Memory Ordering in Modern Microprocessors, Part I

Paul E. McKenney

Abstract

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

Since the 2.0 kernel release, Linux has supported a large number of SMP systems based on a variety of CPUs. Linux has done an excellent job of abstracting differences among these CPUs, even in kernel code. This article is an overview of one important difference: how CPUs allow memory accesses to be reordered in SMP systems.

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 tables with fixed size buckets and no chaining, as shown in Figure 1. This cache has 16 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 it makes the hexadecimal arithmetic much simpler. In hardware parlance, this is a two-way set-associative cache. It is analogous to a software hash table with 16 buckets, where each bucket's hash chain is limited to two elements at most. Because 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 that 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 they contain. Because the cache lines must be 256-byte aligned, the low eight bits of each address are zero. The choice of hardware hash function means the next-higher four bits match the line number.

The situation depicted in Figure 1 might arise if the program's code was located at address 0x43210E00 through 0x43210EFF, and this program accessed data sequentially from 0x12345000 through 0x12345EFF. Suppose that the program now was to access location 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 was 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.

	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 for a Cache with 16 Lines and Two Entries Per Line

Why Reorder Memory Accesses?

In a word, performance! CPUs have become so fast that the large multimegabyte caches cannot keep up with them. Therefore, caches often are partitioned into nearly independent banks, as shown in Figure 2. This allows each of the banks to run in parallel, thus keeping up better with the CPU. Memory normally is 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 now can complete out of order, which can result in some confusion, as illustrated in Figure 3. CPU 0 might write first to location 0x12345000, an even-numbered cache line, and then to location 0x12345100, an odd-numbered cache line. If bank 0 is busy with earlier requests but bank 1 is idle, the first write is visible to CPU 1 after the second write. In other words, the writes are perceived out of order by CPU 1. Reads can be reordered in a similar manner. This reordering can cause many textbook parallel algorithms to fail.

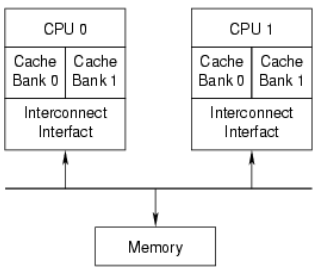

Figure 2. Hardware parallelism divides one large cache into multiple banks.

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 are consistent with a global ordering of the combined operations. Sequentially consistent systems have some nice properties, but high performance does not tend 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, three orderings must be accounted for:

1. Program order: the order in which the memory operations are specified in the code running on a given CPU.
2. Execution order: the order in which 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 in which a given CPU perceives its and other CPUs' memory operations. The perceived order can differ from the execution order due to caching, interconnect and memory-system optimizations. Different CPUs might well perceive the same memory operations as occurring in different orders.

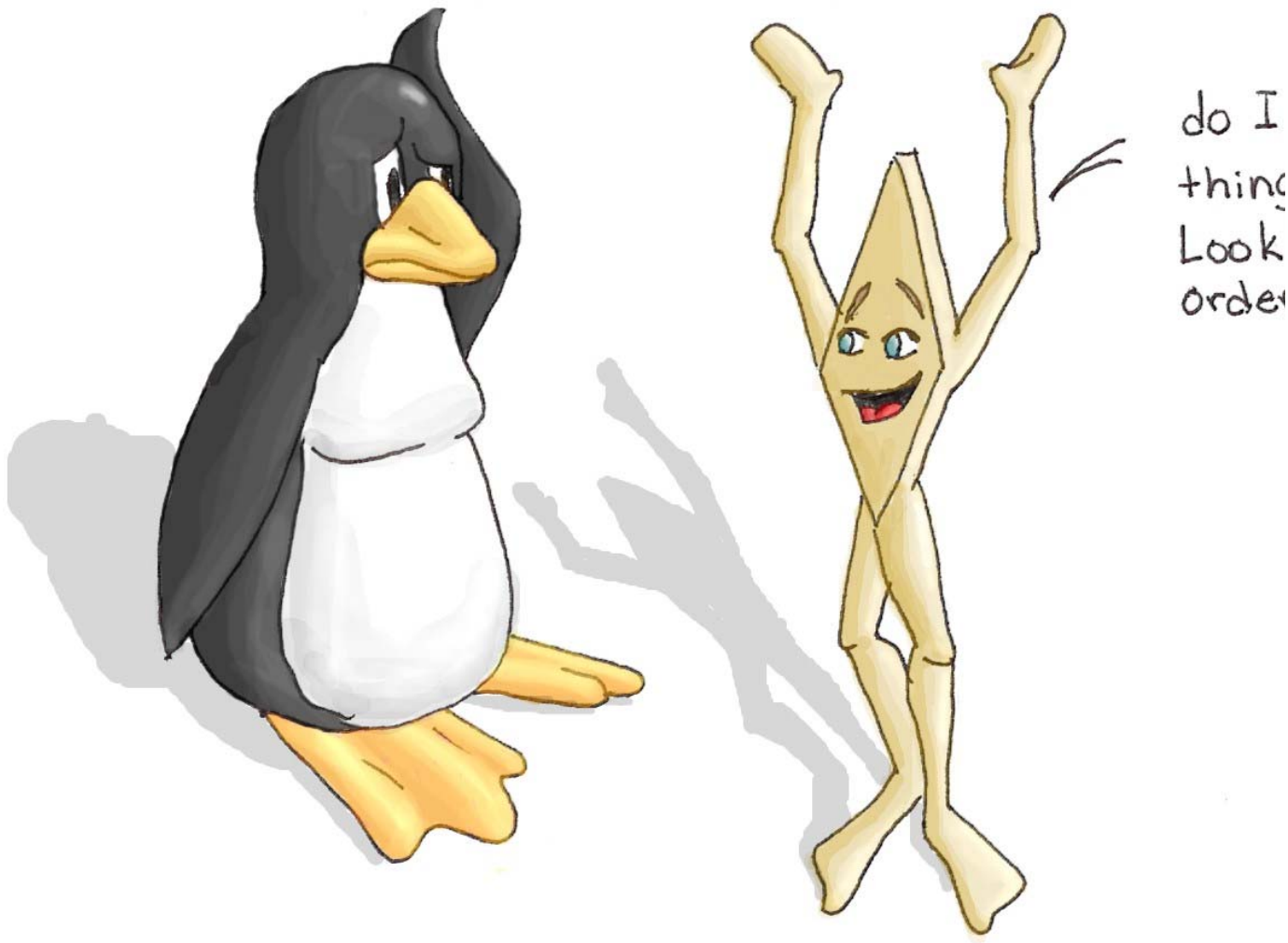

Figure 3. CPUs can do things out of order.

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, listed in the on-line Resources.

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

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.

Conclusions

As noted earlier, the good news is 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 learning quite strenuously!

Legal Statement

Resources for this article: http://www.linuxjournal.com/article/8331.