Memory Ordering in Modern Microprocessors, Part II

Paul E. McKenney

Abstract

Anybody who says computers give only right answers hasn't seen what happens when several SMP processors, each with its own cache, try to get at the same data. Here's how to keep the kernel's view of memory correct, no matter what architecture you're on.

The first installment of this series was an overview of memory barriers, why they are needed in SMP kernels and how the Linux kernel handles them [August 2005]. This installment gives an overview of how several of the more popular CPUs—Alpha, AMD64, IA64, PA-RISC, POWER, SPARC, x86 and zSeries, otherwise known as IBM mainframe—implement memory barriers. Table 1 is reproduced here from the first installment of this series for reference.

Alpha

It may seem strange to say much of anything about a CPU whose end of life has been announced, but Alpha is interesting because, with the weakest memory-ordering model, it reorders memory operations the most aggressively. It therefore has defined the Linux kernel memory-ordering primitives that must work on all CPUs. Understanding Alpha, therefore, is surprisingly important to the Linux kernel hacker.

The difference between Alpha and the other CPUs is illustrated by the code shown in Listing 1. This smp_wmb() on line 9 guarantees that the element initialization in lines 6–8 is executed before the element is added to the list on line 10, so that the lock-free search works correctly. That is, it makes this guarantee on all CPUs except Alpha.

Alpha has extremely weak memory ordering, such that the code on line 20 of Listing 1 could see the old garbage values that were present before the initialization on lines 6–8.

Figure 1 shows how this can happen on an aggressively parallel machine with partitioned caches, so that alternating cache lines are processed by the different partitions of the caches. Assume that the list header head is processed by cache bank 0 and the new element is processed by cache bank 1. On Alpha, the smp_wmb() guarantees that the cache invalidation performed by lines 6–8 of Listing 1 reaches the interconnect before that of line 10. But, it makes absolutely no guarantee about the order in which the new values reach the reading CPU's core. For example, it is possible that the reading CPU's cache bank 1 is busy, while cache bank 0 is idle. This could result in the cache invalidates for the new element being delayed, so that the reading CPU gets the new value for the pointer but sees the old cached values for the new element.

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64	Y			Y				
IA64	Y	Y	Y	Y	Y	Y		Y
(PA–RISC)	Y	Y	Y	Y				
PA–RISC CPUs								
POWER	Y	Y	Y	Y	Y	Y		Y
SPARC RMO	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86	Y	Y		Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries				Y				Y

Table 1. Summary of Memory Ordering

One could place an smp_rmb() primitive between the pointer fetch and dereference. However, this imposes unneeded overhead on systems such as x86, IA64, PPC and SPARC that respect data dependencies on the read side. An smp_read_barrier_depends() primitive has been added to the Linux 2.6 kernel to eliminate overhead on these systems. This primitive may be used as shown on line 19 of Listing 2. However, please note that RCU code should use rcu_dereference() instead.

It also is possible to implement a software barrier that could be used in place of smp_wmb(), which would force all reading CPUs to see the writing CPU's writes in order. However, this approach was deemed by the Linux community to impose excessive overhead on extremely weakly ordered CPUs, such as Alpha. This software barrier could be implemented by sending interprocessor interrupts (IPIs) to all other CPUs. Upon receipt of such an IPI, a CPU would execute a memory-barrier instruction, implementing a memory-barrier shoot-down. Additional logic is required to avoid deadlocks. Of course, CPUs that respect data dependencies would define such a barrier simply to be smp_wmb(). Perhaps this decision should be revisited in the future when Alpha fades off into the sunset.

Listing 1. Insert and Lock-Free Search

```
 1 struct el *insert(long key, long data)
 2 {
 3       struct el *p;
 4       p = kmalloc(sizeof(*p), GPF_ATOMIC);
 5       spin_lock(&mutex);
 6       p->next = head.next;
 7       p->key = key;
 8       p->data = data;
 9       smp_wmb();
10       head.next = p;
11       spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
```

```
15 {
16      struct el *p;
17      p = head.next;
18      while (p != &head) {
19          /* BUG ON ALPHA!!! */
20          if (p->key == key) {
21               return (p);
22          }
23          p = p->next;
24      };
25      return (NULL);
26 }
```

The Linux memory-barrier primitives took their names from the Alpha instructions, so smp_mb() is mb, smp_rmb() is rmb and smp_wmb() is wmb. Alpha is the only CPU where smp_read_barrier_depends() is an smp_mb() rather than a no-op. For more detail on Alpha, see the reference manual, listed in the on-line Resources.

AMD64

Although AMD64 is compatible with x86, it offers a slightly stronger memory-consistency model, in that it does not reorder a store ahead of a load. After all, loads are slow and cannot be buffered, so why reorder a store ahead of a load? Although it is possible in theory to create a parallel program that works on some x86 CPUs but fails on AMD64 due to this difference in memory-consistency model, in practice this difference has little effect on porting code from x86 to AMD64.

The AMD64 implementation of the Linux smp_mb() primitive is mfence, smp_rmb() is lfence and smp_wmb() is sfence.

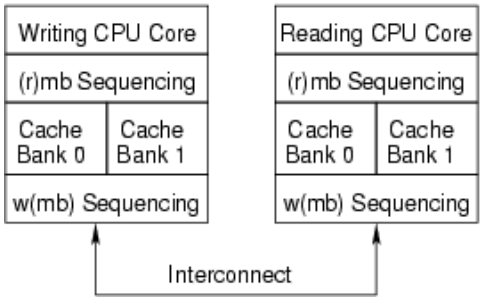

Figure 1. Why smp_read_barrier_depends() Is Required

IA64

IA64 offers a weak consistency model, so that in absence of explicit memory-barrier instructions, IA64 is within its rights to reorder memory references arbitrarily. IA64 has a memory-fence instruction named mf, as well as a half-memory fence modifier to load and store some of its atomic instructions. The acq modifier prevents subsequent memory-reference instructions from being reordered before the acq, but it permits prior memory-reference instructions to be reordered after the acq, as fancifully illustrated by Figure 2. Similarly, the rel modifier prevents prior memory-reference instructions from being reordered after the rel, but it allows subsequent memory-reference instructions to be reordered before the rel.

These half-memory fences are useful for critical sections, as it is safe to push operations into a critical section. It can be fatal, however, to allow them to bleed out.

The IA64 mf instruction is used for the smp_rmb(), smp_mb() and smp_wmb() primitives in the Linux kernel. Oh, and despite persistent rumors to the contrary, the mf mnemonic really does stand for memory fence.

PA-RISC

Although the PA-RISC architecture permits full reordering of loads and stores, actual CPUs run fully ordered. This means the Linux kernel's memory-ordering primitives generate no code; they do, however, use the GCC memory attribute to disable compiler optimizations that would reorder code across the memory barrier.

Listing 2. Safe Insert and Lock-Free Search

```
 1 struct el *insert(long key, long data)
 2 {
 3      struct el *p;
 4      p = kmalloc(sizeof(*p), GPF_ATOMIC);
 5      spin_lock(&mutex);
 6      p->next = head.next;
 7      p->key = key;
 8      p->data = data;
 9      smp_wmb();
10      head.next = p;
11      spin_unlock(&mutex);
12 }
13
14 struct el *search(long key)
15 {
16      struct el *p;
17      p = head.next;
18      while (p != &head) {
19          smp_read_barrier_depends();
20          if (p->key == key) {
21               return (p);
22          }
23          p = p->next;
24      };
25      return (NULL);
26 }
```

POWER

The POWER and PowerPC CPU families have a wide variety of memory-barrier instructions:

- sync causes all preceding instructions, not only memory references, to appear to have completed before any subsequent operations are started. This instruction, therefore, is quite expensive.
- lwsync, or lightweight sync, orders loads with respect to subsequent loads and stores, and it also orders stores. However, it does not order stores with respect to subsequent loads. Interestingly enough, the lwsync instruction enforces the same ordering as does the zSeries and, coincidentally, the SPARC TSO.
- eieio, enforce in-order execution of I/O, in case you were wondering, causes all preceding cacheable stores, which are normal memory references, to appear to have completed before all subsequent cacheable stores. It also causes all preceding non-cacheable, memory-mapped I/O (MMIO) stores to appear to have completed before all subsequent non-cacheable stores. However, the stores to cacheable memory are ordered separately from the stores to non-cacheable memory, which, for example, means that eieio does not force an MMIO store to precede a spinlock release.
- isync forces all preceding instructions to appear to have completed before any subsequent instructions start execution. This means that the preceding instructions must have progressed far enough that any traps they might generate either have happened or are guaranteed not to happen. Furthermore, any side effects of these instructions—for example, page-table changes—are seen by the subsequent instructions.

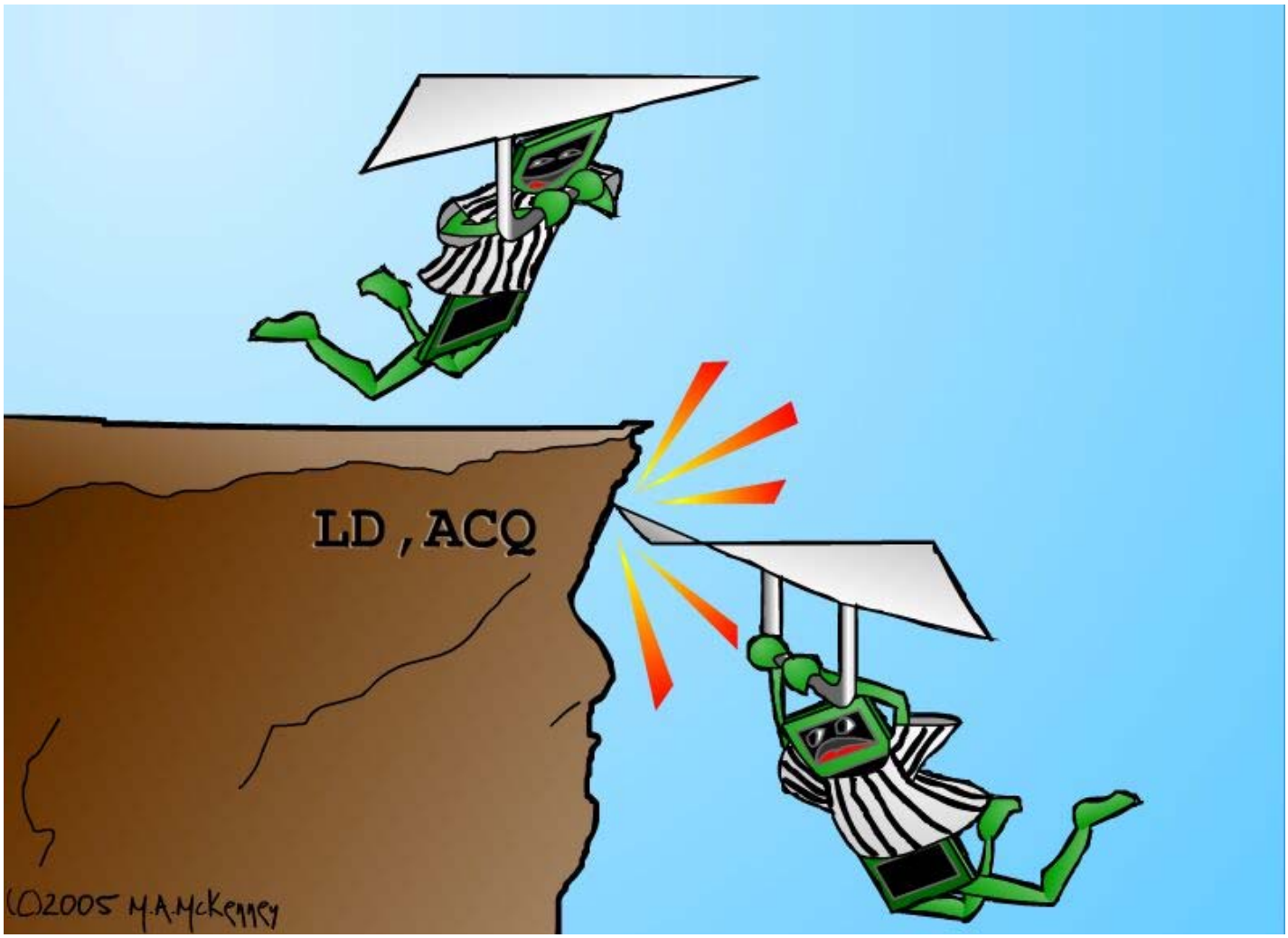

Figure 2. Half-Memory Barrier

Unfortunately, none of these instructions line up exactly with Linux's wmb() primitive, which requires all stores to be ordered. It does not require the other high-overhead actions of the sync instruction. But there is no choice: ppc64 versions of wmb() and mb() are defined to be the heavyweight sync instruction. However, Linux's smp_wmb() primitive cannot be used for MMIO, because a driver must carefully order MMIOs in UP as well as SMP kernels. So, it is defined to be the lighter-weight eieio instruction, which may be unique in having a five-vowel mnemonic. The smp_mb() primitive also is defined to be the sync instruction, but both smp_rmb() and rmb() are defined to be the lighter-weight lwsync instruction.

Many members of the POWER architecture have incoherent instruction caches, so a store to memory is not necessarily reflected in the instruction cache. Thankfully, few people write self-modifying code these days, but JITs do it all the time. Furthermore, recompiling a recently run program looks like self-modifying code from the CPU's viewpoint. The icbi instruction, instruction cache block invalidate, invalidates a specified cache line from the instruction cache and may be used in these situations.

SPARC RMO, PSO and TSO

Solaris on SPARC uses total-store order (TSO); however, Linux runs SPARC in relaxed-memory order (RMO) mode. The SPARC architecture also offers an intermediate partial-store order (PSO). Any program that runs in RMO also can run in either PSO or TSO. Similarly, a program that runs in PSO also can run in TSO. Moving a shared-memory parallel program in the other direction may require careful insertion of memory barriers; although, as noted earlier, programs that make standard use of synchronization primitives need not worry about memory barriers.

SPARC has a flexible memory-barrier instruction that permits fine-grained control of ordering:

- StoreStore: order preceding stores before subsequent stores. This option is used by the Linux smp_wmb() primitive.
- LoadStore: order preceding loads before subsequent stores.
- StoreLoad: order preceding stores before subsequent loads.
- LoadLoad: order preceding loads before subsequent loads. This option is used by the Linux smp_rmb() primitive.
- Sync: fully complete all preceding operations before starting any subsequent operations.
- MemIssue: complete preceding memory operations before subsequent memory operations, which is important for some instances of memory-mapped I/O.
- Lookaside: same as MemIssue but applies only to preceding stores and subsequent loads, and even then only for stores and loads that access the same memory location.

The Linux smp_mb() primitive uses the first four options together, as in:

```
membar #LoadLoad | #LoadStore | #StoreStore | #StoreLoad
```

This fully orders memory operations.

So, why is membar #MemIssue needed? Because a membar #StoreLoad could permit a subsequent load to get its value from a write buffer, which would be disastrous if the write goes to an MMIO register that induces side effects on the value to be read. In contrast, membar #MemIssue would wait until the write buffers were flushed before permitting the loads to execute, thereby ensuring that the load actually gets its value from the MMIO register. Drivers instead could use membar #Sync, but the lighter-weight membar #MemIssue is preferred in cases where the additional function of the more-expensive membar #Sync are not required.

The membar #Lookaside is a lighter-weight version of membar #MemIssue, which is useful when writing to a given MMIO register that affects the value read next from that same register. However, the heavier-weight membar #MemIssue must be used when a write to a given MMIO register affects the value read next from some other MMIO register.

It is not clear why SPARC does not define wmb() to be membar #MemIssue and smb_wmb() to be membar #StoreStore, as the current definitions seem vulnerable to bugs in some drivers. It is quite possible that all the SPARC CPUs that Linux runs on implement a more conservative memory-ordering model than the architecture would permit.

SPARC requires a flush instruction be used between the time that an instruction is stored and executed. This is needed to flush any prior value for that location from the SPARC's instruction cache. Notice that flush takes an address and flushes only that address from the instruction cache. On SMP systems, all CPUs' caches are flushed, but there is no convenient way to determine when the off-CPU flushes complete, although there is a reference to an implementation note.

x86

The x86 CPUs provide process ordering so that all CPUs agree on the order of a given CPU's writes to memory, so the smp_wmb() primitive is a no-op for the CPU. However, a compiler directive is required to prevent the compiler from performing optimizations that would result in reordering across the smp_wmb() primitive.

On the other hand, x86 CPUs give no ordering guarantees for loads, so the smp_mb() and smp_rmb() primitives expand to lock;addl. This atomic instruction acts as a barrier to both loads and stores. Some SSE instructions are ordered weakly; for example, clflush and nontemporal move instructions. CPUs that have SSE can use mfence for smp_mb(), lfence for smp_rmb() and sfence for smp_wmb(). A few versions of the x86 CPU have a mode bit that enables out-of-order stores, and for these CPUs, smp_wmb() also must be defined to be lock;addl.

Although many older x86 implementations accommodated self-modifying code without the need for any special instructions, newer revisions of the x86 architecture no longer require x86 CPUs to be so accommodating. Interestingly enough, this relaxation comes just in time to inconvenience JIT implementors.

zSeries

The zSeries machines make up the IBM mainframe family previously known as the 360, 370 and 390. Parallelism came late to zSeries, but given that these mainframes first shipped in the mid-1960s, this is not saying much. The bcr 15,0 instruction is used for the Linux smp_mb(), smp_rmb() and smp_wmb() primitives. It also has comparatively strong memory-ordering semantics, as shown in Table 1. This should allow the smp_wmb() primitive to be a no-op, and by the time you read this, this change may have happened.

As with most CPUs, the zSeries architecture does not guarantee a cache-coherent instruction stream. Hence, self-modifying code must execute a serializing instruction between updating the instructions and executing them. That said, many actual zSeries machines do in fact accommodate self-modifying code without serializing instructions. The zSeries instruction set provides a large set of serializing instructions, including compare-and-swap, some types of branches—for example, the aforementioned bcr 15,0 instruction—and test-and-set, among others.

Conclusion

This final installment of the memory-barrier series has given an overview of how a number of CPUs implement memory barriers. Although these overviews should by no means be considered a substitute for carefully reading the architecture manuals (see Resources), I hope that it has served as a useful introduction.

Acknowledgements

I owe thanks to many CPU architects for patiently explaining the instruction and memory-reordering features of their CPUs, particularly Wayne Cardoza, Ed Silha, Anton Blanchard, Tim Slegel, Juergen Probst, Ingo Adlung and Ravi Arimilli. Wayne deserves special thanks for his patience in explaining Alpha's reordering of dependent loads, a lesson that I resisted quite strenuously!

Legal Statement

Resources for this article: http://www.linuxjournal.com/article/8406.