Translation Lookaside Buffer Synchronization in a Multi−Processor System

M. Y. Thompson

J. M. Barton

T. A. Jermoluk

J. C. Wagner

Silicon Graphics, Incorporated

ABSTRACT

Most current computer architectures use a high−speed cache to translate user virtual addresses into physical memory addresses. On machines that require software to implement cache fills and invalidations, the software task is fairly straightforward. In a multi−processor multi−cache configuration, however, where processes are allowed to migrate across processors, there is an inherant synchronization problem, as well as performance issues.

This paper discusses a solution to these issues that is general enough to implement without specialized hardware, yet offers good performance.

1. Introduction

Most current computer architectures use a high−speed cache to translate user virtual addresses into physical memory addresses (a *translation lookaside buffer*, or TLB). When a translation entry does not exist for a particular user virtual address, some combination of software and hardware must be employed to create that translation and supply it to the TLB. When a current virtualphysical translation changes or becomes invalid, as happens when a physical page is ‘‘stolen’’ from one process and assigned to another, an extant TLB entry must be replaced or removed. The methodology to perform these functions is well−known on a traditional single−processor (SP) computer system.

It was found, however, that the methodology available was insufficient when applied to a multi−processor (MP) configuration where processes are allowed to migrate across processors. In particular, the methodology fails on a multi−processor system where each processor is coupled with a private TLB: replacing or removing an entry in one TLB does not change or invalidate other, possibly extant, entries on other system processors.

This paper discusses the overall strategy that was devised to manage the TLB. The various situations in which TLB entries must be replaced or invalidated are enumerated, as are the details of both the SP and MP implementation.

2. Translation Lookaside Buffer

The target hardware is a system using the MIPS R2000 simplified−instruction−set processor. The TLB is part of the system coprocessor, one of which is associated with each processor. The TLB does not have a direct connection to memory, and it knows neither the form nor the location for page tables. TLB management is accomplished by software via coprocessor instructions. This approach requires slightly longer refill times than might occur with dedicated hardware, but has the advantages of simplified hardware and flexibility.[MIPS86]

Each TLB entry consists of two words. The low word contains a physical page frame number and various hardware bits (valid, dirty, etc.). The high word contains a virtual page number (VPN) and an id (TLBID). The id field is currently six bits −− thus, 64 TLB ids are available. Additionally, there exist two index registers which are used to address TLB entries (an Index register and a Random register), and an EntryLo and EntryHi register pair. The formats of the EntryLo and EntryHi registers pairs are the same as the TLB entries. Figure 1 shows these formats. A TLB match occurs when there exists an entry which matches the input virtual address and the current TLBID field in the EntryHi register. Misses cause an exception, as do references to invalid entries, or stores to an address that matches a TLB entry that is not marked dirty.

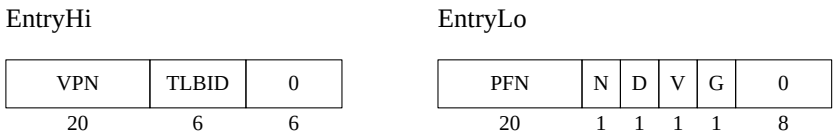

Figure 1. Translation Lookaside Buffer Format

Coprocessor instructions exist to probe for an extant entry (the index of the entry is left in the Index register); to read a specific TLB entry (EntryHi and EntryLo receive the contents of the TLB entry indexed by the Index register); to write to a specific entry (EntryHi and EntryLo via the Index register); and to write to a ‘‘random’’ TLB entry (the pseudo−random Random register is used as the index).

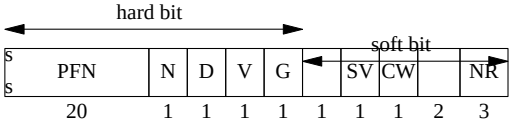

Figure 2. Software Page Table Format

The page table is the software counterpart of the TLB. When a TLB entry is written, it is the software page table entry that is copied into the TLB. Bit fields not used by the TLB hardware are used for (software) valid and copy−on−write flags, and for a reference counter. Figure 2 shows the page table entry format.

3. Operating System Support

There are five different situations in which, on our system, (a port of the 5.3 UNIX Operating System) TLB entries can become inconsistent with process state. They are:

1. A process shrinking its address space.
2. Physical pages being ‘‘stolen’’ from a process.
3. System virtual address reallocation.
4. System physical address reallocation.
5. Writes to copy−on−write pages.

The first situation occurs when a process sets its maximum data value to a lower value, when it releases a shared memory segment, or when it releases all its address space on exit.

The second scenario occurs in low−memory situations when the memory management daemon takes physical pages from a process to be available to others.

The system also keeps a map of virtual addresses which are allocated for short durations for purposes such as mapping user physical addresses into system virtual space for DMA. After each use, the virtual addresses are returned to the system address map for reuse.

"UNIX is a trademark of AT&T Bell Laboratories"

Similarly, physical pages are often assigned for varying durations to steadfast system virtual addresses such as file system buffers. Over time, pages may be assigned, usurped, and new pages assigned.

Lastly, when a process writes to a shared copy−on−write page, a copy of the page is created and the new page is assigned to the writing process.

In all of these situations, there can exist entries in the various TLBs that are suddenly incorrect. In all of these situations it is necessary to ensure that the process doesn't access addresses that it has surrendered. To that end, there must be no entries in the TLB on the processor on which a user process is running that map virtual/physical addresses which are no longer correct. Similarly, the kernel process must take pains not to access kernel virtual addresses which are no longer valid.[1]

In our original SP port, the TLB replacement and invalidation policies were situational. That is, for each situation an expedient method was devised to keep the TLB synchronized with the system state, but there existed no overall strategy for TLB management. On the MP system, it became clear that an over−all policy was essential, both to make the various mechanisms work efficiently (severally and together), and to make the problem manageable.

While on an SP system it was often appropriate to replace or remove TLB entries immediately as the entry became invalid, on an MP system this strategy suffers from overenthusiasm. It could well be the case that a process which has divested itself of pages or has had new physical pages assigned to particular virtual addresses never runs on other processors on the system, or runs on another processor only after ''natural'' events have caused the invalid entries to be replaced or removed. We decided to accentuate this tendency and put off TLB invalidations until absolutely necessary.

To implement this strategy (which we labeled ''lazy devaluation''), system and process state is recorded to understand when TLB entries on a particular processor must be replaced or invalidated. When such an event does occur, the entire TLB is flushed, and the state structures are adjusted so that, logically, the flush creates the greatest effect.

The following sections explicate the various situations and mechanisms involved.

3.1. Shrinking Processes

There are several scenarios in which a process might divest itself of current address space. These range from a process resetting its break value to a process detaching a shared memory region or unmapping a mapped file region to a process exiting.

The last case is benign −− an exiting process no longer has the ability to reference its address space.

The other cases are surmountable. Since a TLB match requires that an entry match both the input virtual address and the current TLBID (in EntryHi), assigning a new TLB id to the process effectively renders current (possibly stale) entries inaccessible. This approach is more efficient than the alternatives −− flushing the entire TLB whenever a process shrinks its address space, or probing for and invalidating each possible (now invalid) TLB entry.

It is only when there are no readily available TLB ids that drastic action needs to be taken. In that case, each process' TLB id is set to an invalid value (the id is kept in the proc structure) and the TLB is flushed. It is safe to invalidate the id field of an active process since it is guaranteed that, on an SP system, no other process besides the one requesting an id is currently running, and thus, there is no process actively using TLB ids. When a process resumes, it checks if its TLB id is still valid; if not, it requests a valid id from the id allocator.

On an MP system, there is no such guarantee −− processes on other processors may well be active. The TLB id reallocation problem is easily solved, however, by freeing only those ids whose associated process is not currently running. A field in the proc structure indicates whether the process is currently running on any processor. With suitable spin locks and semaphores to protect bit fields and TLB id allocation code, process shrinking becomes quite tenable for the operating system.

TLBIDs are managed as a site−wide resource, so, at the time that ids must be recycled all TLBs on the site must be flushed. To effect site−wide flushing, it is only necessary to set bits in a global bit field, one bit for

1. The implication is that, while user processes are not to be trusted, the kernel can certainly understand its own memory management state and take care not to abrogate its policies.

each active processor. Whenever a processor flushes its TLB, it clears its corresponding bit in the field. The initiating routine merely sets all appropriate bits beforehand, flushes its own TLB, and waits until the entire field has been cleared. (On systems that have an inter−processor interrupt facility, this wait is minimal. On systems without hardware support, simple messaging can be used to initiate TLB flushing on the various processors.)

3.2. Reclaiming Pages

The major functions of the paging daemon are to determine page usage and to free pages into the page pool when memory gets tight. As there are no hardware reference bits available, page usage on our system is determined by periodically decrementing software reference counters and turning off the hardware valid bits in the page table entries. The paging daemon has only to invalidate the corresponding entries in the TLB to cause subsequent references to produce reference faults. The fault code resets the valid bit and the reference counter for the faulting page, and drops the entry into the TLB.

Similarly, to reclaim a page for the free pool, the paging daemon clears the software and hardware valid bits in the page table entries, and inserts the pages into the free list. Semaphores associated with each virtual memory region [Bach86] are used to ensure that page faults and page manumission are, effectively, atomic.

For both reference fault enabling and page manumission, TLB entries are not invalidated individually. Instead, a number of pages, possibly spanning several regions, are operated on at once. Before the region semaphores are released, TLBs are flushed site−wide.

3.3. System Virtual Addresses

In general, the operating system runs without TLB mappings. The kernel is divided into three segments which carve out the addresses from 0x80000000 through 0xffffffff (the user segment −− kuseg −− includes all virtual addresses from zero through 0x7fffffff). References to kseg0 (0x80000000 to 0xa0000000) are cached but not mapped into the TLB. Most of the kernel's executable code and some of its data reside here. The kseg1 segment (0xa0000000 to 0xc0000000) provides uncached, unmapped references −− IO registers and ROM code are mapped to these addresses. Both kseg0 and kseg1 addresses are direct−mapped onto the first 512MB of physical address space. Like kuseg, the kseg2 segment (0xc0000000 through 0xffffffff) uses TLB entries to map virtual address to arbitrary physical ones. The operating system allocates kseg2 addresses for some dynamic structures and for performing DMA into user space.

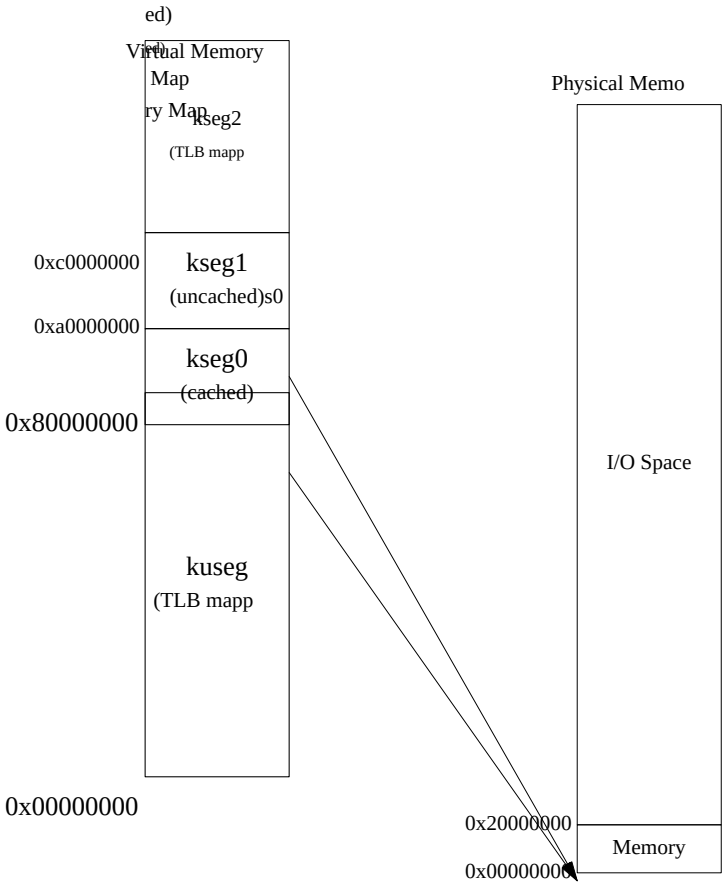

Figure 3. Hardware Defined Virtual Memory Map

For user DMA, the system allocates kernel virtual addresses from a system address map and double−maps the user's pages into the system space. The interrupt code which transfers data then does not need knowledge of the user process for which the transfer occurs. On an SP system, dropping in new TLB entries for the system virtual pages when they are allocated is sufficient to ensure that no stale TLB entries exist from the previous allocation. (The dropin code probes for a current entry for the TLBID/virtual−address pair and replaces that entry if it exists.) But on an MP system, dropping in new TLB entries on one processor does not affect other processors' TLBs. Again, instead of signalling each processor and having each processor replace or invalidate entries, we take the lazy devaluation approach. The various TLBs are allowed to fill with new entries ''naturally'', that is, by reference. Upon deallocation, however, the page is not returned to the free map, but is instead placed in a stale address map. If the system map becomes depleted, the site−wide TLB flush routine is called. This routine always merges the stale address map back into the system map while waiting for other processors to flush their TLBs.

3.4. System Kseg2 Mappings

A variation of the page reclaiming problem exists with certain kernel routines that allocate and free physical pages associated with kernel virtual addresses (for example, pages for file system buffers). Unlike the memory management paging daemon, which frees large numbers of pages at a time, pages are released in small numbers. Because of this, wholesale TLB flushing is inappropriate. Instead, we apply the precept of lazy devaluation. We track page usage through state tables and postpone TLB flushing.

When a page is returned to the free list, the valid bits are reset, but the page frame number persists in the system page tables. It is only when the virtual address is surrendered that the page table entry's page frame

number is cleared, indicating that there is no ‘‘remembered’’ association with a physical page.

When allocating a page for a system virtual address, if there exists a page frame number in the page table entry, the named page it is reassigned to the virtual address if the page is free. If the page is not available, the system−wide TLB flush routine is called. At this time, all invalid system page table entries have their physical page frame number fields cleared, indicating that there is no longer a residual relationship between the virtual addresses and the physical pages.

As a performance enhancement, when a physical page that was mapped to a system virtual address gets returned to the free memory list, its corresponding system page table entry is linked on a dirty list. The TLB flush routine traverses this list when clearing the physical page frame numbers. If a process is surrendering both the virtual and physical pages, this linking is not necessary −− returning the virtual addresses into a stale map ensures that the system won’t use the address without first flushing the TLB.

When a previously−assigned page is reassigned to the same virtual address, it must, of course, be dequeued from the dirty list.

Overloaded fields in the parallel disk block descriptor (DBD) [Bach86] are currently used for this chore. (The descriptors are unused since the corresponding pages are never swapped to disk, and, in the current implementation, the DBDs are not separably allocatable.) To facilitate dequeuing, a doubly−linked list is used; in order to fit into the DBDs, the fields are actually offsets into the system page table.

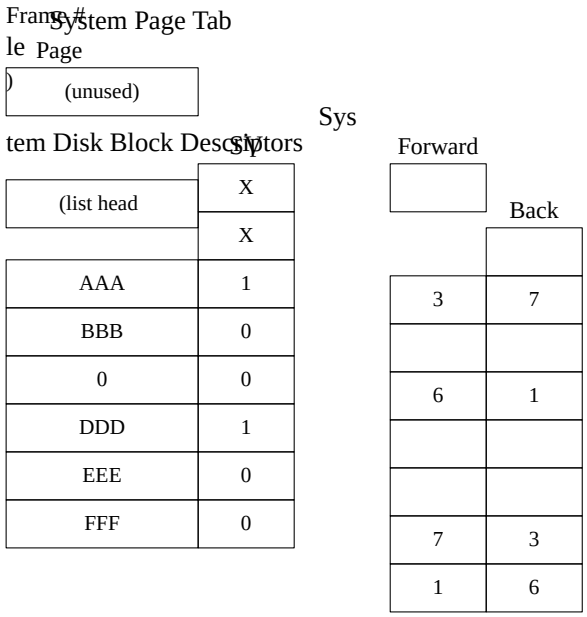

Figure 4. System Page Table with Stale Relationships

Figure 4 shows an example in which page table entries three, six and seven have been chained into the ‘‘stale relationships’’ list. Entry four is not chained −− the virtual address was released with the physical page. Figure 5 shows the same system page table entries after a system−TLB flush.

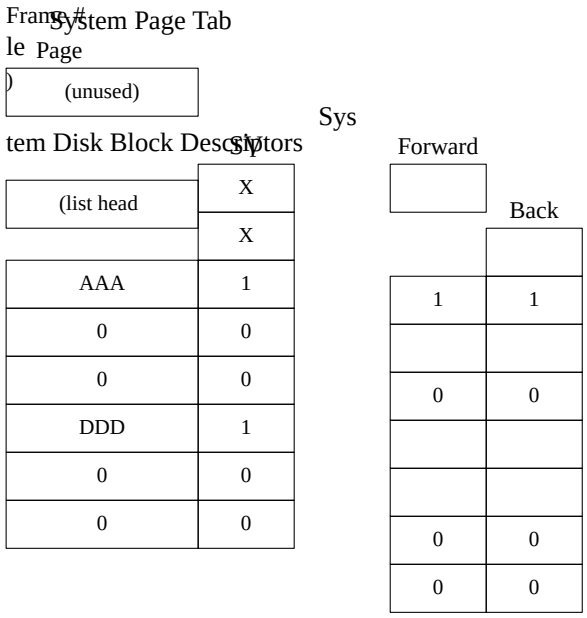

Figure 5. System Page Table After TLB FLush"

3.5. Faults −− Misses, Reference, Protection

The strategy for handling TLB misses is fairly straightforward. For first−level misses, the page table entry is copied to EntryLo, the VPN/TLBID pair is written into EntryHi, and the pair is randomly deposited into the TLB. Second−level misses are handled in a similar manner, except that the second−level entry (the TLB entry for the page table itself) is deposited into a specific TLB location, that location determined by software. On the current implementation, the processor constrains the Random register to contain a value from eight to 63. This allows entries zero through seven to be reserved for page tables, the kernel stack, and the like.

Reference faults and protection faults are handled similarly −− the page table entry for the faulting address is fetched (possibly causing a second−level miss), sanity checking is performed, and the new entry is dropped in, either replacing an extant entry, or, if none exists, dropped into a random TLB location. When a valid reference is made to an address to which a physical page is not currently assigned, the fault code must assign a physical page for the process and fill it appropriately.

None of these actions are a problem on an SP system, and, for the most part, on an MP system. Dropping an unchanged entry into a TLB is innocuous. Dropping in an entry for a newly−assigned page is trouble−free, too −− the assumption is made that the routine that disassociated the page from its previous processaddress took care to purge the (possibly extant) entries from the TLB(s).

Protection faults pose a problem on an MP system, however, when the fault is on a copy−on−write page. A copy−on−write page might be referenced by multiple processes at the time one process writes to it. The SP approach is simple: if more than one process is currently referencing the page, a new page is assigned for the writer, the data are copied, and a new TLB entry is deposited (with the dirty bit set). But on an MP system, there could exist entries on other TLBs (had the process previously run on other processors) that reflect the previous virtual/physical mapping. If the process migrated to another processor (on which it had previously run) without ensuring that the entry was purged, further references could access the wrong page.

Again, the approach is to keep state tables and avoid action until necessary. Instead of actively searching out and removing entries on TLBs throughout the system, it is just noted that there exist (possibly) stale TLB entries for this process on other processors. The minimal data structure is a bit field the size of the number of processors in the system, one for each TLB id (or, one for each proc structure, for small

systems). After assigning a new page, it is only necessary to set the bits corresponding to all but the current processor, indicating there might be invalid entries for this TLBID (process) on the flagged processors. (A new entry is deposited in the current processor's TLB, so it is not necessary to set the dirty bit for the current processor.) When a process resumes, it checks whether a bit is set for the processor on which the process is now running. If so, the current processor's TLB is flushed. To further performance, a parallel bit field is kept that indicates the processors on which the process has actually run. It is only necessary to set dirty bits for processors other than the current one on which the process has previously run.

When a process is assigned a new TLB id (either because the process is just starting up, or because it shrank, or because its id was taken away for reassignment), dirty bits in the entry indexed by the TLB id are cleared, as are the history bits for all but the current processor. (TLB ids are delivered ''clean'', that is, without any entries in any TLB using that id.) Similarly, whenever a processor flushes its TLB, the dirty bits for that processor in all entries are cleared, as are the history bits in all entries except for the currently running process.

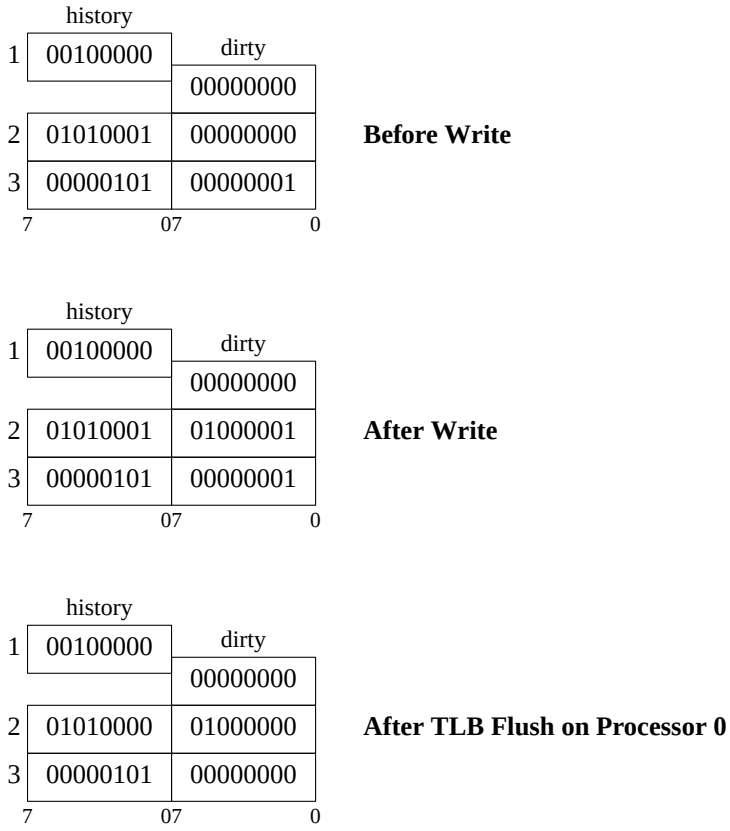

Figure 6. Example TLBID State Structures

Figure 6 shows three snapshots of an abbreviated TLBPID state structure. The first and second are just before a process owning TLBPID 2 running on processor 4 writes to a shared copy−on−write page. At the time of the write, the process has a history of having run on processors 0, 4 and 6. The last snapshot is the same TLBPID state structure just after processor 0 has flushed its TLB. Note that the persistence of the history bit 0 for TLBPID 3 implies that it is currently running on processor 0.

4. Summary

There are certainly other approaches that might have been followed to solve the problem of keeping multiple TLBs correct. Preliminary performance figures, however, indicate that the lazy devaluation approach succeeds without causing excessive TLB flushing.

Most importantly, the various state structures can be enhanced and routines tuned to take advantage of added information without changing the underlying mechanisms.

For example, the TLBPID state structures could be extended to list the individual stale entries. Instead of flushing the entire TLB, those entries (if still extant) could be individually flushed. If TLB id information were to be made available to the memory management daemon (currently, there is no path from a region structure, upon which the daemon operates, to process and TLB ids), a similar refinement could be made. It is not clear if these changes would be of benefit, but the changes could be implemented and tested without restructuring the entire TLB management system.

References

MIPS System Programmer Guide, Beta Version. Mips Computer Systems, Inc., Mountain View, CA, 1986.

Bach, Maurice J., *The Design of the UNIX Operating System*, Prentice Hall, New Jersey, 1986.