Memory Ordering in Modern Microprocessors

Paul E. McKenney

Draft of 2007/01/25 17:32

Linux® has supported a large number of SMP systems based on a variety of CPUs since the 2.0 kernel. Linux has done an excellent job of abstracting away differences among these CPUs, even in kernel code. One important difference is how CPUs allow memory accesses to be reordered in SMP systems.

SMMP Hardware

Memory accesses are among the slowest of a CPU's operations, due to the fact that Moore's law has increased CPU instruction performance at a much greater rate than it has increased memory performance. This difference in performance increase means that memory operations have been getting increasingly expensive compared to simple register-to-register instructions. Modern CPUs sport increasingly large caches in order to reduce the overhead of these expensive memory accesses.

These caches can be thought of as simple hardware hash table with fixed size buckets and no chaining, as shown in Figure 1. This cache has sixteen "lines" and two "ways" for a total of 32 "entries", each entry containing a single 256-byte "cache line", which is a 256-byte-aligned block of memory. This cache line size is a little on the large size, but makes the hexadecimal arithmetic much simpler. In hardware parlance, this is a two-way set-associative cache, and is analogous to a software hash table with sixteen buckets, where each bucket's hash chain is limited to at most two elements. Since this cache is implemented in hardware, the hash function is extremely simple: extract four bits from the memory address.

In Figure 1, each box corresponds to a cache entry, which can contain a 256-byte cache line. However, a cache entry can be empty, as indicated by the empty boxes in the figure. The rest of the boxes are flagged with the memory address of the cache line that they contain. Since the cache lines must be 256-byte aligned, the low eight bits of each address are zero, and the choice of hardware hash function means that the next-higher four bits match the hash line number.

The situation depicted in the figure might arise if the program's code were located at address 0x43210E00 through 0x43210EFF, and this program accessed data sequentially from 0x12345000 through 0x12345EFF. Suppose that the program were now to access location

	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x43210E00
0xF		

Figure 1: CPU Cache Structure

0x12345F00. This location hashes to line 0xF, and both ways of this line are empty, so the corresponding 256-byte line can be accommodated. If the program were to access location 0x1233000, which hashes to line 0x0, the corresponding 256-byte cache line can be accommodated in way 1. However, if the program were to access location 0x1233E00, which hashes to line 0xE, one of the existing lines must be ejected from the cache to make room for the new cache line.

This background on hardware caching allows us to look at why CPUs reorder memory accesses.

Why Reorder Memory Accesses?

In a word, performance! CPUs have become so fast that the large multi-megabyte caches cannot keep up with them. Therefore, caches are often partitioned into nearly independent "banks", as shown in Figure 2, allowing each of the banks to run in parallel, thus better keeping up with the CPU. Memory is normally divided among the cache banks by address; for example, all the even-numbered cache lines might be processed by bank 0 and all of the odd-numbered cache lines by bank 1.

However, this hardware parallelism has a dark side: memory operations can now complete out of order, which can result in some confusion, as shown in Figure 3. CPU 0 might write first to location 0x12345000 (an even-numbered cache line) and then to location

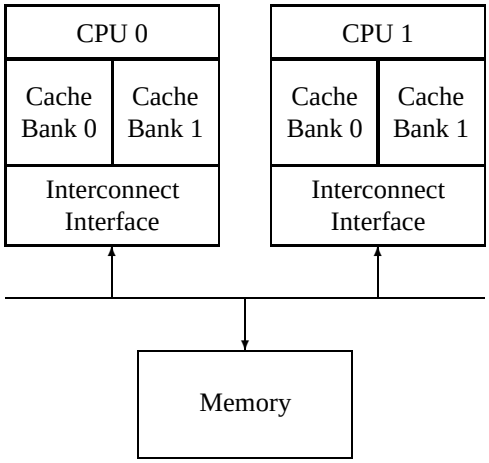

Figure 2: Hardware Parallelism

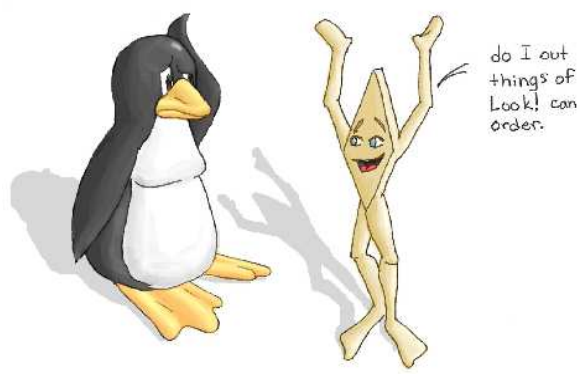

Figure 3: CPUs Can Do Things Out of Order

0x12345100 (an odd-numbered cache line). If bank 0 is busy with earlier requests but bank 1 is idle, the first write will be visible to CPU 1 *after* the second write, in other words, the writes will be perceived out of order by CPU 1. Reads can be reordered in a similar manner.

This reordering will cause many textbook parallel algorithms to fail.

Memory Reordering and SMP Software

A few machines offer "sequential consistency", in which all operations happen in the order specified by the code, and where all CPUs' views of these operations is consistent with a global ordering of the combined operations. Sequentially consistent systems have some very nice properties, but high performance has not tended to be one of them. The need for global ordering severely constrains the hardware's ability to exploit parallelism, and therefore commodity CPUs and systems do not offer sequential consistency.

On these systems, there are three orderings that must be accounted for:

1. **Program order**: the order that the memory operations are specified in the code running on a given CPU.
2. **Execution order**: the order that the individual memory-reference instructions are executed on a given CPU. The execution order can differ from program order due to both compiler and CPU-implementation optimizations.
3. **Perceived order**: the order that a given CPU perceives its and other CPUs' memory operations. The perceived order can differ from the execution order due to cachine, interconnect, and memory-system optimizations.

Popular memory-consistency models include x86's "process consistency", in which writes from a given CPU are seen in order by all CPUs, and weak consistency, which permits arbitrary reorderings, limited only by explicit memory-barrier instructions.

For more information on memory-consistency models, see Gharachorloo's exhaustive technical report [2].

Summary of Memory Ordering

When it comes to how memory ordering works on different CPUs, there is good news and bad news.

The bad news is that each CPU's memory ordering is a bit different. The good news is that there are a few things you can count on:

- A given CPU will always perceive its own memory operations as occurring in program order. That is, memory ordering issues arise only when a CPU is observing other CPUs' memory operations.
- An operation will be reordered with a store *only* if the operation accesses a different location than does the store.
- Aligned simple loads and stores are atomic.
- Linux-kernel synchronization primitives contain any needed memory barriers (which is a good reason to use these primitives!).

The most important differences are called out in Table 1. More detailed descriptions of specific CPUs' features are called out in following sections. The cells marked with a "Y" indicate weak memory ordering; the more "Y"s, the more reordering is possible. In general, it is easier to port SMP code from a CPU with many "Y"s to a CPU with fewer "Y"s, though your mileage may vary. However, note that code that uses standard synchronization primitives (spinlocks, semaphores, RCU) should not need explicit memory barriers, since any required barriers are already present in these primitives. Only "tricky" code that bypasses these synchronization primitives needs barriers, see Figure 6 for one example of such code. It is important to note that most

Table 1: Summary of Memory Ordering

	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
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries®				Y				Y

atomic operations (for example, `atomic_inc()` and `atomic_add()`) do *not* include any memory barriers.

The first four columns indicate whether a given CPU allows the four possible combinations of loads and stores to be reordered. The next two columns indicate whether a given CPU allows loads and stores to be reordered with atomic instructions. With only six CPUs, we have five different combinations of load-store reorderings, and three of the four possible atomic-instruction reorderings.

Parenthesized CPU names indicate modes that are architecturally allowed, but rarely used in practice.

The seventh column, dependent reads reordered, requires some explanation, which is undertaken in the following section covering Alpha CPUs. The short version is that Alpha requires memory barriers for readers as well as updaters of linked data structures. Yes, this does mean that Alpha can in effect fetch the data pointed to *before* it fetches the pointer itself, strange but true. Please see: `http://www.openvms.compaq.com/wizard/wiz_2637.html` if you think that I am just making this up. The benefit of this extremely weak memory model is that Alpha can use simpler cache hardware, which in turn permitted higher clock frequency in Alpha's heyday.

The last column indicates whether a given CPU has a incoherent instruction cache and pipeline. Such CPUs require special instructions be executed for self-modifying code. In absence of these instructions, the CPU might well execute the old rather than the new version of the code. This might seem unimportant–after all, who writes self-modifying code these days? The answer is that every JIT out there does. Writers of JIT code generators for such CPUs must take special care to flush instruction caches and pipelines before attempting to execute any newly generated code. These CPUs also require that the exec() and page-fault code flush the instruction caches and pipelines before attempting to execute any binaries just read into memory, lest the CPU end up executing the prior contents of the affected pages.

How Linux Copes

One of Linux's great advantages is that it runs on a wide variety of different CPUs. Unfortunately, as we have seen, these CPUs sport a wide variety of memory-consistency models. So what is a portable operating system to do?

Linux provides a carefully chosen set of memory-barrier primitives, which are as follows:

- `smp_mb()`: "memory barrier" that orders both loads and stores. This means that loads and stores preceding the memory barrier will be committed to memory before any loads and stores following the memory barrier.
- `smp_rmb()`: "read memory barrier" that orders only loads.
- `smp_wmb()`: "write memory barrier" that orders only stores.
- `smp_read_barrier_depends()` that forces subsequent operations that depend on prior operations to be ordered. This primitive is a no-op on all platforms except Alpha.
- `mmiowb()` that forces ordering on MMIO writes that are guarded by global spinlocks. This primitive is a no-op on all platforms on which the memory barriers in spinlocks already enforce MMIO ordering. The platforms with a non-no-op `mmiowb()` definition include some (but not all) IA64, FRV, MIPS, and SH systems. This primitive is relatively new, so relatively few drivers take advantage of it.

The `smp_mb()`, `smp_rmb()`, and `smp_wmb()` primitives also force the compiler to eschew any optimizations that would have the effect of reordering memory optimizations across the barriers. The `smp_read_barrier_depends()` primitive has a similar effect, but only on Alpha CPUs.

These primitives generate code only in SMP kernels, however, each also has a UP version (`smp_mb()`, tt smp_rmb(), `smp_wmb()`, and `smp_read_barrier_depends()`, respectively) that generate a memory barrier even in UP kernels. The `smp_` versions should be used in most cases. However, these latter primitives are useful when writing drivers, because MMIO accesses must remain ordered even in UP kernels. In absence of memory-barrier instructions, both CPUs and compilers would happily rearrange these accesses, which at best would make the device act strangely, and could crash your kernel or, in some cases, even damage your hardware.

So most kernel programmers need not worry about the memory-barrier peculiarities of each and every CPU, as long as they stick to these interfaces. If you are working deep in a given CPU's architecture-specific code, of course, all bets are off.

But it gets better. All of Linux's locking primitives (spinlocks, reader-writer locks, semaphores, RCU, ...) include any needed barrier primitives. So if you are working with code that uses these primitives, you don't even need to worry about Linux's memory-ordering primitives.

That said, deep knowledge of each CPU's memory-consistency model can be very helpful when debugging, to say nothing of when writing architecture-specific code or synchronization primitives.

Besides, they say that a little knowledge is a very dangerous thing. Just imagine the damage you could do with a lot of knowledge! For those who wish to understand more about individual CPUs' memory consistency models, the next sections describes those of the most popular and prominent CPUs. Although nothing can replace actually reading a given CPU's documentation, these sections give a good overview.

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, which must work on all CPUs, including Alpha. Understanding Alpha is therefore surprisingly important to the Linux kernel hacker.

The difference between Alpha and the other CPUs is illustrated by the code shown in Figure 4. This `smp_wmb()` on line 9 of this figure 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 will work correctly. That is, it makes this guarantee on all CPUs *except* Alpha.

Alpha has extremely weak memory ordering such that

```
 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 }
```

Figure 4: Insert and Lock-Free Search

the code on line 20 of Figure 4 could see the old garbage values that were present before the initialization on lines 6-8.

Figure 5 shows how this can happen on an aggressively parallel machine with partitioned caches, so that alternating caches lines are processed by the different partitions of the caches. Assume that the list header `head` will be processed by cache bank 0, and that the new element will be processed by cache bank 1. On Alpha, the `smp_wmb()` will guarantee that the cache invalidates performed by lines 6-8 of Figure 4 will reach the interconnect before that of line 10 does, but makes absolutely no guarantee about the order in which the new values will reach the reading CPU's core. For example, it is possible that the reading CPU's cache bank 1 is very busy, but 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. See the Web site called out earlier for more information, or, again, if you think that I am just making all this up.

One could place an `smp_rmb()` primitive between the pointer fetch and dereference. However, this imposes unneeded overhead on systems (such as i386, IA64, PPC, and SPARC) that respect data dependencies on the read side. A `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 Figure 6.

It is also 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

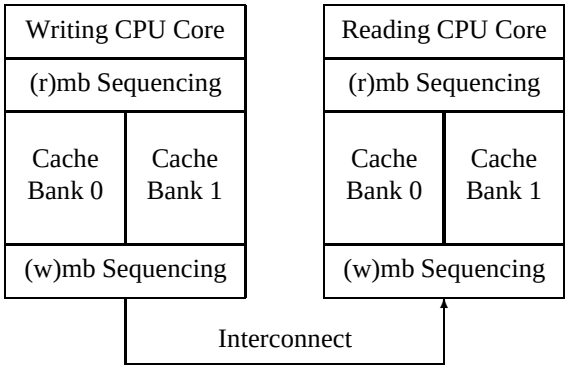

Figure 5: Why smp_read_barrier_depends() is Required

```
 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 }
```

Figure 6: Safe Insert and Lock-Free Search

community to impose excessive overhead on extremely weakly ordered CPUs such as Alpha. This software barrier could be implemented by sending inter-processor interrupts (IPIs) to all other CPUs. Upon receipt of such an IPI, a CPU would execute a memory-barrier instruction, implementing a memory-barrier shootdown. Additional logic is required to avoid deadlocks. Of course, CPUs that respect data dependencies would define such a barrier to simply be `smp_wmb()`. Perhaps this decision should be revisited in the future as Alpha fades off into the sunset.

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.

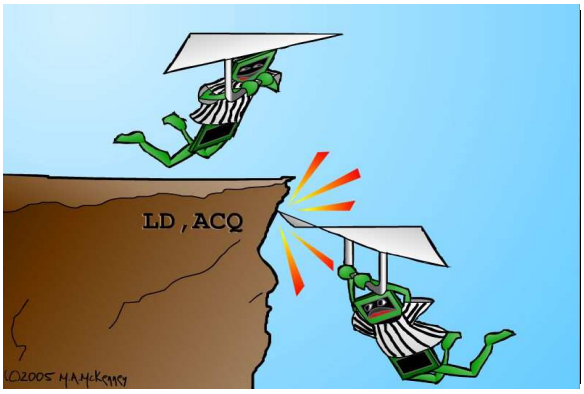

Figure 7: Half Memory Barrier

For more detail on Alpha, see the reference manual [11].

AMD64

Although AMD64 is compatible with x86, it offers a stronger slightly memory-consistency model, in that it will not reorder a store ahead of a load [1]. 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 will work on some x86 CPUs but fail 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`.

IA64

IA64 offers a weak consistency model, so that in absence of explicit memory-barrier instructions, IA64 is within its rights to arbitrarily reorder memory references [5]. IA64 has a memory-fence instruction named `mf`, but also has "half-memory fence" modifiers to loads, stores, and to some of its atomic instructions [4]. The `acq` modifier prevents subsequent memory-reference instructions from being reordered before the `acq`, but permits prior memory-reference instructions to be reordered after the `acq`, as fancifully illustrated by Figure 7. Similarly, the `rel` modifier prevents prior memory-reference instructions from being reordered after the `rel`, but allows subsequent memory-reference instructions to be reordered before the `rel`.

These half-memory fences are useful for critical sections, since it is safe to push operations into a critical section, but can be fatal 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 rumors to the contrary, the "mf" mneumonic 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 [9]. This means that the Linux kernel's memory-ordering primitives generate no code, however, they do use the gcc `memory` attribute to disable compiler optimizations that would reorder code across the memory barrier.

POWER

The POWER and Power PC® CPU families have a wide variety of memory-barrier instructions [3, 10]:

1. `sync` causes all preceding operations to *appear to have* completed before any subsequent operations are started. This instruction is therefore quite expensive.
2. `lwsync` (light-weight sync) orders loads with respect to subsequent loads and stores, and 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 zSeries, and coincidentally, SPARC TSO.
3. `eieio` (enforce in-order execution of I/O, in case you were wondering) causes all preceding cacheable stores to appear to have completed before all subsequent stores. However, stores to cacheable memory are ordered separately from stores to non-cacheable memory, which means that `eieio` will not force an MMIO store to precede a spinlock release.
4. `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 have either happened or are guaranteed not to happen, and that any side-effects of these instructions (for example, page-table changes) are seen by the subsequent instructions.

Unfortunately, none of these instructions line up exactly with Linux's `wmb()` primitive, which requires *all* stores to be ordered, but 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()` instruction is never used for MMIO (since a driver must carefully order MMIOs in UP as well as SMP kernels, after all), so it is defined to be the lighter weight `eieio` instruction. This instruction may well be unique in having a five-vowel mneumonic, which stands for "enforce in-order execution of I/O". The `smp_mb()` instruction is also 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 that a store to memory will not necessarily be reflected in the instruction cache. Thankfully, few people write self-modifying code these days, but JITs and compilers do it all the time. Furthermore, recompiling a recently run program looks just 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 TSO (total-store order), as does Linux when built for the "sparc" 32-bit architecture. However, a 64-bit Linux kernel (the "sparc64" architecture) runs SPARC in RMO (relaxed-memory order) mode [12]. The SPARC architecture also offers an intermediate PSO (partial store order). Any program that runs in RMO will also run in either PSO or TSO, and similarly, a program that runs in PSO will also 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 very flexible memory-barrier instruction [12] 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, important for some instances of memory-mapped I/O.

`Lookaside`: same as MemIssue, but only applies 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, thus fully ordering 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 was to an MMIO register that induced 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 could instead 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 affects the value that will next be read from that register. However, the heavier-weight membar #MemIssue must be used when a write to a given MMIO register affects the value that will next be read 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 [12]. This is needed to flush any prior value for that location from the SPARC's instruction cache. Note that flush takes an address, and will flush 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, though there is a reference to an implementation note.

x86

Since the x86 CPUs provide "process ordering" so that all CPUs agree on the order of a given CPU's writes to memory, the smp_wmb() primitive is a no-op for the CPU [7]. 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. Note that some SSE instructions are weakly ordered (clflush and non-temporal move instructions [6]). 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() must also 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 requires 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 [8]. 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, which should allow the smp_wmb() primitive to be a nop (and by the time you read this, this change may well have happened). The table actually understates the situation, as the zSeries memory model is otherwise sequentially consistent, meaning that all CPUs will agree on the order of unrelated stores from different CPUs.

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.

Conclusions

As noted earlier, the good news is that Linux's memory-ordering primitives and synchronization primitives make it unnecessary for most Linux kernel hackers to worry about memory barriers. This is especially good news given the large number of CPUs and systems that Linux supports, and the resulting wide variety of memory-consistency models. However, there are times when knowing about memory barriers can be helpful, and I hope that this article has served as a good introduction to them.

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

References

[1] ADVANCED MICRO DEVICES. *AMD x86-64 Architecture Programmer's Manual Volumes 1-5*, 2002.

[2] GHRACHORLOO, K. Memory consistency models for shared-memory multiprocessors. Tech. Rep. CSL-TR-95-685, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA, December 1995. Available: `http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-9.pdf` [Viewed: October 11, 2004].

[3] IBM MICROELECTRONICS AND MOTOROLA. *PowerPC Microprocessor Family: The Programming Environments*, 1994.

[4] INTEL CORPORATION. *Intel Itanium Architecture Software Developer's Manual Volume 3: Instruction Set Reference*, 2002.

[5] INTEL CORPORATION. *Intel Itanium Architecture Software Developer's Manual Volume 3: System Architecture*, 2002.

[6] INTEL CORPORATION. *IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, 2004. Available: `ftp://download.intel.com/design/Pentium4/manuals/25366714.pdf` [Viewed: February 16, 2005].

[7] INTEL CORPORATION. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2004. Available: `ftp://download.intel.com/design/Pentium4/manuals/25366814.pdf` [Viewed: February 16, 2005].

[8] INTERNATIONAL BUSINESS MACHINES CORPORATION. z/Architecture principles of operation. Available: `http://publibz.boulder.ibm.com/epubs/pdf/dz9zr003.pdf` [Viewed: February 16, 2005], May 2004.

[9] KANE, G. *PA-RISC 2.0 Architecture*. Hewlett-Packard Professional Books, 1996.

[10] LYONS, M., SILHA, E., AND HAY, B. PowerPC storage model and AIX programming. Available: `http://www-106.ibm.com/developerworks/eserver/articles/powerpc.html` [Viewed: January 31, 2005], August 2002.

[11] SITES, R. L., AND WITEK, R. T. *Alpha AXP Architecture*, second ed. Digital Press, 1995.

[12] SPARC INTERNATIONAL. *The SPARC Architecture Manual*, 1994.