Memory Barriers: a Hardware View for Software Hackers

Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@us.ibm.com

April 5, 2009

So what possessed CPU designers to cause them to inflict memory barriers on poor unsuspecting SMP software designers?

In short, because reordering memory references allows much better performance, and so memory barriers are needed to force ordering in things like synchronization primitives whose correct operation depends on ordered memory references.

Getting a more detailed answer to this question requires a good understanding of how CPU caches work, and especially what is required to make caches really work well. The following sections:

1. present the structure of a cache,
2. describe how cache-coherency protocols ensure that CPUs agree on the value of each location in memory, and, finally,
3. outline how store buffers and invalidate queues help caches and cache-coherency protocols achieve high performance.

We will see that memory barriers are a necessary evil that is required to enable good performance and scalability, an evil that stems from the fact that CPUs are orders of magnitude faster than are both the interconnects between them and the memory they are attempting to access.

1 Cache Structure

Modern CPUs are much faster than are modern memory systems. A 2006 CPU might be capable of executing ten instructions per nanosecond, but will require many tens of nanoseconds to fetch a data item from main memory. This disparity in speed — more than two orders of magnitude — has resulted in the multi-megabyte caches found on modern CPUs. These caches are associated with the CPUs as shown in Figure 1, and can typically be accessed in a few cycles.[1]

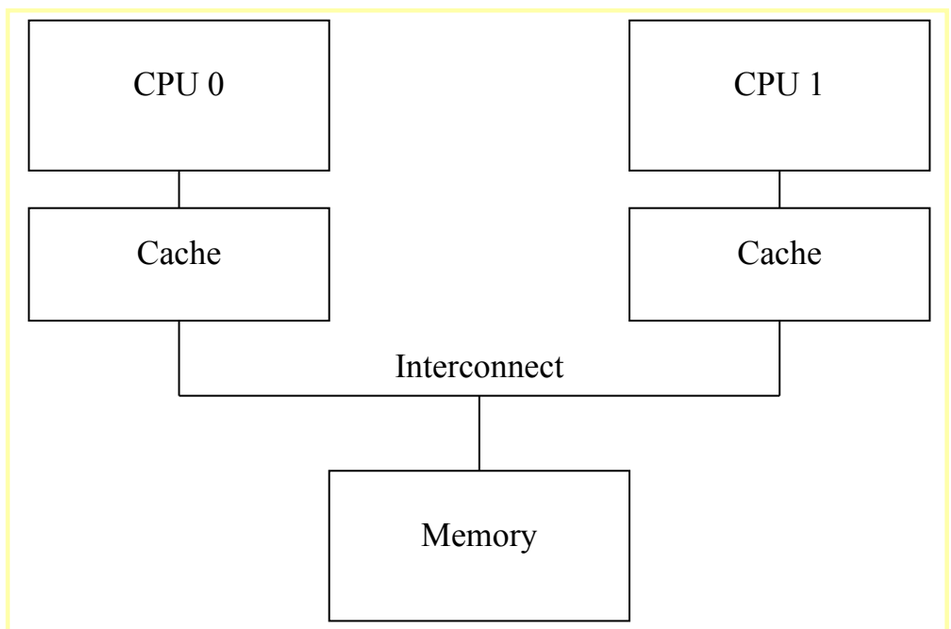

Figure 1: Modern Computer System Cache Structure

Data flows among the CPUs' caches and memory in fixed-length blocks called "cache lines", which are normally a power of two in size, ranging from 16 to 256 bytes. When a given data item is first accessed by

[1] It is standard practice to use multiple levels of cache, with a small level-one cache close to the CPU with single-cycle access time, and a larger level-two cache with a longer access time, perhaps roughly ten clock cycles. Higher-performance CPUs often have three or even four levels of cache.

a given CPU, it will be absent from that CPU's cache, meaning that a "cache miss" (or, more specifically, a "startup" or "warmup" cache miss) has occurred. The cache miss means that the CPU will have to wait (or be "stalled") for hundreds of cycles while the item is fetched from memory. However, the item will be loaded into that CPU's cache, so that subsequent accesses will find it in the cache and therefore run at full speed.

After some time, the CPU's cache will fill, and subsequent misses will likely need to eject an item from the cache in order to make room for the newly fetched item. Such a cache miss is termed a "capacity miss", because it is caused by the cache's limited capacity. However, most caches can be forced to eject an old item to make room for a new item even when they are not yet full. This is due to the fact that large caches are implemented as hardware hash tables with fixed-size hash buckets (or "sets", as CPU designers call them) and no chaining, as shown in Figure 2.

This cache has sixteen "sets" and two "ways" for a total of 32 "lines", 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. The size (32 cache lines in this case) and the associativity (two in this case) are collectively called the cache's "geometry". Since this cache is implemented in hardware, the hash function is extremely simple: extract four bits from the memory address.

In Figure 2, 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

	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 2: CPU Cache Structure

0x43210E00 through 0x43210EFF, and this program accessed data sequentially from 0x12345000 through 0x12345EFF. Suppose that the program were now 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 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. If this ejected line were accessed later, a cache miss would result. Such a cache miss is termed an "associativity miss".

Thus far, we have been considering only cases where a CPU reads a data item. What happens when it does a write? Because it is important that all CPUs agree on the value of a given data item, before a given CPU writes to that data item, it must first cause it to be removed, or "invalidated", from other CPUs' caches. Once this invalidation has completed, the CPU may safely modify the data item. If the data item was present in this CPU's cache, but was read-only, this process is termed a "write miss". Once a given CPU has completed invalidating a given data item from other CPUs' caches, that CPU may repeat-

edly write (and read) that data item.

Later, if one of the other CPUs attempts to access the data item, it will incur a cache miss, this time because the first CPU invalidated the item in order to write to it. This type of cache miss is termed a "communication miss", since it is usually due to several CPUs using the data items to communicate (for example, a lock is a data item that is used to communicate among CPUs using a mutual-exclusion algorithm).

Clearly, much care must be taken to ensure that all CPUs maintain a coherent view of the data. With all this fetching, invalidating, and writing, it is easy to imagine data being lost or (perhaps worse) different CPUs having conflicting values for the same data item in their respective caches. These problems are prevented by "cache-coherency protocols", described in the next section.

2 Cache-Coherence Protocols

Cache-coherency protocols manage cache-line states so as to prevent inconsistent or lost data. These protocols can be quite complex, with many tens of states,[2] but for our purposes we need only concern ourselves with the four-state MESI cache-coherence protocol.

2.1 MESI States

MESI stands for "modified", "exclusive", "shared", and "invalid", the four states a given cache line can take on using this protocol. Caches using this protocol therefore maintain a two-bit state "tag" on each cache line in addition to that line's physical address and data.

A line in the "modified" state has been subject to a recent memory store from the corresponding CPU, and the corresponding memory is guaranteed not to appear in any other CPU's cache. Cache lines in the "modified" state can thus be said to be "owned" by the CPU. Because this cache holds the only up-to-date copy of the data, this cache is ultimately responsible for either writing it back to memory or handing it off to some other cache, and must do so before reusing this line to hold other data.

The "exclusive" state is very similar to the "modified" state, the single exception being that the cache line has not yet been modified by the corresponding CPU, which in turn means that the copy of the cache line's data that resides in memory is up-to-date. However, since the CPU can store to this line at any time, without consulting other CPUs, a line in the "exclusive" state can still be said to be owned by the corresponding CPU. That said, because the corresponding value in memory is up to date, this cache can discard this data without writing it back to memory or handing it off to some other CPU.

A line in the "shared" state might be replicated in at least one other CPU's cache, so that this CPU is not permitted to store to the line without first consulting with other CPUs. As with the "exclusive" state, because the corresponding value in memory is up to date, this cache can discard this data without writing it back to memory or handing it off to some other CPU

A line in the "invalid" state is empty, in other words, it holds no data. When new data enters the cache, it is placed into a cache line that was in the "invalid" state if possible. This approach is preferred because replacing a line in any other state could result in an expensive cache miss should the replaced line be referenced in the future.

Since all CPUs must maintain a coherent view of the data carried in the cache lines, the cache-coherence protocol provides messages that coordinate the movement of cache lines through the system.

2.2 MESI Protocol Messages

Many of the transitions described in the previous section require communication among the CPUs. If the CPUs are on a single shared bus, the following messages suffice:

Read: The "read" message contains the physical address of the cache line to be read.

[2] See Culler et al. [3] pages 670 and 671 for the nine-state and 26-state diagrams for SGI Origin2000 and Sequent (now IBM) NUMA-Q, respectively. Both diagrams are significantly simpler than real life.

Read Response: The "read response" message contains the data requested by an earlier "read" message. This "read response" message might be supplied either by memory or by one of the other caches. For example, if one of the caches has the desired data in "modified" state, that cache must supply the "read response" message.

Invalidate: The "invalidate" message contains the physical address of the cache line to be invalidated. All other caches must remove the corresponding data from their caches and respond.

Invalidate Acknowledge: A CPU receiving an "invalidate" message must respond with an "invalidate acknowledge" message after removing the specified data from its cache.

Read Invalidate: The "read invalidate" message contains the physical address of the cache line to be read, while at the same time directing other caches to remove the data. Hence, it is a combination of a "read" and an "invalidate", as indicated by its name. A "read invalidate" message requires both a "read response" and a set of "invalidate acknowledge" messages in reply.

Writeback: The "writeback" message contains both the address and the data to be written back to memory (and perhaps "snooped" into other CPUs' caches along the way). This message permits caches to eject lines in the "modified" state as needed to make room for other data.

Interestingly enough, a shared-memory multiprocessor system really is a message-passing computer under the covers. This means that clusters of SMP machines that use distributed shared memory are using message passing to implement shared memory at two different levels of the system architecture.

Quick Quiz 1: What happens if two CPUs attempt to invalidate the same cache line concurrently? □

Quick Quiz 2: When an "invalidate" message appears in a large multiprocessor, every CPU must give an "invalidate acknowledge" response. Wouldn't the resulting "storm" of "invalidate acknowledge" responses totally saturate the system bus? □

Quick Quiz 3: If SMP machines are really using message passing anyway, why bother with SMP at all? □

2.3 MESI State Diagram

A given cache line's state changes as protocol messages are sent and received, as shown in Figure 3.

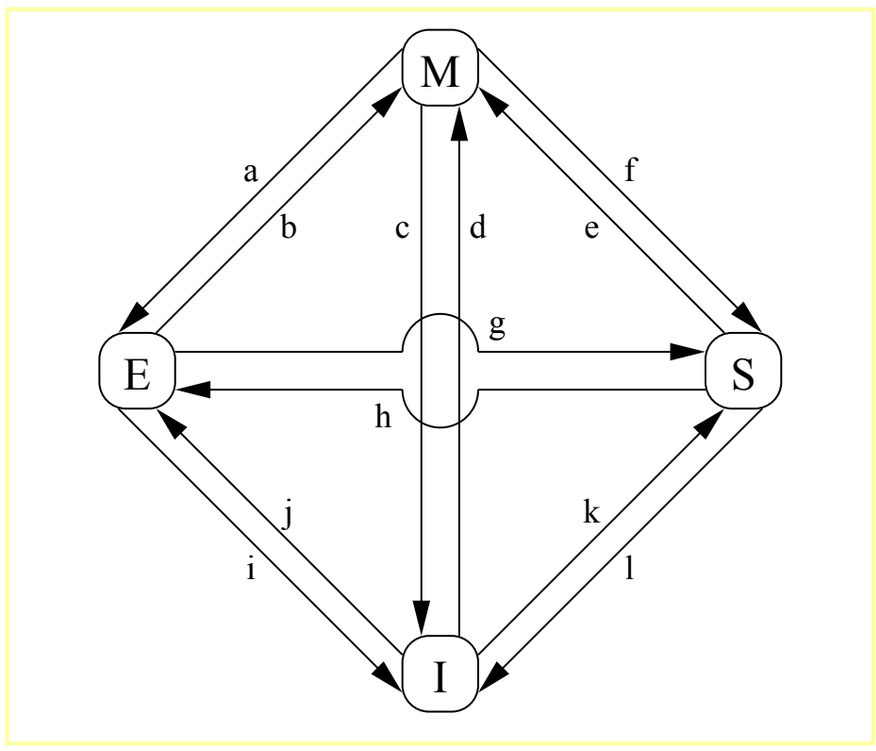

Figure 3: MESI Cache-Coherency State Diagram

The transition arcs in this figure are as follows:

Transition (a): A cache line is written back to memory, but the CPU retains it in its cache and further retains the right to modify it. This transition requires a "writeback" message.

Transition (b): The CPU writes to the cache line that it already had exclusive access to. This transition does not require any messages to be sent or received.

Transition (c): The CPU receives a "read invalidate" message for a cache line that it has modified. The CPU must invalidate its local copy, then respond with both a "read response" and an "invalidate acknowledge" message, both sending the data to the requesting CPU and indicating that it no longer has a local copy.

Transition (d): The CPU does an atomic read-modify-write operation on a data item that was not present in its cache. It transmits a "read invalidate", receiving the data via a "read response". The CPU can complete the transition once it has also received a full set of "invalidate acknowledge" responses.

Transition (e): The CPU does an atomic read-modify-write operation on a data item that was previously read-only in its cache. It must transmit "invalidate" messages, and must wait for a full set of "invalidate acknowledge" responses before completing the transition.

Transition (f): Some other CPU reads the cache line, and it is supplied from this CPU's cache, which retains a read-only copy. This transition is initiated by the reception of a "read" message, and this CPU responds with a "read response" message containing the requested data.

Transition (g): Some other CPU reads a data item in this cache line, and it is supplied either from this CPU's cache or from memory. In either case, this CPU retains a read-only copy. This transition is initiated by the reception of a "read" message, and this CPU responds with a "read response" message containing the requested data.

Transition (h): This CPU realizes that it will soon need to write to some data item in this cache line, and thus transmits an "invalidate" message. The CPU cannot complete the transition until it receives a full set of "invalidate acknowledge" responses. Alternatively, all other CPUs eject this cache line from their caches via "writeback" messages (presumably to make room for other cache lines), so that this CPU is the last CPU caching it.

Transition (i): Some other CPU does an atomic read-modify-write operation on a data item in a cache line held only in this CPU's cache, so this CPU invalidates it from its cache. This transition is initiated by the reception of a "read invalidate" message, and this CPU responds with both a "read response" and an "invalidate acknowledge" message.

Transition (j): This CPU does a store to a data item in a cache line that was not in its cache, and thus transmits a "read invalidate" message. The CPU cannot complete the transition until it receives the "read response" and a full set of "invalidate acknowledge" messages. The cache line will presumably transition to "modified" state via transition (b) as soon as the actual store completes.

Transition (k): This CPU loads a data item in a cache line that was not in its cache. The CPU transmits a "read" message, and completes the transition upon receiving the corresponding "read response".

Transition (l): Some other CPU does a store to a data item in this cache line, but holds this cache line in read-only state due to its being held in other CPUs' caches (such as the current CPU's cache). This transition is initiated by the reception of an "invalidate" message, and this CPU responds with an "invalidate acknowledge" message.

Quick Quiz 4: How does the hardware handle the delayed transitions described above? □

2.4 MESI Protocol Example

Let's now look at this from the perspective of a cache line's worth of data, initially residing in memory at address 0, as it travels through the various single-line direct-mapped caches in a four-CPU system. Table 1 shows this flow of data, with the first column showing the sequence of operations, the second the CPU performing the operation, the third the operation being performed, the next four the state of each CPU's cache line (memory address followed by MESI state), and the final two columns whether the corresponding memory contents are up to date ("V") or not ("I").

Initially, the CPU cache lines in which the data would reside are in the "invalid" state, and the data is valid in memory. When CPU 0 loads the data at

address 0, it enters the "shared" state in CPU 0's cache, and is still valid in memory. CPU 3 also loads the data at address 0, so that it is in the "shared" state in both CPUs' caches, and is still valid in memory. Next CPU 3 loads some other cache line (at address 8), which forces the data at address 0 out of its cache via a writeback, replacing it with the data at address 8. CPU 2 now does a load from address 0, but this CPU realizes that it will soon need to store to it, and so it uses a "read invalidate" message in order to gain an exclusive copy, invalidating it from CPU 3's cache (though the copy in memory remains up to date). Next CPU 2 does its anticipated store, changing the state to "modified". The copy of the data in memory is now out of date. CPU 1 does an atomic increment, using a "read invalidate" to snoop the data from CPU 2's cache and invalidate it, so that the copy in CPU 1's cache is in the "modified" state (and the copy in memory remains out of date). Finally, CPU 1 reads the cache line at address 8, which uses a "writeback" message to push address 0's data back out to memory.

Note that we end with data in some of the CPU's caches.

Quick Quiz 5: What sequence of operations would put the CPUs' caches all back into the "invalid" state? □

3 Stores Result in Unnecessary Stalls

Although the cache structure shown in Figure 1 provides good performance for repeated reads and writes from a given CPU to a given item of data, its performance for the first write to a given cache line is quite poor. To see this, consider Figure 4, which shows a timeline of a write by CPU 0 to a cacheline held in CPU 1's cache. Since CPU 0 must wait for the cache line to arrive before it can write to it, CPU 0 must stall for an extended period of time.[3]

[3]The time required to transfer a cache line from one CPU's cache to another's is typically a few orders of magnitude more than that required to execute a simple register-to-register instruction.

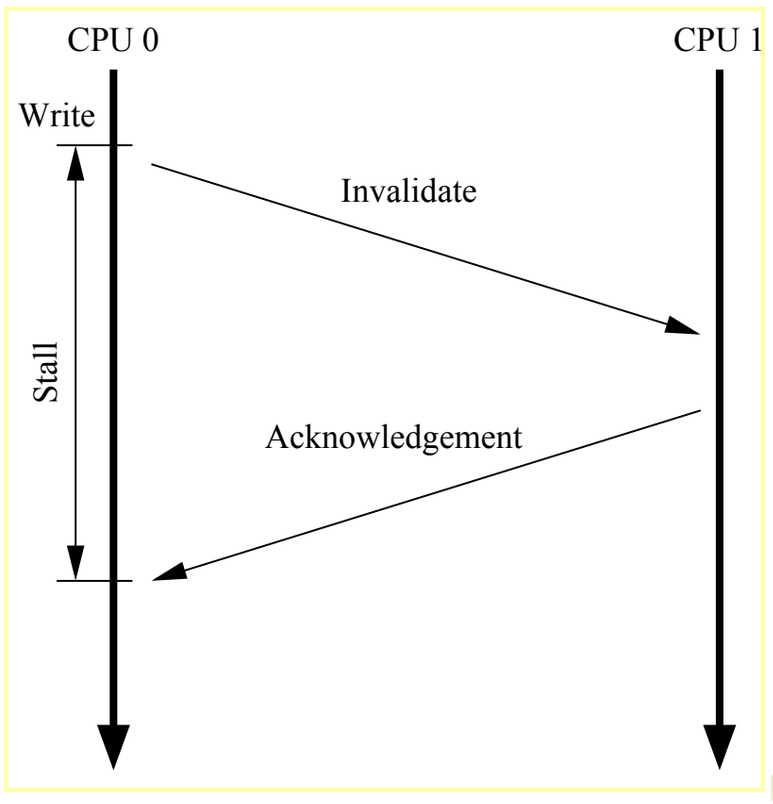

Figure 4: Writes See Unnecessary Stalls

But there is no real reason to force CPU 0 to stall for so long — after all, regardless of what data happens to be in the cache line that CPU 1 sends it, CPU 0 is going to unconditionally overwrite it.

3.1 Store Buffers

One way to prevent this unnecessary stalling of writes is to add "store buffers" between each CPU and its cache, as shown in Figure 5. With the addition of these store buffers, CPU 0 can simply record its write in its store buffer and continue executing. When the cache line does finally make its way from CPU 1 to CPU 0, the data will be moved from the store buffer to the cache line.

However, there are complications that must be addressed, which are covered in the next two sections.

3.2 Store Forwarding

To see the first complication, a violation of self-consistency, consider the following code with variables "a" and "b" both initially zero, and with the cache line containing variable "a" initially owned by CPU 1 and that containing "b" initially owned by CPU 0:

Sequence #	CPU #	Operation	CPU Cache				Memory	
			0	1	2	3	0	8
0		Initial State	-/I	-/I	-/I	-/I	V	V
1	0	Load	0/S	-/I	-/I	-/I	V	V
2	3	Load	0/S	-/I	-/I	0/S	V	V
3	0	Writeback	8/S	-/I	-/I	0/S	V	V
4	2	RMW	8/S	-/I	0/E	-/I	V	V
5	2	Store	8/S	-/I	0/M	-/I	I	V
6	1	Atomic Inc	8/S	0/M	-/I	-/I	I	V
7	1	Writeback	8/S	8/S	-/I	-/I	V	V

Table 1: Cache Coherence Example

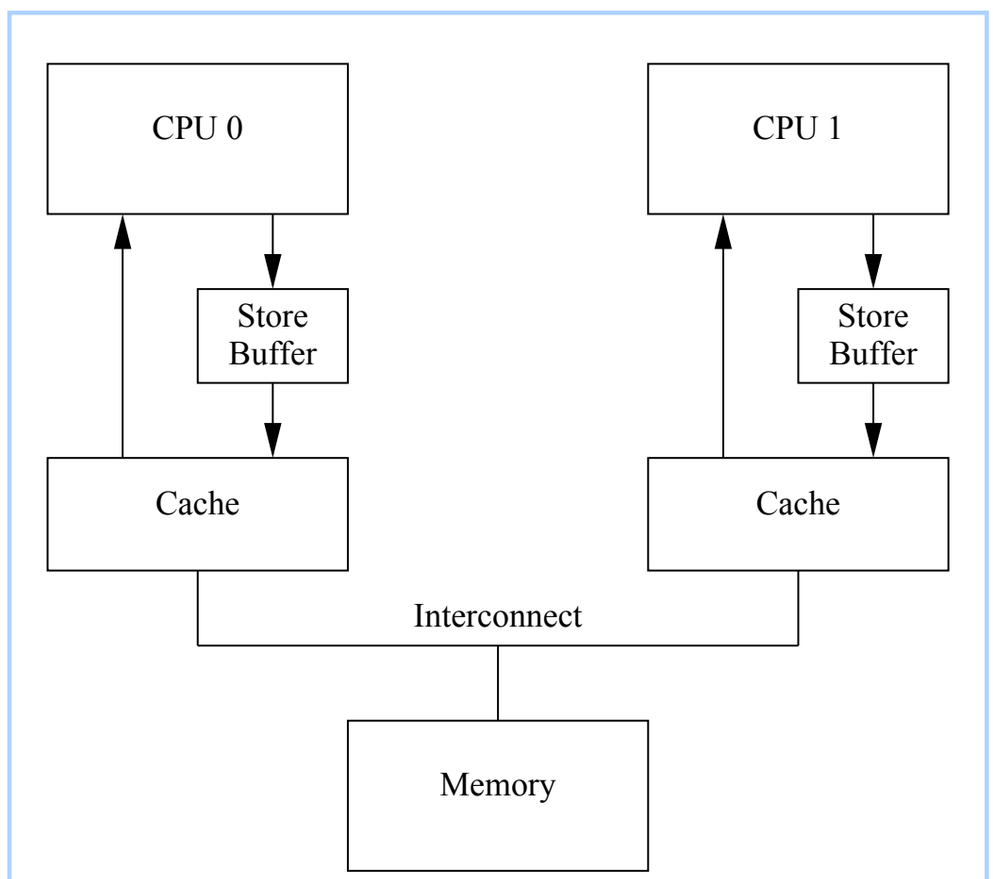

Figure 5: Caches With Store Buffers

```
1   a = 1;
2   b = a + 1;
3   assert(b == 2);
```

One would not expect the assertion to fail. However, if one were foolish enough to use the very simple architecture shown in Figure 5, one would be surprised. Such a system could potentially see the following sequence of events:

1. CPU 0 starts executing the `a=1`.
2. CPU 0 looks "a" up in the cache, and finds that it is missing.
3. CPU 0 therefore sends a "read invalidate" message in order to get exclusive ownership of the cache line containing "a".
4. CPU 0 records the store to "a" in its store buffer.
5. CPU 1 receives the "read invalidate" message, and responds by transmitting the cache line and removing that cacheline from its cache.
6. CPU 0 starts executing the `b=a+1`.
7. CPU 0 receives the cache line from CPU 1, which still has a value of zero for "a".
8. CPU 0 loads "a" from its cache, finding the value zero.
9. CPU 0 applies the entry from its store queue to the newly arrived cache line, setting the value of "a" in its cache to one.
10. CPU 0 adds one to the value zero loaded for "a" above, and stores it into the cache line containing "b" (which we will assume is already owned by CPU 0).
11. CPU 0 executes `assert(b==2)`, which fails.

The problem is that we have two copies of "a", one in the cache and the other in the store buffer.

This example breaks a very important guarantee, namely that each CPU will always see its own operations as if they happened in program order.

This guarantee is violently counter-intuitive to software types, so much so that the hardware guys took pity and implemented "store forwarding", where each CPU refers to (or "snoops") its store buffer as well as its cache when performing loads, as shown in Figure 6. In other words, a given CPU's stores are directly forwarded to its subsequent loads, without having to pass through the cache.

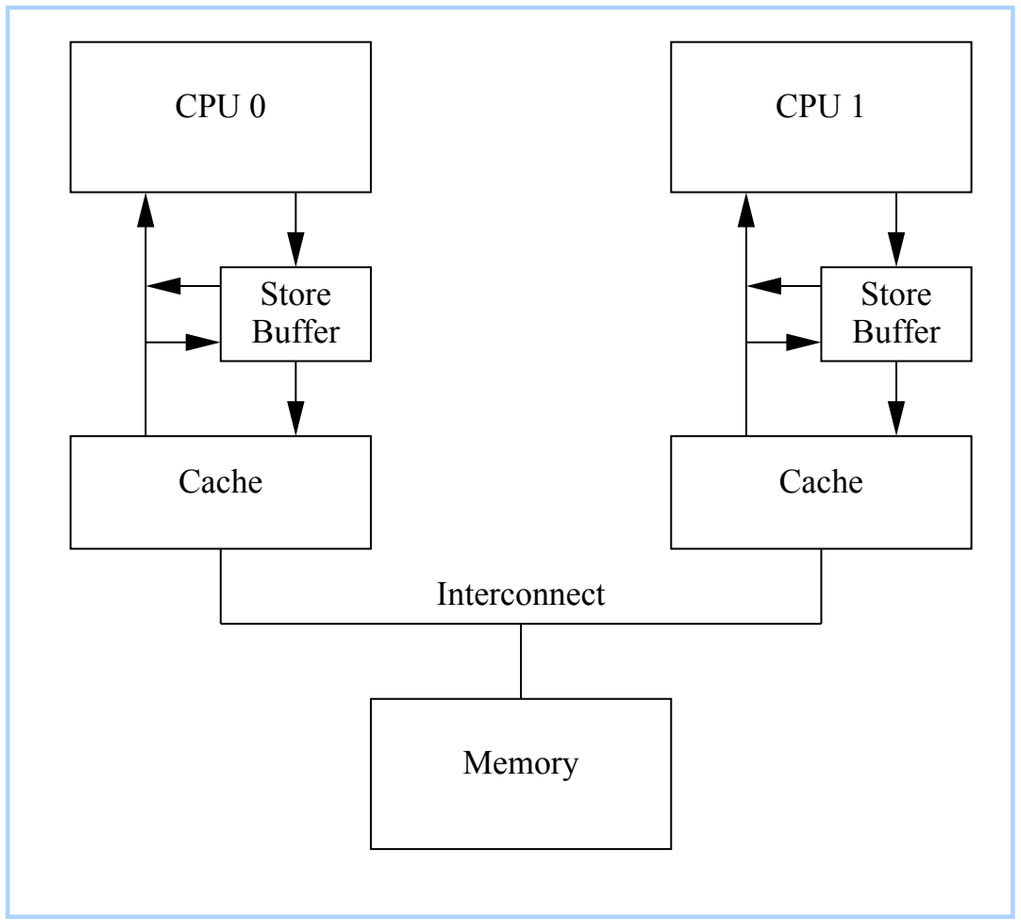

Figure 6: Caches With Store Forwarding

With store forwarding in place, item 8 in the above sequence would have found the correct value of 1 for "a" in the store buffer, so that the final value of "b" would have been 2, as one would hope.

3.3 Store Buffers and Memory Barriers

To see the second complication, a violation of global memory ordering, consider the following code sequences with variables "a" and "b" initially zero:

```
 1 void foo(void)
 2 {
 3   a = 1;
 4   b = 1;
 5 }
 6
 7 void bar(void)
 8 {
 9   while (b == 0) continue;
10   assert(a == 1);
11 }
```

Suppose CPU 0 executes foo() and CPU 1 executes bar(). Suppose further that the cache line containing "a" resides only in CPU 1's cache, and that the cache line containing "b" is owned by CPU 0. Then the sequence of operations might be as follows:

1. CPU 0 executes `a=1`. The cache line is not in CPU 0's cache, so CPU 0 places the new value of "a" in its store buffer and transmits a "read invalidate" message.
2. CPU 1 executes `while(b==0)continue`, but the cache line containing "b" is not in its cache. It therefore transmits a "read" message.
3. CPU 0 executes `b=1`. It already owns this cache line (in other words, the cache line is already in either the "modified" or the "exclusive" state), so it stores the new value of "b" in its cache line.
4. CPU 0 receives the "read" message, and transmits the cache line containing the now-updated value of "b" to CPU 1, also marking the line as "shared" in its own cache.
5. CPU 1 receives the cache line containing "b" and installs it in its cache.
6. CPU 1 can now finish executing `while(b==0) continue`, and since it finds that the value of "b" is 1, it proceeds to the next statement.
7. CPU 1 executes the `assert(a==1)`, and, since CPU 1 is working with the old value of "a", this assertion fails.
8. CPU 1 receives the "read invalidate" message, and transmits the cache line containing "a" to

CPU 0 and invalidates this cache line from its own cache. But it is too late.

9. CPU 0 receives the cache line containing "a" and applies the buffered store just in time to fall victim to CPU 1's failed assertion.

The hardware designers cannot help directly here, since the CPUs have no idea which variables are related, let alone how they might be related. Therefore, the hardware designers provide memory-barrier instructions to allow the software to tell the CPU about such relations. The program fragment must be updated to contain the memory barrier:

```
 1 void foo(void)
 2 {
 3   a = 1;
 4   smp_mb();
 5   b = 1;
 6 }
 7
 8 void bar(void)
 9 {
10   while (b == 0) continue;
11   assert(a == 1);
12 }
```

The memory barrier `smp_mb()` will cause the CPU to flush its store buffer before applying subsequent stores to their cache lines. The CPU could either simply stall until the store buffer was empty before proceeding, or it could use the store buffer to hold subsequent stores until all of the prior entries in the store buffer had been applied.

With this latter approach the sequence of operations might be as follows:

1. CPU 0 executes `a=1`. The cache line is not in CPU 0's cache, so CPU 0 places the new value of "a" in its store buffer and transmits a "read invalidate" message.

2. CPU 1 executes `while(b==0)continue`, but the cache line containing "b" is not in its cache. It therefore transmits a "read" message.

3. CPU 0 executes `smp_mp()`, and marks all current store-buffer entries (namely, the `a=1`).

4. CPU 0 executes `b=1`. It already owns this cache line (in other words, the cache line is already in either the "modified" or the "exclusive" state), but there is a marked entry in the store buffer. Therefore, rather than store the new value of "b" in the cache line, it instead places it in the store buffer (but in an *unmarked* entry).

5. CPU 0 receives the "read" message, and transmits the cache line containing the original value of "b" to CPU 1. It also marks its own copy of this cache line as "shared".

6. CPU 1 receives the cache line containing "b" and installs it in its cache.

7. CPU 1 can now finish executing `while(b==0) continue`, but since it finds that the value of "b" is still 0, it repeats the `while` statement. The new value of "b" is safely hidden in CPU 0's store buffer.

8. CPU 1 receives the "read invalidate" message, and transmits the cache line containing "a" to CPU 0 and invalidates this cache line from its own cache.

9. CPU 0 receives the cache line containing "a" and applies the buffered store.

10. Since the store to "a" was the only entry in the store buffer that was marked by the `smp_mb()`, CPU 0 can also store the new value of "b" — except for the fact that the cache line containing "b" is now in "shared" state.

11. CPU 0 therefore sends an "invalidate" message to CPU 1.

12. CPU 1 receives the "invalidate" message, invalidates the cache line containing "b" from its cache, and sends an "acknowledgement" message to CPU 0.

13. CPU 1 executes `while(b==0)continue`, but the cache line containing "b" is not in its cache. It therefore transmits a "read" message to CPU 0.

14. CPU 0 receives the "acknowledgement" message, and puts the cache line containing "b" into the "exclusive" state. CPU 0 now stores the new value of "b" into the cache line.

15. CPU 0 receives the "read" message, and transmits the cache line containing the original value of "b" to CPU 1. It also marks its own copy of this cache line as "shared".

16. CPU 1 receives the cache line containing "b" and installs it in its cache.

17. CPU 1 can now finish executing `while(b==0) continue`, and since it finds that the value of "b" is 1, it proceeds to the next statement.

18. CPU 1 executes the `assert(a==1)`, but the cache line containing "a" is no longer in its cache. Once it gets this cache from CPU 0, it will be working with the up-to-date value of "a", and the assertion therefore passes.

As you can see, this process involves no small amount of bookkeeping. Even something intuitively simple, like "load the value of a" can involves lots of complex steps in silicon.

4 Store Sequences Result in Unnecessary Stalls

Unfortunately, each store buffer must be relatively small, which means that a CPU executing a modest sequence of stores can fill its store buffer (for example, if all of them result in cache misses). At that point, the CPU must once again wait for invalidations to complete in order to drain its store buffer before it can continue executing. This same situation can arise immediately after a memory barrier, when *all* subsequent store instructions must wait for invalidations to complete, regardless of whether or not these stores result in cache misses.

This situation can be improved by making invalidate acknowledge messages arrive more quickly. One way of accomplishing this is to use per-CPU queues of invalidate messages, or "invalidate queues".

4.1 Invalidate Queues

One reason that invalidate acknowledge messages can take so long is that they must ensure that the corresponding cache line is actually invalidated, and this invalidation can be delayed if the cache is busy, for example, if the CPU is intensively loading and storing data, all of which resides in the cache. In addition, if a large number of invalidate messages arrive in a short time period, a given CPU might fall behind in processing them, thus possibly stalling all the other CPUs.

However, the CPU need not actually invalidate the cache line before sending the acknowledgement. It could instead queue the invalidate message with the understanding that the message will be processed before the CPU sends any further messages regarding that cache line.

4.2 Invalidate Queues and Invalidate Acknowledge

Figure 7 shows a system with invalidate queues. A CPU with an invalidate queue may acknowledge an invalidate message as soon as it is placed in the queue, instead of having to wait until the corresponding line is actually invalidated. Of course, the CPU must refer to its invalidate queue when preparing to transmit invalidation messages — if an entry for the corresponding cache line is in the invalidate queue, the CPU cannot immediately transmit the invalidate message; it must instead wait until the invalidate-queue entry has been processed.

Placing an entry into the invalidate queue is essentially a promise by the CPU to process that entry before transmitting any MESI protocol messages regarding that cache line. As long as the corresponding data structures are not highly contended, the CPU will rarely be inconvenienced by such a promise.

However, the fact that invalidate messages can be buffered in the invalidate queue provides additional opportunity for memory-misordering, as discussed in the next section.

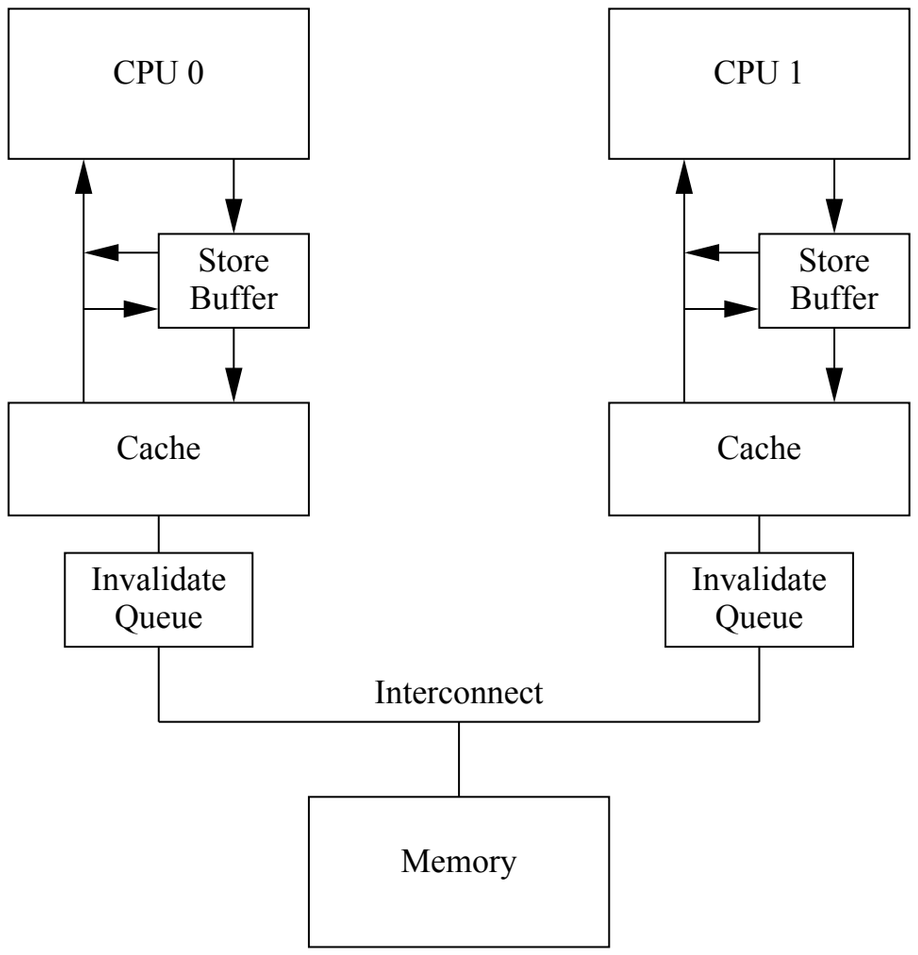

Figure 7: Caches With Invalidate Queues

4.3 Invalidate Queues and Memory Barriers

Suppose the values of "a" and "b" are initially zero, that "a" is replicated read-only (MESI "shared" state), and that "b" is owned by CPU 0 (MESI "exclusive" or "modified" state). Then suppose that CPU 0 executes `foo()` while CPU 1 executes function `bar()` in the following code fragment:

```
 1 void foo(void)
 2 {
 3   a = 1;
 4   smp_mb();
 5   b = 1;
 6 }
 7
 8 void bar(void)
 9 {
10   while (b == 0) continue;
11   assert(a == 1);
12 }
```

Then the sequence of operations might be as follows:

1. CPU 0 executes `a=1`. The corresponding cache line is read-only in CPU 0's cache, so CPU 0 places the new value of "a" in its store buffer and transmits an "invalidate" message in order to flush the corresponding cache line from CPU 1's cache.

2. CPU 1 executes `while(b==0)continue`, but the cache line containing "b" is not in its cache. It therefore transmits a "read" message.

3. CPU 0 executes `b=1`. It already owns this cache line (in other words, the cache line is already in either the "modified" or the "exclusive" state), so it stores the new value of "b" in its cache line.

4. CPU 0 receives the "read" message, and transmits the cache line containing the now-updated value of "b" to CPU 1, also marking the line as "shared" in its own cache.

5. CPU 1 receives the "invalidate" message for "a", places it into its invalidate queue, and transmits an "invalidate acknowledge" message to CPU 0. Note that the old value of "a" still remains in CPU 1's cache.

6. CPU 1 receives the cache line containing "b" and installs it in its cache.

7. CPU 1 can now finish executing `while(b==0) continue`, and since it finds that the value of "b" is 1, it proceeds to the next statement.

8. CPU 1 executes the `assert(a==1)`, and, since the old value of "a" is still in CPU 1's cache, this assertion fails.

9. CPU 1 processes the queued "invalidate" message, and invalidates the cache line containing "a" from its own cache. But it is too late.

10. CPU 0 receives the "invalidate acknowledge" message for "a" from CPU 0, and therefore applies the buffered store just in time to fall victim to CPU 1's failed assertion.

Once again, the CPU designers cannot do much about this situation, since the hardware does not know what relationships there might be among what to the CPU are just different piles of bits. However, the memory-barrier instructions can interact with the invalidate queue, so that when a given CPU executes a memory barrier, it marks all the entries currently in its invalidate queue, and forces any subsequent load to wait until all marked entries have been applied to the CPU's cache. Therefore, we can add a memory barrier as follows:

```
 1 void foo(void)
 2 {
 3   a = 1;
 4   smp_mb();
 5   b = 1;
 6 }
 7
 8 void bar(void)
 9 {
10   while (b == 0) continue;
11   smp_mb();
12   assert(a == 1);
13 }
```

With this change, the sequence of operations might be as follows:

1. CPU 0 executes `a=1`. The corresponding cache line is read-only in CPU 0's cache, so CPU 0 places the new value of "a" in its store buffer and transmits an "invalidate" message in order to flush the corresponding cache line from CPU 1's cache.

2. CPU 1 executes `while(b==0)continue`, but the cache line containing "b" is not in its cache. It therefore transmits a "read" message.

3. CPU 0 executes `b=1`. It already owns this cache line (in other words, the cache line is already in either the "modified" or the "exclusive" state), so it stores the new value of "b" in its cache line.

4. CPU 0 receives the "read" message, and transmits the cache line containing the now-updated value of "b" to CPU 1, also marking the line as "shared" in its own cache.

5. CPU 1 receives the "invalidate" message for "a", places it into its invalidate queue, and transmits an "invalidate acknowledge" message to CPU 0. Note that the old value of "a" still remains in CPU 1's cache.

6. CPU 1 receives the cache line containing "b" and installs it in its cache.

7. CPU 1 can now finish executing `while(b==0) continue`, and since it finds that the value of "b" is 1, it proceeds to the next statement.

8. CPU 1 executes the `smp_mb()`, marking the entry in its invalidate queue.

9. CPU 1 executes the `assert(a==1)`, but since there is a marked entry for the cache line containing "a" in the invalidate queue, CPU 1 must stall this load until that entry in the invalidate queue has been applied.

10. CPU 1 processes the "invalidate" message, removing the cacheline containing "a" from its cache.

11. CPU 1 is now free to load the value of "a", but since this results in a cache miss, it must send a "read" message to fetch the corresponding cache line.

12. CPU 0 receives the "invalidate acknowledge" message for "a" from CPU 0, and therefore applies the buffered store, changing the MESI state of the corresponding cache line to "modified".

13. CPU 0 receives the "read" message for "a" from CPU 1, and therefore changes the state of the corresponding cache line to "shared", and transmits the cache line to CPU 1.

14. CPU 1 receives the cache line containing "a", and can therefore do the load. Since this load returns the updated value of "a", the assertion passes.

With much passing of MESI messages, the CPUs arrive at the correct answer.

5 Read and Write Memory Barriers

In the previous section, memory barriers were used to mark entries in both the store buffer and the invalidate queue. But in our code fragment, `foo()` had no reason to do anything with the invalidate queue, and `bar()` simlarly had no reason to do anything with the store queue.

Many CPU architectures therefore provide weaker memory-barrier instructions that do only one or the other of these two. Roughly speaking, a "read memory barrier" marks only the invalidate queue and a "write memory barrier" marks only the store buffer. while a full-fledged memory barrier does both.

The effect of this is that a read memory barrier orders only loads on the CPU that executes it, so that all loads preceding the read memory barrier will appear to have completed before any load following the read memory barrier. Similarly, a write memory barrier orders only stores, again on the CPU that executes it, and again so that all stores preceding the write memory barrier will appear to have completed before any store following the write memory barrier. A full-fledged memory barrier orders both loads and stores, but again only on the CPU executing the memory barrier.

If we update `foo` and `bar` to use read and write memory barriers, they appear as follows:

```
 1 void foo(void)
 2 {
 3   a = 1;
 4   smp_wmb();
 5   b = 1;
 6 }
 7
 8 void bar(void)
 9 {
10   while (b == 0) continue;
11   smp_rmb();
12   assert(a == 1);
13 }
```

Some computers have even more flavors of memory barriers, but understanding these three variants will provide a good introduction to memory barriers in general.

6 Example Memory-Barrier Sequences

This section presents some seductive but subtly broken uses of memory barriers. Although many of them will work most of the time, and some will work all the time on some specific CPUs, these uses must be avoided if the goal is to produce code that works reliably on all CPUs. To help us better see the subtle breakage, we first need to focus on an ordering-hostile architecture.

6.1 Ordering-Hostile Architecture

Paul has come across a number of ordering-hostile computer systems, but the nature of the hostility has always been extremely subtle, and understanding it has required detailed knowledge of the specific hardware. Rather than picking on a specific hardware vendor, and as a presumably attractive alternative to dragging the reader through detailed technical specifications, let us instead design a mythical but maximally memory-ordering-hostile computer architecture.[4]

This hardware must obey the following ordering constraints [16, 17]:

1. Each CPU will always perceive its own memory accesses as occuring in program order.

2. CPUs will reorder a given operation with a store only if the two operations are referencing different locations.

3. All of a given CPU's loads preceding a read memory barrier (`smp_rmb()`) will be perceived by all CPUs to precede any loads following that read memory barrier.

[4] Readers preferring a detailed look at real hardware architectures are encouraged to consult CPU vendors' manuals [19, 1, 8, 6, 15, 20, 10, 9, 13] or Gharachorloo's dissertation [4].

CPU 0	CPU 1	CPU 2
`a=1;`		
`smp_wmb();`	`while(b==0);`	
`b=1;`	`c=1;`	`z=c;`
		`smp_rmb();`
		`x=a;`
		`assert(z==0||x==1);`

Table 2: Memory Barrier Example 1

4. All of a given CPU's stores preceding a write memory barrier (`smp_wmb()`) will be perceived by all CPUs to precede any stores following that write memory barrier.

5. All of a given CPU's accesses (loads and stores) preceding a full memory barrier (`smp_mb()`) will be perceived by all CPUs to precede any accesses following that memory barrier.

Quick Quiz 6: Does the guarantee that each CPU sees its own memory accesses in order also guarantee that each user-level thread will see its own memory accesses in order? Why or why not? □

Imagine a large non-uniform cache architecture (NUCA) system that, in order to provide fair allocation of interconnect bandwidth to CPUs in a given node, provided per-CPU queues in each node's interconnect interface, as shown in Figure 8. Although a given CPU's accesses are ordered as specified by memory barriers executed by that CPU, however, the relative order of a given pair of CPUs' accesses could be severely reordered, as we will see.[5]

6.2 Example 1

Figure2 shows three code fragments, executed concurrently by CPUs 1, 2, and 3. Each of "a", "b", and "c" are initially zero.

[5] Any real hardware architect or designer will no doubt be loudly calling for Ralph on the porcelain intercom, as they just might be just a bit upset about the prospect of working out which queue should handle a message involving a cache line that both CPUs accessed, to say nothing of the many races that this example poses. All I can say is "Give me a better example".

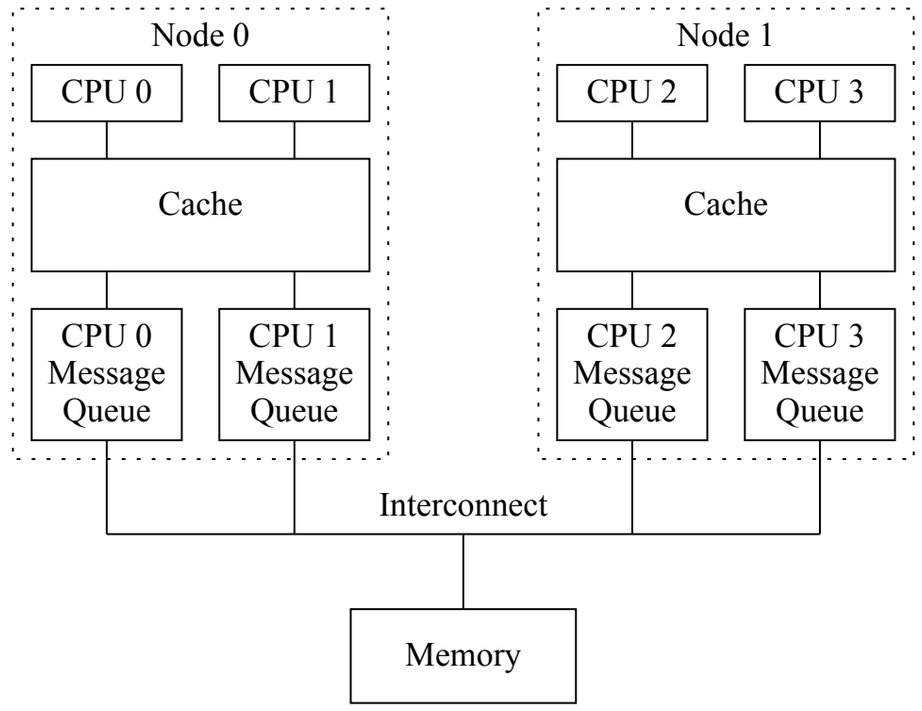

Figure 8: Example Ordering-Hostile Architecture

Suppose CPU 0 recently experienced many cache misses, so that its message queue is full, but that CPU 1 has been running exclusively within the cache, so that its message queue is empty. Then CPU 0's assignment to "a" and "b" will appear in Node 0's cache immediately (and thus be visible to CPU 1), but will be blocked behind CPU 0's prior traffic. In contrast, CPU 1's assignment to "c" will sail through CPU 1's previously empty queue. Therefore, CPU 2 might well see CPU 1's assignment to "c" before it sees CPU 0's assignment to "a", causing the assertion to fire, despite the memory barriers.

In theory, portable code could not on this example coding technique, however, in practice it actually does work on all mainstream computer systems.

Quick Quiz 7: Could this code be fixed by inserting a memory barrier between CPU 1's "while" and assignment to "c"? Why or why not? □

6.3 Example 2

Figure3 shows three code fragments, executed concurrently by CPUs 1, 2, and 3. Both "a" and "b" are initially zero.

Again, suppose CPU 0 recently experienced many cache misses, so that its message queue is full, but that CPU 1 has been running exclusively within the

CPU 0	CPU 1	CPU 2
a=1;	while(a==0);	
	smp_mb();	y=b;
	b=1;	smp_rmb();
		x=a;
		assert(y==0||x==1);

Table 3: Memory Barrier Example 2

	CPU 0	CPU 1	CPU 2
1	a=1;		
2	smb_wmb();		
3	b=1;	while(b==0);	while(b==0);
4		smp_mb();	smb_mb();
5		c=1;	d=1;
6	while(c==0);		
7	while(d==0);		
8	smp_mb();		
9	e=1;		assert(e==0||a==1);

Table 4: Memory Barrier Example 3

cache, so that its message queue is empty. Then CPU 0's assignment to "a" and "b" will appear in Node 0's cache immediately (and thus be visible to CPU 1), but will be blocked behind CPU 0's prior traffic. In contrast, CPU 1's assignment to "b" will sail through CPU 1's previously empty queue. Therefore, CPU 2 might well see CPU 1's assignment to "b" before it sees CPU 0's assignment to "a", causing the assertion to fire, despite the memory barriers.

In theory, portable code could not on this example coding technique, however, as before, in practice it actually does work on all mainstream computer systems.

6.4 Example 3

Figure4 shows three code fragments, executed concurrently by CPUs 1, 2, and 3. All variables are initially zero.

Note that neither CPU 1 nor CPU 2 can proceed to line 4 until they see CPU 0's assignment to "b" on line 3. Once CPU 1 and 2 have executed their memory barriers on line 3, they are both guaranteed to see all assignments by CPU 0 preceding its memory barrier on line 2. Similarly, CPU 0's memory barrier on line 8 pairs with those of CPUs 1 and 2 on line 4, so that CPU 0 will not execute the assignment to "e" on line 9 until after its assignment to "a" is visible to both of the other CPUs. Therefore, CPU 2's assertion on line 9 is guaranteed *not* to fire.

Quick Quiz 8: Suppose that lines 3-5 for CPUs 1 and 2 are in an interrupt handler, and that the CPU 2's line 9 is run at process level. What changes, if any, are required to enable the code to work correctly, in other words, to prevent the assertion from firing? □

The Linux kernel's `synchronize_rcu()` primitive uses an algorithm similar to that shown in this example.

7 Memory-Barrier Instructions For Specific CPUs

Each CPU has its own peculiar memory-barrier instructions, which can make portability a challenge, as indicated by Table 5. In fact, many software environments, including pthreads and Java, simply prohibit direct use of memory barriers, restricting the programmer to mutual-exclusion primitives that incorporate them to the extent that they are required. In the table, 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.

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

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

Table 5: Summary of Memory Ordering

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.

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

The common "just say no" approach to memory barriers can be eminently reasonable where it applies, but there are environments, such as the Linux kernel, where direct use of memory barriers is required. Therefore, Linux provides a carefully chosen least-common-denominator 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.

Furthermore, 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.

7.1 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 9. 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 the code on line 20 of Figure 9 could see the old garbage values that were present before the initialization on lines 6-8.

```
 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 9: Insert and Lock-Free Search

Figure 10 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 9 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.[6]

One could place an `smp_rmb()` primitive be-

[6] Of course, the astute reader will have already recognized that Alpha is nowhere near as mean and nasty as it could be, the (thankfully) mythical architecture in Section 6.1 being a case in point.

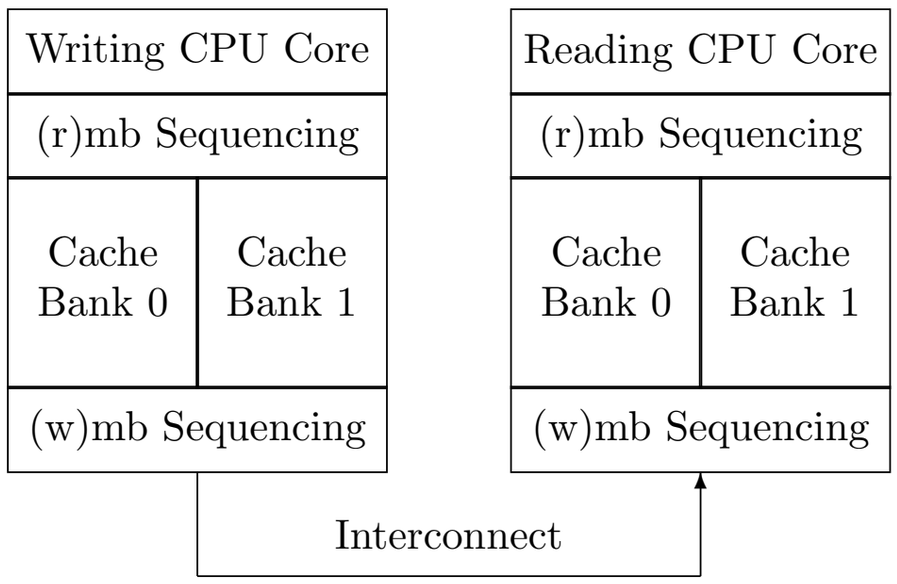

Figure 10: Why smp_read_barrier_depends() is Required

```
 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 11: Safe Insert and Lock-Free Search

tween 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 11.

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 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.

For more detail on Alpha, see the reference manual [19].

7.2 AMD64

AMD64 is compatible with x86, and has recently updated its memory model [2] to enforce the tighter ordering that actual implementations have provided for some time. The AMD64 implementation of the Linux `smp_mb()` primitive is `mfence`, `smp_rmb()` is `lfence`, and `smp_wmb()` is `sfence`. In theory, these might be relaxed, but any such relaxation must take SSE and 3DNOW instructions into account.

7.3 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 [8]. IA64 has a memory-fence instruction named `mf`, but also has "half-memory fence" modifiers to loads, stores, and to some of its atomic

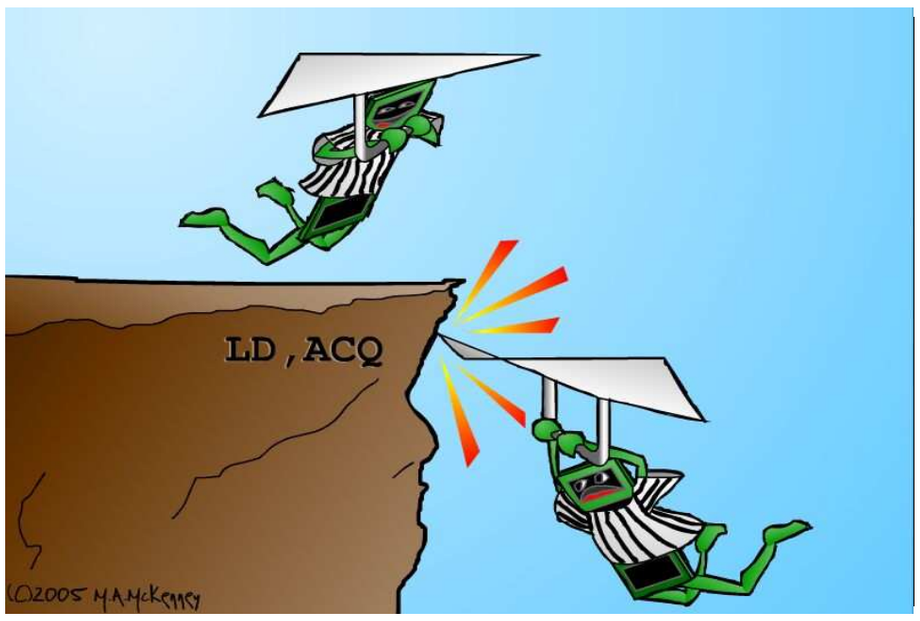

Figure 12: Half Memory Barrier

instructions [7]. 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 12. 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. However, as one of the only CPUs with this property, IA64 defines Linux's semantics of memory ordering associated with lock acquisition and release.

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".

Finally, IA64 offers a global total order for "release" operations, including the "mf" instruction. This provides the notion of transitivity, where if a given code fragment sees a given access as having happened, any later code fragment will also see that earlier access has having happened. Assuming, that is, that all the code fragments involved correctly use memory barriers.

7.4 PA-RISC

Although the PA-RISC architecture permits full reordering of loads and stores, actual CPUs run fully ordered [14]. 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.

7.5 POWER

The POWER and Power PC® CPU families have a wide variety of memory-barrier instructions [6, 15]:

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.

Power features "cumulativity", which can be used to obtain transitivity. When used properly, any code seeing the results of an earlier code fragment will also see the accesses that this earlier code fragment itself saw. Much more detail is available from McKenney and Silvera [18].

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.

7.6 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 [20]. 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 [20] 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 [20]. 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.

7.7 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 [10]. 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 have traditionally given 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.

More recently, Intel has published a memory model for x86 [11]. It turns out that Intel's actual CPUs enforced tighter ordering than was claimed in the previous specifications, so this model is in effect simply mandating the earlier de-facto behavior. Even more recently, Intel published an updated memory model for x86 [12], which mandates a total global order for stores, although individual CPUs are still permitted to see their own stores as having happened earlier than this total global order would indicate. This exception to the total ordering is needed to allow important hardware optimizations involving store buffers. Software may use atomic operations to override these hardware optimizations, which is one reason that atomic operations tend to be more expensive than their non-atomic counterparts. This total store order is *not* guaranteed on older processors.

However, note that some SSE instructions are weakly ordered (`clflush` and non-temporal move instructions [9]). 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.

7.8 zSeries

The zSeries machines make up the IBM™ mainframe family, previously known as the 360, 370, and 390 [13]. 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 5, 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.

8 Are Memory Barriers Forever?

There have been a number of recent systems that are significantly less aggressive about out-of-order execution in general and re-ordering memory references in particular. Will this trend continue to the point where memory barriers are a thing of the past?

The argument in favor would cite proposed massively multi-threaded hardware architectures, so that each thread would wait until memory was ready, with tens, hundreds, or even thousands of other threads making progress in the meantime. In such an architecture, there would be no need for memory barriers, because a given thread would simply wait for all outstanding operations to complete before proceeding to the next instruction. Because there would be potentially thousands of other threads, the CPU would be completely utilized, so no CPU time would be wasted.

The argument against would cite the extremely limited number of applications capable of scaling up to a thousand threads, as well as increasingly severe realtime requirements, which are in the tens of microseconds for some applications. The realtime-response requirements are difficult enough to meet as is, and would be even more difficult to meet given the extremely low single-threaded throughput implied by the massive multi-threaded scenarios.

Another argument in favor would cite increasingly sophisticated latency-hiding hardware implementation techniques that might well allow the CPU to provide the illusion of fully sequentially consistent execution while still providing almost all of the performance advantages of out-of-order execution. A counter-argument would cite the increasingly severe power-efficiency requirements presented both by battery-operated devices and by environmental responsibility.

Who is right? We have no clue, so are preparing to live with either scenario.

9 Advice to Hardware Designers

There are any number of things that hardware designers can do to make the lives of software people difficult. Here is a list of a few such things that we have encountered in the past, presented here in the hope that it might help prevent future such problems:

1. I/O devices that ignore cache coherence.

   This charming misfeature can result in DMAs from memory missing recent changes to the output buffer, or, just as bad, cause input buffers to be overwritten by the contents of CPU caches just after the DMA completes. To make your system work in face of such misbehavior, you must carefully flush the CPU caches of any location in any DMA buffer before presenting that buffer to the I/O device. And even then, you need to be *very* careful to avoid pointer bugs, as even a misplaced read to an input buffer can result in corrupting the data input!

2. Device interrupts that ignore cache coherence.

   This might sound innocent enough — after all, interrupts aren't memory references, are they? But imagine a CPU with a split cache, one bank of which is extremely busy, therefore holding onto the last cacheline of the input buffer. If the corresponding I/O-complete interrupt reaches this CPU, then that CPU's memory reference to the last cache line of the buffer could return old data, again resulting in data corruption, but in a form that will be invisible in a later crash dump. By the time the system gets around to dumping the offending input buffer, the DMA will most likely have completed.

3. Inter-processor interrupts (IPIs) that ignore cache coherence.

   This can be problematic if the IPI reaches its destination before all of the cache lines in the corresponding message buffer have been committed to memory.

4. Context switches that get ahead of cache coherence.

   If memory accesses can complete too wildly out of order, then context switches can be quite harrowing. If the task flits from one CPU to another before all the memory accesses visible to the source CPU make it to the destination CPU, then the task could easily see the corresponding variables revert to prior values, which can fatally confuse most algorithms.

5. Overly kind simulators and emulators.

   It is difficult to write simulators or emulators that force memory re-ordering, so software that runs just fine in these these environments can get a nasty surprise when it first runs on the real hardware. Unfortunately, it is still the rule that the hardware is more devious than are the simulators and emulators, but we hope that this situation changes.

Again, we encourage hardware designers to avoid these practices!

Acknowledgements

I own thanks to many CPU architects for patiently explaining the instruction- and memoyr-reordering features of their CPUs, particularly Wayne Cardoza, Ed Silha, Anton Blanchard, Brad Frey, Cathy May, Derek Williams, Tim Slegel, Juergen Probst, Ingo Adlung, and Ravi Arimilli. Wayne deserves special thanks for his patience in explaining Alphas reordering of dependent loads, a lesson that I resisted quite strenu- ously!

Legal Statement

This work represents the view of the author and does not necessarily represent the view of IBM.
IBM, zSeries, and Power PC are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.
Linux is a registered trademark of Linus Torvalds.
i386 is a trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.
Other company, product, and service names may be trademarks or service marks of such companies.
Copyright © 2005 by IBM Corporation.

Answers to Quick Quizzes

Quick Quiz 1:
Answer: What happens if two CPUs attempt to invalidate the same cache line concurrently? □
One of the CPUs gains access to the shared bus first, and that CPU "wins". The other CPU must invalidate its copy of the cache line and transmit an "invalidate acknowledge" message to the other CPU. Of course, the losing CPU can be expected to immediately issue a "read invalidate" transaction, so the winning CPU's victory will be quite ephemeral.

Quick Quiz 2:
Answer: When an "invalidate" message appears in a large multiprocessor, every CPU must give an "invalidate acknowledge" response. Wouldn't the resulting "storm" of "invalidate acknowledge" responses totally saturate the system bus? □
It might, if large-scale multiprocessors were in fact implemented that way. Larger multiprocessors, particularly NUMA machines, tend to use so-called "directory-based" cache-coherence protocols to avoid this and other problems.

Quick Quiz 3:
Answer: If SMP machines are really using message

passing anyway, why bother with SMP at all? □
There has been quite a bit of controversy on this topic over the past few decades. One answer is that the cache-coherence protocols are quite simple, and therefore can be implemented directly in hardware, gaining bandwidths and latencies unattainable by software message passing. Another answer is that the real truth is to be found in economics due to the relative prices of large SMP machines and that of clusters of smaller SMP machines. A third answer is that the SMP programming model is easier to use than that of distributed systems, but a rebuttal might note the appearance of HPC clusters and MPI. And so the argument continues.

Quick Quiz 4:
Answer: How does the hardware handle the delayed transitions described above? □
Usually by adding additional states, though these additional states need not be actually stored with the cache line, due to the fact that only a few lines at a time will be transitioning. The need to delay transitions is but one issue that results in real-world cache coherence protocols being much more complex than the over-simplified MESI protocol described in this appendix. Hennessy and Patterson's classic introduction to computer architecture [5] covers many of these issues.

Quick Quiz 5:
Answer: What sequence of operations would put the CPUs' caches all back into the "invalid" state? □
There is no such sequence, at least in absence of special "flush my cache" instructions in the CPU's instruction set. Most CPUs do have such instructions.

Quick Quiz 6:
Answer: Does the guarantee that each CPU sees its own memory accesses in order also guarantee that each user-level thread will see its own memory accesses in order? Why or why not? □
No. Consider the case where a thread migrates from one CPU to another, and where the destination CPU perceives the source CPU's recent memory operations out of order. To preserve user-mode sanity, kernel hackers must use memory barriers in the context-switch path. However, the locking already required to safely do a context switch should automatically provide the memory barriers needed to cause the user-level task to see its own accesses in order. That said, if you are designing a super-optimized scheduler, either in the kernel or at user level, please keep this scenario in mind!

Quick Quiz 7:
Answer: Could this code be fixed by inserting a memory barrier between CPU 1's "while" and assignment to "c"? Why or why not? □
No. Such a memory barrier would only force ordering local to CPU 1. It would have no effect on the relative ordering of CPU 0's and CPU 1's accesses, so the assertion could still fail. However, all mainstream computer systems provide one mechanism or another to provide "transitivity", which provides intuitive causal ordering: if B saw the effects of A's accesses, and C saw the effects of B's accesses, then C must also see the effects of A's accesses.

Quick Quiz 8:
Answer: Suppose that lines 3-5 for CPUs 1 and 2 are in an interrupt handler, and that the CPU 2's line 9 is run at process level. What changes, if any, are required to enable the code to work correctly, in other words, to prevent the assertion from firing? □
The assertion will need to coded so as to ensure that the load of "e" precedes that of "a". In the Linux kernel, the barrier() primitive may be used to accomplish this in much the same way that the memory barrier was used in the assertions in the previous examples.

References

[1] Advanced Micro Devices. *AMD x86-64 Ar-*

chitecture Programmer's Manual Volumes 1-5, 2002.

[2] Advanced Micro Devices. *AMD x86-64 Architecture Programmer's Manual Volume 2: System Programming*, 2007.

[3] Culler, D. E., Singh, J. P., and Gupta, A. *Parallel Computer Architecture: a Hardware/Software Approach*. Morgan Kaufman, 1999.

[4] Gharachorloo, 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].

[5] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 1995.

[6] IBM Microelectronics and Motorola. *PowerPC Microprocessor Family: The Programming Environments*, 1994.

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

[8] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 3: System Architecture*, 2002.

[9] 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].

[10] 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].

[11] Intel Corporation. *Intel 64 Architecture Memory Ordering White Paper*, 2007. Available: `http://developer.intel.com/products/processor/manuals/318147.pdf` [Viewed: September 7, 2007].

[12] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3A: System Programming Guide, Part 1*, 2009. Available: `http://download.intel.com/design/processor/manuals/253668.pdf` [Viewed: September 7, 2007].

[13] 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.

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

[15] 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.

[16] McKenney, P. E. Memory ordering in modern microprocessors, part I. *Linux Journal 1*, 136 (August 2005), 52–57. Available: `http://www.linuxjournal.com/article/8211` `http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.19a.pdf` [Viewed November 30, 2007].

[17] McKenney, P. E. Memory ordering in modern microprocessors, part II. *Linux Journal 1*, 137 (September 2005), 78–82. Available: `http://www.linuxjournal.com/article/8212` `http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.19a.pdf` [Viewed November 30, 2007].

[18] McKenney, P. E., and Silvera, R. Example power implementation for c/c++ memory

model. Available: `http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2009.02.27a.html` [Viewed: April 5, 2009], February 2009.

[19] Sites, R. L., and Witek, R. T. *Alpha AXP Architecture*, second ed. Digital Press, 1995.

[20] SPARC International. *The SPARC Architecture Manual*, 1994.