cm = N$, and a frame is interleaved across a cluster's memory modules. Processors and memory are interconnected by a shared bus or by a more general network, such as a multistage network. In bus-based architectures, the shared bus can help solve the cache and TLB consistency problems. In multiprocessors with more-general interconnection networks and no bus-interconnecting processors, these problems are more difficult.

Although the solutions I describe that require virtual-address caches kept consistent by hardware are limited to bus-based multiprocessors, the algorithms themselves are not so limited. Thus far, hardware cache consistency has been implemented only in bus-based multiprocessors. However, distinct solutions to both the cache and TLB consistency problems have been proposed for multiprocessors with more-general interconnection networks. For example, directory-based schemes or software-assisted cache management can solve the cache consistency problem in multiprocessors with multistage interconnection networks. Directory-based schemes incur an overhead that does not scale with the dimensions of the network[3], so this approach to cache consistency might only be suitable for a limited class of applications, although it could provide an efficient solution to the TLB consistency problem. Software-assisted cache management[4,5] depends on information available to the compiler. This approach could prove effective for ensuring cache consistency, but it is not suitable for solving the TLB consistency problem because information required to manage virtual memory is available to the operating system and not to the compiler.

**Solutions for uniprocessors.** In a uniprocessor, the operating system ensures TLB consistency by inhibiting context switching while the processor modifies a page-table entry and updates its TLB accordingly. No memory accesses occur during this time except to access the page table, so no inconsistent TLB entries are used to access memory. In a shared-memory multiprocessor, this approach can only ensure the consistency of the TLB accessed by a processor modifying a page table, so no more than one TLB image of the page's translation information is guaranteed to be consistent with the page table.

**Solutions requiring virtual-address caches and hardware cache consistency.** In bus-based multiprocessors, we can prevent the use of inconsistent TLB entries by not allowing memory accesses after a page table's main-memory image is modified and before all TLBs are consistent with the modified page table. We accomplish this by associating a bus monitor with each cache to recognize all memory accesses. The TLB consistency solutions proposed by Cheriton, Slavenburg, and Boyle,[6] Goodman,[7] and Wood et al.[8] use this method.

These solutions assume each processor has a virtual-address, general-purpose cache that stores translation information with other data and is kept consistent via a chosen protocol.[9] Solving the cache consistency problem also solves the TLB consistency problem, since would-be TLB entries are stored in, and accessed from, the cache. Memory requests transmitted on the bus trigger the bus monitors to ensure cache consistency. When a page-table entry is modified, each monitor checks to see if translation information for the associated page is cache resident; if so, the monitor invalidates or updates the corresponding cache entry.

Two relevant issues give a general idea of how each of these solutions works (I have omitted some important implementation details). First, I show how the bus monitor can query the cache when the cache is addressed by a virtual address and the memory request is targeted to a physical address. Second, since a many-to-one mapping of pages to frames can yield more than one cache entry for the same data, I explain how the solutions handle this problem, called the *address synonym problem*.[2]

The solution of Wood et al.,[8] implemented as part of the Symbolic Processing Using RISC (SPUR) workstation project at the Univeristy of California at Berkeley, assumes a dual-address bus, where a memory request includes both the virtual and physical addresses of the referenced data. The bus monitor can thus query the cache with the transmitted virtual address. The synonym problem is avoided by assuming a one-to-one mapping from pages to frames, which defines a single, global address space.

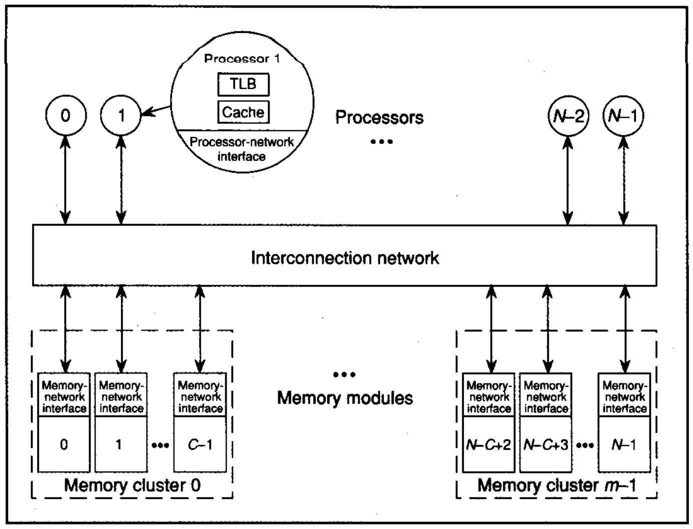

**Figure 3. A multiprocessor architecture.**

This idea can be taken one step further to support a many-to-one mapping of pages to frames, increasing the ways memory can be shared. In Goodman's solution,[7] the cache has two copies of the cache directory, one accessed by the processor with virtual addresses and the other by the bus monitor with physical addresses. Thus, physical rather than virtual addresses can trigger the bus monitor. The system binds and unbinds virtual addresses to physical addresses in response to bus transactions, solving the synonym problem.

Stanford University has incorporated the solution proposed by Cheriton, Slavenburg, and Boyle[6] in the design of the VMP multiprocessor, an experimental shared-memory, bus-based multiprocessor. If the cache cannot satisfy a data request, the frame containing the data is cached. An action table maintained by each bus monitor contains information for each cached page frame. Bus transactions trigger the bus monitors to ensure cache consistency. Since the bus monitors' actions are keyed to frame numbers, the synonym problem is solved.

When implemented in a bus-based multiprocessor, the overhead associated with these solutions is very low because only the modifying processor participates in the algorithm and its participation per page-table modification is small and constant. The bus monitors maintain consistency independently of the processors. As mentioned earlier, only the implementations are limited to bus-based multiprocessors; the algorithms are not. In any case, solutions to the TLB consistency problem are also needed for multiprocessor architectures without virtual-address caches kept consistent by hardware.

**Solutions without hardware cache consistency.** Six TLB consistency solutions can be used in multiprocessors without hardware cache consistency. Two solutions require essentially no hardware: TLB shootdown and read-locked TLBs. Three solutions — TLB shootdown, modified TLB shootdown, and validation (and, if the targeted multiprocessor has only one cluster of memory modules, memory-based TLBs) — do not limit memory sharing among processes. But none of these solutions have the small overhead exhibited by the solutions described above. If the targeted architecture has sufficient bandwidth, however, memory-based TLBs and validation (which are meant for scalable multiprocessor architectures with multistage interconnection networks) come close to achieving this goal, since each meets the following criteria:

- The participation of a processor modifying a page table is small and constant.
- The participation of another processor

is necessary only when a page affected by an unsafe change is accessed from memory (memory-based TLBs) or when an inconsistent TLB entry is used (validation).

• Locks are placed on the smallest possible entities.

• Serialization is not introduced.

• Explicit interprocessor communication and synchronization are not required.

*TLB shootdown.* Black et al.[10] have proposed an essentially hardware-independent solution called *TLB shootdown.* This algorithm is included in Carnegie Mellon University's Mach operating system, which has been ported to multiprocessor architectures including the BBN Butterfly, Encore's Multimax, IBM's RP3, and Sequent's Balance and Symmetry systems. In this algorithm, a lock associated with each page table must be secured to modify either a page table or the list of processors that may be using the table. The following sequence describes generally how the algorithm works.

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

• When a processor receives the interrupt, it clears its active flag.

• The modifying processor busy-waits until the active flags of all interrupted processors are clear and then modifies the page table. Finally, after releasing the page-table lock and setting its active flag, the modifying processor resumes execution.

• After clearing its active flag, each interrupted processor busy-waits until none of the page tables it is using are locked. Then, after executing the required TLB actions and setting its active flag, it resumes execution.

This algorithm idles all processors that may be using a page table while it is being modified. In addition, the modifying processor cannot make any modifications untilall interrupted processors have cleared their active flags. This synchronization can be very costly for applications where many processes share data. Black et al. state that their algorithm could pose problems for large-scale systems because it scales linearly with the number of processors. Extrapolating measurements from an instrumented Mach kernel running on a 16-processor Encore Multimax indicates that machines with a few hundred processors might not experience a performance problem, except with regard to kernel space. To surmount this problem, Black et al. suggest restructuring the operating system's use of memory so that TLB shootdowns are limited to groups of processors rather than all processors. Note, however, that parallel programs with the same degree of sharing as exhibited by the operating system will encounter the same performance problems as the operating system.

*Modified TLB shootdown.* As Rosenburg explains,[11] the level of synchronization exhibited by the original TLB shootdown algorithm is necessitated by architectural features of multiprocessors to which Mach is targeted. Rosenburg's version of this algorithm uses features of IBM's RP3 and requires less synchronization. Experiments on the RP3 show that the time spent by the interrupted processors can remain constant even as the number of processors grows. In particular, processors using a page table need not busy-wait while the table is being modified, and a processor can modify the table without waiting for other processors to synchronize. In addition, even though page-table modifications are serialized, processors can use a shared page table during a modification, since modifications are performed atomically. The semantics of this algorithm are slightly different from the original TLB shootdown algorithm, since at any given moment one processor can use a page's old translation while another processor uses a new one. Rosenburg's algorithm has been implemented in the version of Mach running on the RP3. In both versions:

• The participation of a modifying processor grows with the number of processors involved in the algorithm.

• The execution of all processors that might have a TLB entry that will become inconsistent (or that is inconsistent, in the case of modified TLB shootdown) as a result of an unsafe change are forcibly interrupted, whether or not they have used or will use the translation information in question.

• Parallelism is limited, since only one process can modify a given page table at one time and processors must synchronize and communicate with one another.

• The scheduling of a process is delayed if a page table it may use is being modified.

TLB shootdown ensures TLB consistency no matter how memory is shared, since processors cannot use translation information associated with a page table while the table is being modified. As mentioned above, Rosenburg's version has slightly different semantics and requires extra hardware support, but it is as effective and performs better.

TLB shootdown is a good solution to the TLB consistency problem for multiprocessors with no hardware support for this purpose. In addition, if unsafe changes rarely occur and memory is not widely shared among processes, then either version of the algorithm should perform well. Otherwise, the overhead of these solutions is likely to adversely effect multiprocessor performance.

*Lazy devaluation.* Why not postpone invalidating TLB entries until absolutely necessary? Thompson et al.[12] proposed this strategy, called *lazy devaluation,* to ensure TLB consistency when virtual memory is deallocated and when page protection is increased. Lazy devaluation has been implemented in a multiprocessor architecture based on the MIPS R2000 processor running the System 5.3 Unix operating system. Although this solution is effective for the targeted multiprocessor system, as observed by Black et al.,[10] neither this Unix implementation nor this TLB consistency solution supports multiple processes executing in the same address space or allows a process to modify the address space of another executing process.

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

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

This approach has problems when processes execute in parallel within the same address space or when a process modifies the address space of another process that might be executing concurrently on another processor. When virtual memory is deallocated, modifying a process' TLBID makes associated TLB entries inaccessible to that process but not to other processes with the same address space. The procedure suggested for maintaining TLB consistency in the face of increases in protection is only effective for a modifying processor's TLB. It does not work if processes executing concurrently on other processors share the same address space, although it can be used to implement the copy-on-write optimization.

If processes do not migrate frequently and unsafe changes rarely occur, then lazy devaluation could be efficient for systems similar to its target system. In fact, preliminary performance figures indicate this approach succeeds for the target system without excessive TLB flushing. However, lazy devaluation could adversely affect multiprocessor performance if unsafe changes are not rare events. To improve performance, Thompson et al. suggest maintaining extra information to allow selective invalidation of TLB entries rather than systemwide TLB flushes. In either case, interprocessor communication and synchronization are required. In addition, lazy devaluation does not support general memory-sharing patterns.

*Read-locked TLBs.* Preventing a TLB entry from becoming invalid prevents TLB inconsistency. This is the premise of *read-locked TLBs*, proposed by myself, Kenner, and Snir.[13] This strategy prohibits an unsafe change while a valid copy of the translation information to be modified resides in one or more TLBs. In essence, a processor maintains a read lock on a page-table entry as long as the page's translation information resides in its TLB. Thus, TLB entries never become inconsistent, and their modification and use is serialized. However, some unsafe changes might have to be postponed, and memory sharing among processes is limited.

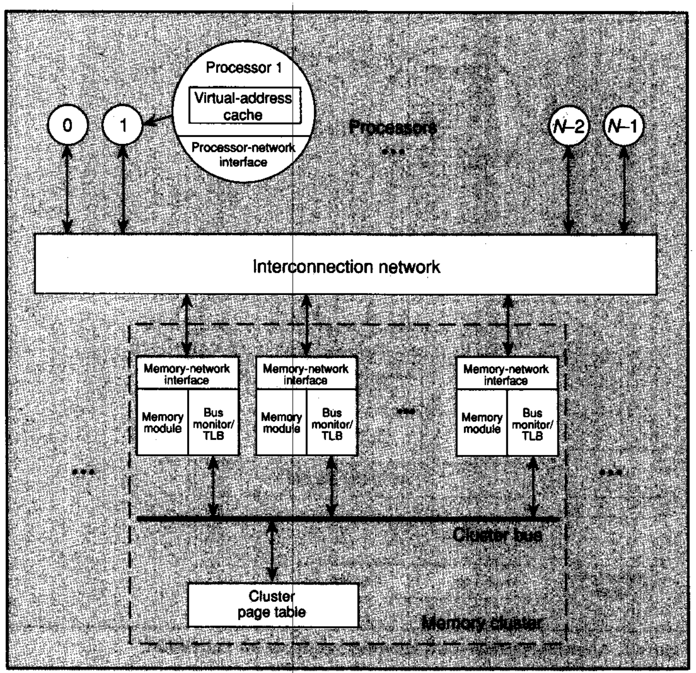

**Figure 4. A multiprocessor with hardware support for memory-based TLBs.**

To implement read-locked TLBs, a counter associated with each resident page is incremented on a TLB reload and decremented on a TLB invalidation or replacement of the page's translation information. This increase in the cost of TLB management constitutes this solution's explicit overhead. If the page-replacement policy inherently adopted by read-locked TLBs proves effective, this overhead can be partially attributed to the cost of virtual memory management. On the other hand, this solution could incur additional overhead if its restrictions increase the number of page-ins and page faults or if the inability to use the copy-on-write optimization causes unnecessary page copying.

Our strategy dictates that a page is not a candidate for eviction if any TLB contains a valid entry for it. Sequent's Dynix operating system uses a similar strategy, where a page is not a candidate for eviction if it is mapped to the address space of an active process. Virtual memory can be deallocated at any time, but a mapping for a deallocated page must not be used after the virtual page has been reallocated. We can assure this by postponing the reuse of a virtual page until no TLB contains an entry for it, except the TLB of the processor reusing the page. When these unsafe changes cannot be postponed, we must use an alternate strategy, such as TLB shootdown.

Like lazy devaluation, the read-locked TLB strategy is not effective if processes execute in the same address space or if address spaces are modified remotely. Unlike lazy devaluation, this strategy does not require flushing TLBs when a page is evicted, but it might be forced to forego the copy-on-write optimization. Either solution is suitable for a multiprocessor with restricted memory sharing if unsafe changes involving many processors and incurring an overhead similar to that of TLB shootdown do not occur frequently. In addition, techniques in each solution can effectively reduce performance overhead in other TLB consistency solutions.

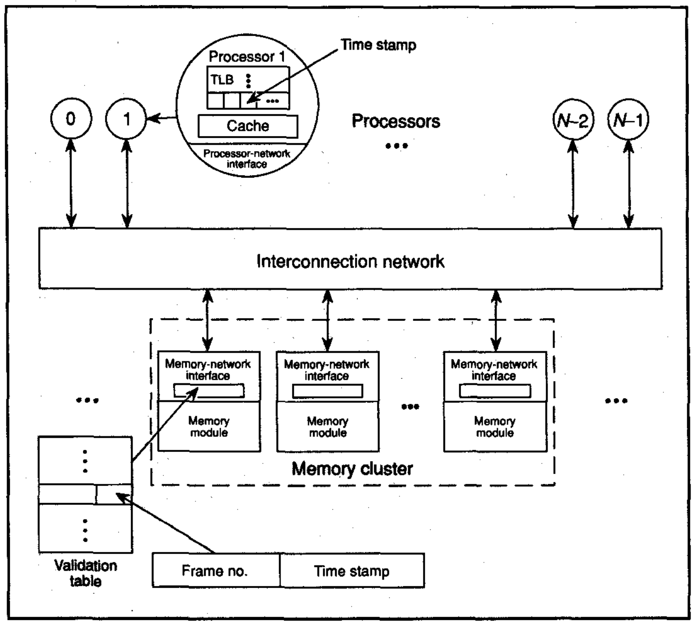

**Figure 5. A multiprocessor with hardware support for validation.**

*Memory-based TLBs.* If TLBs are associated with memory modules rather than processors, a TLB reload does not require network traffic. My colleagues and I suggested such a configuration[13] (see Figure 4). Since translation is done at the memory, general-purpose caches associated with each processor must be virtual-address caches. A memory request contains the virtual rather than physical address of the referenced data, so the virtual address defines the targeted memory module. If physical memory is organized into more than one cluster of memory modules, the number of frames to which a page can be mapped is limited; when a page resides in main memory, it is assumed to be stored in a specific cluster of memory modules. This implies that the page-replacement policy applies to each cluster independently.

The TLB is implemented as a cache with a bus monitor, as proposed by Goodman.[7] Within each memory cluster, the TLBs and the cluster page table are interconnected by a bus. A bus protocol[9] ensures consistency among a cluster's TLBs and page table. Memory access can overlap address translation, since this bus-based subsystem is independent of the path to main memory.

Remapping of virtual or physical memory is transparent to a processor, but logic at the memory must alert the processor to other unsafe changes. That is, a processor must be informed if a referenced page is not resident, if its protection has been increased, or if the page has been deallocated. In response, the processor executes a corresponding trap routine.

To solve the synonym problem (since multiple, possibly independent, processes can access the same TLB), the address presented to a TLB must include an address-space identifier, or a single, global address space must be assumed. In addition, the existence of more than one cluster of memory modules compromises the flexibility with which memory can be shared, since virtual pages that map to the same physical page must map to the same cluster.

If unsafe changes are not rare events or pages are widely shared among processes, a small performance overhead might be important. This is true for memory-based TLBs. No processor participates in the algorithm, and network traffic is generated only to inform processors of unsafe changes. In addition, this solution affords a large degree of parallelism in maintaining TLB consistency.

However, other issues can affect performance:

- Since the TLBs are interconnected by a bus, the number of memory modules per cluster is limited.
- As the number of TLBs in a cluster grows, bus traffic increases if the intracluster rate at which unsafe changes or TLB misses occur also grows. This increase in traffic could raise the cost of a TLB reload and lengthen the time to satisfy a memory request.
- If the path between the processors and memory is limited so that the size of the virtual address increases the average message length, the time to satisfy a memory request could increase.
- Since more than one processor can access any one TLB, TLBs might have to be larger than those accessed by only one processor.

If these side effects do not degrade performance, then memory-based TLBs should perform well in a multiprocessor architecture with the necessary hardware support, even if unsafe changes are frequent and memory is widely shared among processes. However, if there is more than one cluster of memory modules, the function of shared memory is somewhat limited and virtual-memory management is restricted.

*Validation.* Rather than invalidating inconsistent TLB entries to prevent their use, my colleagues and I suggested another approach, called *validation.* In this strategy, translation information is validated before it is used to access data stored in memory. This is accomplished by associating a generation count with each frame and modifying the count whenever an unsafe change affects the page mapped to that frame. A memory request is augmented with the generation count stored in the issuing processor's TLB entry for the referenced page. A frame's most recent generation count is stored in the validation tables associated with a memory cluster (see Figure 5). In addition, possibly outdated generation counts are stored in TLBs and in the page-frame table, which has an entry for each page residing in physical memory. Whether the translation information used by the processor is stale is determined by comparing the generation count transmitted with the memory request with the generation count stored in the targeted memory module's validation table. If the information is not stale, the memory request is satisfied; otherwise, the processor is notified to invalidate the referenced page's TLB entry. The validation of the transmitted generation count and memory access is assumed to be atomic.

A cluster's validation table is replicated at each memory module in the cluster. (Assume that a frame is interleaved across the memory modules in a cluster.) Before an unsafe change is made, the modifying processor multicasts a message to the memory modules in the cluster where the page resides. The receipt of this message initiates the updating of the generation count associated with the frame where the page is stored. It also maintains consistency among a cluster's validation tables. The change commences only after the tables are updated. After the unsafe change is complete, the generation count stored in the page-frame table is also updated. A TLB reload, in addition to loading translation information stored in a page table, loads a generation count stored in the page-frame table. In this way, no reference to the page can be satisfied until the referencing processor has a current image of the translation information stored in its TLB. This makes validation effective no matter how memory is shared.

To implement validation, however, a frame's generation count must be updated carefully. If a generation count is incremented to change its value, a problem arises when the generation count wraps around: There could be an inconsistent TLB entry associated with the newly updated generation count value, and this inconsistent TLB entry will appear to be consistent. We can solve this problem in several ways:

- Use long generation counts, and use a TLB shootdown-like algorithm when wraparound occurs to invalidate relevant TLB entries.
- Bound the time interval during which a TLB entry is guaranteed to be safe with respect to generation-count wraparound, and invalidate the entry before the interval ends.
- Maintain a set of counters for each frame, one for each allowable generation-count value. Each counter has $N$ bits, where $N$ is the number of processors in the system. A TLB reload increments a counter, and invalidating or replacing a TLB entry decrements a counter. The generation count stored in the TLB entry selects the counter of the related frame to be incremented or decremented. A frame's generation count is updated by assigning it a value associated with a zero counter. If there are $N$ counters, there is always a zero counter. If small generation counts are used, however, and if each allowable value of a frame's generation count is present in at least one TLB (which is unlikely, since stale TLB entries are likely to be replaced during program execution), then a TLB shootdown-like algorithm must be used to create a zero counter.

The overhead is low for several reasons.

- A modifying processor acts independently of other processors, and its participation translates into a small and constant overhead that involves only memory accesses.
- A processor's participation is enlisted only when it uses an inconsistent TLB entry, and this participation has a small overhead (one extra round-trip through the network).
- It is likely that stale entries will be replaced before they are used.
- Participating processors act independently of one another.
- Each processor could simultaneously participate in ensuring the consistency of a TLB entry associated with a distinct page.

Thus, even if unsafe changes are not rare events and memory is widely shared among processes, validation should perform well in multiprocessor architectures with the necessary hardware support. Validation has two drawbacks, however. First, if the network bandwidth is not sufficient to transmit a generation count with each memory request without increasing the number of packets that comprise a message, the average time to satisfy a memory request could increase. Second, a solution to the generation-count wraparound problem must be implemented.

A solution to the TLB consistency problem for a particular multiprocessor system must support the defined functionality of shared memory and should not degrade the speedups attainable by application programs. Since we do not know what behavior will be exhibited by programs executing in different multiprocessor architectures, we can only estimate which solution is best suited for a scalable multiprocessor with a multistage network. In particular, the following issues confront us:

- None of the proposed solutions has been implemented in a large-scale multiprocessor system.
- We do not know how either the rate of unsafe changes or the number of TLBs affected by such modifications will change as the number of processors, $N$, increases.
- It is not clear how much parallelism is needed in consistency-ensuring TLB management. In a bus-based multiprocessor employing one of the solutions that requires virtual-address caches and hardware cache consistency, TLB management is serial, that is, only inconsistencies caused by one page-table modification are corrected at any one time. In an architecture with a multistage network, inconsistencies caused by a number of unsafe changes can be corrected in parallel, the number depending on the solution.
- There are questions about how to measure a problem's execution speedup and, thus, how to evaluate the performance of solutions to the TLB consistency problem. Do we look at the behavior of a certain-sized problem executing on one processor versus its behavior executing on $N$ processors? Or do we look at its behavior versus the behavior of that same problem with a size that grows with $N$, executing on $N$ processors?

It seems reasonable to increase the problem size with $N$. In this case, I feel the rate of unsafe changes and the number of processes sharing memory will increase as $N$ increases. Therefore, algorithms whose performance depends on $N$ will not scale well. If this is the case, then TLB shootdown, modified TLB shootdown, lazy devaluation, and read-locked TLBs are not good solutions for scalable multiprocessor architectures. In contrast, the memory-based TLBs strategy exhibits the following characteristics, which yield good performance in scalable architectures:

- A processor is not interrupted to participate in the solution.
- Little extra communication is required to maintain TLB consistency.
- No processor synchronization is required.
- Parallelism is inherent in the algorithm it implements.

But, if the memory modules are organized in more than one cluster, then the memory-based TLBs strategy does not support all memory-sharing patterns and restricts virtual-memory management.

Therefore, validation seems the most promising solution for scalable multiprocessor architectures, since it has all the characteristics of memory-based TLBs, although extra communication costs can be inherent in its implementation. Specifically:

- If the transmission of a generation count with each memory request increases the average network latency, then the time required to satisfy a memory request increases.
- When a processor uses an inconsistent TLB entry, a round-trip through the network is required to discover that the entry is stale.
- Communication might be required to solve the generation-count wraparound problem.

We can not decisively conclude which solutions are best suited to scalable multiprocessor architectures until we know how programs behave in these environments. The solutions described demonstrate ways of solving the TLB consistency problem. They may either prove to be efficient solutions in multiprocessor systems or provide a basis on which to design other solutions to this problem. ■

## Acknowledgments

Many thanks to the referees and Michel Dubois, whose constructive criticism shaped the final form of this paper, and to Harold Stone for much helpful advice. Also, my thanks to the other researchers who have addressed the problem of TLB consistency, without whom this paper would not be possible, and to Susan Flynn-Hummel, Bryan Rosenburg, and Edith Schonberg who reviewed the final draft of this paper. This research is part of the author's doctoral dissertation at New York University. Part of the research was conducted while the author was at NYU and was supported by the US Department of Energy under grant number DE-FG02-88ER25052, administered by Donald Austin.

## References

1. D. Clark and J. Emer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurements," *ACM Trans. Computer Systems*, Vol. 3, No. 1, Feb. 1985, pp. 31-62.

2. A. Smith, "Cache Memories," *Computing Surveys*, Vol. 14, No. 3, Sept. 1982, pp. 473-530.

3. A. Agarwal et al., "An Evaluation of Directory Schemes for Cache Coherence," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, 1988, Computer Society Press, Los Alamitos, Calif., Order No. 861, pp. 280-289.

4. H. Cheong and A. Veidenbaum, "A Cache Coherence Scheme With Fast Selective Invalidation," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, 1988, Computer Society Press, Los Alamitos, Calif., Order No. 861, pp. 299-307.

5. R. Cytron, S. Karlovsky, and K. McAuliffe, "Automatic Management of Programmable Caches," *Proc. 1988 Int'l Conf. Parallel Processing*, 1988, Computer Society Press, Los Alamitos, Calif., Order No. 889, pp. 229-238.

6. D. Cheriton, G. Slavenburg, and P. Boyle, "Software-Controlled Caches in the VMP Multiprocessor," *Proc. 13th Ann. Int'l Symp. Computer Architecture*, 1986, Computer Society Press, Los Alamitos, Calif., Order No. 719, pp. 366-374.

7. J. Goodman, "Coherency for Multiprocessor Virtual Address Caches," *Proc. Second Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1987, pp. 72-81.

8. D. Wood et al., "An In-Cache Address Translation Mechanism," *Proc. 13th Ann. Int'l Symp. Computer Architecture*, 1986, Computer Society Press, Los Alamitos, Calif., Order No. 719, pp. 358-365.

9. P. Sweazey and A. Smith, "A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus," *Proc. 13th Ann. Int'l Symp. Computer Architecture*, 1986, Computer Society Press, Los Alamitos, Calif., Order No. 719, pp. 414-423.

10. D. Black et al., "Translation Lookaside Buffer Consistency: A Software Approach," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1989, pp. 113-122.

11. B. Rosenburg, "Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors," *Proc. 13th Symp. Operating Systems Principles*, 1989, pp. 137-146.

12. M. Thompson et al., "Translation-Lookaside Buffer Synchronization in a Multiprocessor System," *Proc. Winter 1988 Usenix Tech. Conf.*, 1988, pp. 297-302.

13. P. Teller, R. Kenner, and M. Snir, "TLB Consistency on Highly Parallel Shared-Memory Multiprocessors," *Proc. 21st Hawaii Int'l Conf. System Sciences*, 1988, Computer Society Press, Los Alamitos, Calif., Order No. 841, pp. 184-193.

**Patricia J. Teller** is finishing her PhD in computer science at the Courant Institute of Mathematical Sciences at New York University. She has a visiting position at IBM's Thomas J. Watson Research Center. Her research interests include computer architecture, operating systems, parallel processing, and performance analysis.

Teller received her BA degree magna cum laude and her MS degree in computer science from New York University. She is a member of the IEEE, ACM, SIGArch, SIGOps, and SIGMetrics, and Phi Beta Kappa.

Readers may contact Teller at IBM T.J. Watson Research Center, Room H3-F30, PO Box 704, Yorktown Heights, NY 10598.