Memory Ordering in Modern Microprocessors, Part I

Jun 30, 2005 By Paul E. McKenney (/user/1001349)

in

One important difference among CPU families is how they allow memory accesses to be reordered. Linux has to support them all.

(/issue/136)

From Issue #136 August 2005 (/issue/136)

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

1. A given CPU always perceives its own memory operations as occurring in program order. That is, memory-reordering issues arise only when a CPU is observing other CPUs' memory operations.
2. An operation is reordered with a store only if the operation accesses a different location than does the store.
3. Aligned simple loads and stores are atomic.
4. 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 will be addressed in a later installment. Parenthesized CPU names indicate modes that are allowed architecturally but rarely used in practice. The cells marked with a Y indicate weak memory ordering; the more Ys, the more reordering is possible. In general, it is easier to port SMP code from a CPU with many Ys to a CPU with fewer Ys, though your mileage may vary. However, code that uses standard synchronization primitives—spinlocks, semaphores, RCU—should not need explicit memory barriers, because any required barriers already are present in these primitives. Only tricky code that bypasses these synchronization primitives needs barriers. It is important to note that most atomic operations, for example, atomic_inc() and atomic_add(), do not include any memory barriers.

In Table 1, 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 eight CPUs, we have five different combinations of load-store reorderings and three of the four possible atomic-instruction reorderings.

	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

(/files/linuxjournal.com/linuxjournal/articles/082/8211/8211t1.jpg)

Table 1. Summary of Memory Ordering

The second-to-last column, dependent reads reordered, requires some explanation, which will be undertaken in the second installment of this series. The short version is Alpha requires memory barriers for readers as well as for updaters of linked data structures. Yes, this does mean that Alpha in effect can fetch the data pointed to before it fetches the pointer itself—strange but true. Please see the “Ask the Wizard” column on the manufacturer's site, listed in Resources, if you think that I am making this up. The benefit of this extremely weak memory model is Alpha can use simpler cache hardware, which in turn permitted higher clock frequencies in Alpha's heyday.

The last column in Table 1 indicates whether a given CPU has a incoherent instruction cache and pipeline. Such CPUs require that special instructions be executed for self-modifying code. In absence of these instructions, the CPU might 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 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 kernel to do?

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

- smp_mb(): “memory barrier” that orders both loads and stores. This means loads and stores preceding the memory barrier are 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(): forces subsequent operations that depend on prior operations to be ordered. This primitive is a no-op on all platforms except Alpha.

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 must do the same, but only on Alpha CPUs.

These primitives generate code only in SMP kernels; however, each also has a UP

version—mb(), rmb(), wmb() and 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 memory-mapped I/O accesses must remain ordered even in UP kernels. In absence of memory-barrier instructions, both CPUs and compilers happily would rearrange these accesses. At best, this would make the device act strangely; at worst, it would 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 memory-barrier 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, including spinlocks, reader-writer locks, semaphores and read-copy updates (RCUs), 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 helpful when debugging, to say nothing of writing architecture-specific code or synchronization primitives.

Besides, they say a little knowledge is a dangerous thing. Just imagine the damage you could do with a lot of knowledge! For those who want to understand more about individual CPUs' memory consistency models, the next installment will describe those of the most popular and prominent CPUs.

_____

Comments

Comment viewing options

Threaded list - expanded ▼ | Date - newest first ▼ | 50 comments per page ▼ | Save settings

Select your preferred way to display the comments and click "Save settings" to activate your changes.

First article I've seen on (/article/8211#comment-355160)
Submitted by RossC (not verified) on Sun, 08/22/2010 - 16:31.

First article I've seen on CPU reordering that explained *why* it happens. Great stuff.

Interesting post for anyone reading this article (/article/8211#comment-347240)
Submitted by Sudhanshu (http://www.42klines.com) (not verified) on Wed, 01/06/2010 - 18:27.

http://timetobleed.com/mysql-doesnt-always-suck-this-time-its-amd/ (http://timetobleed.com/mysql-doesnt-always-suck-this-time-its-amd/)

More details (/article/8211#comment-347073)
Submitted by Sudhanshu (http://www.42klines.com) (not verified) on Fri, 01/01/2010 - 15:29.

http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2009.04.05a.pdf (http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2009.04.05a.pdf)
Great article. Thanks Paul.