**Software the Hard way**

Thoughts on Software Development, from an ex-Hardware Engineer

≡ **Menu**

# Myths Programmers Believe about CPU Caches

❏ 2018-04-292018-04-30 ❏ 7 Minutes

As a computer engineer who has spent half a decade working with caches at Intel and Sun, I've learnt a thing or two about cache-coherency (https://en.wikipedia.org/wiki/Cache_coherence). This was one of the hardest concepts to learn back in college – but once you've truly understood it, it gives you a great appreciation for system design principles.

You might be wondering why you as a software developer should care about CPU cache-design. For one thing, many of the concepts learnt in cache-coherency are directly applicable to distributed-system-architecture (https://en.wikipedia.org/wiki/Distributed_computing) and database-isolation-levels (https://en.wikipedia.org/wiki/Isolation_(database_systems)#Isolation_levels) as well. For instance, understanding how coherency is implemented in hardware caches, can help in better understanding strong-vs-eventual consistency (https://hackernoon.com/eventual-vs-strong-consistency-in-distributed-databases-282fdad37cf7). It can spur ideas on how to better enforce consistency in distributed systems, using the same research and principles applied in hardware.

For another thing, misconceptions about caches often lead to false assertions, especially when it comes to concurrency and race conditions. For example, the common refrain that concurrent programming is hard because *"different cores can have different/stale values in their individual caches"*. Or that the reason we need volatiles (https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html) in languages like Java, is to *"prevent shared-data from being cached locally"*, and force them to be *"read/written all the way to main memory (http://tutorials.jenkov.com/java-concurrency/volatile.html)"*.

Such misconceptions are mostly harmless (and maybe even helpful), but can also lead to bad design decisions. For instance, developers can start to believe that they are insulated from the above concurrency bugs, when working with single-core-systems. In reality, even single-core systems are at risk of concurrency bugs (https://stackoverflow.com/questions/23593061/volatile-keyword-in-multicore-vs-single-processor), if the appropriate concurrency constructs aren't used.

For another, if volatile variables were truly written/read from main-memory every single time, they would be horrendously slow – main-memory references are 200x slower than L1 cache references (https://gist.github.com/jboner/2841832). In reality, volatile-reads (in Java) can often be just as cheap as a L1 cache reference (https://stackoverflow.com/questions/4633866/is-volatile-expensive), putting to rest the notion that volatile forces reads/writes all the way to main memory. If you've been avoiding the use of volatiles because of performance concerns, you might have been a victim of the above misconceptions.

## The Importance of Being Consistent

But if different cores each have their own private cache, storing copies of the same data, wouldn't that naturally lead to data mismatches as they start issuing writes? The answer: hardware caches on modern x86 CPUs like Intel's, are kept in-sync with one another. These caches aren't just dumb memory storage units, as many developers seem to think. Rather, there are very intricate protocols and logics, embedded in every cache, communicating with other caches, enforcing consistency across all threads. And all this is happening at the hardware level, meaning that we as software/compiler/systems developers don't have to deal with it.

A quick word about what I mean when I say that caches are *"in sync"*. There is a great wealth of nuance (https://en.wikipedia.org/wiki/Consistency_model) in this topic, but to simplify greatly, we mean the following: If 2 different threads, anywhere in the system, read from the *same* memory address, they should never *simultaneously* read different values.

For a quick example of how non-consistent caches can violate the above rule, simply refer to the first section of this tutorial (http://tutorials.jenkov.com/java-concurrency/volatile.html). No modern x86 CPU behaves the way the tutorial describes it, but a buggy processor certainly can. Everything discussed here is a means towards one simple end: preventing such data-mismatches from happening.

The most common protocol that's used to enforce consistency amongst caches, is known as the MESI protocol (https://en.wikipedia.org/wiki/MESI_protocol). Every processor has its own variant of this design, and these variants bring with them numerous benefits, tradeoffs and potential for unique bugs. However, these variants all share a great deal in common. And that's the following: each line of data sitting in a cache, is tagged with one of the following states:

1. Modified (M)
   1. This data has been modified, and differs from main memory
   2. This data is the source-of-truth, and all other data elsewhere is stale
2. Exclusive (E)
   1. This data has not been modified, and is in sync with the data in main memory
   2. No other sibling cache has this data
3. Shared (S)
   1. This data has not been modified, and is in sync with the data elsewhere
   2. There are other sibling caches that (may) also have this same data
4. Invalid (I)
   1. This data is stale, and should never ever be used

Cache consistency can now be accomplished as long as we enforce and update the above states. Let's look at a few examples for a CPU with 4 cores, each of which has its own L1 cache, along with a global on-chip L2 cache.

## Memory Write

Suppose a thread on core-1 wants to write to address 0xabcd. The following are some possible sequence of events.

### Cache Hit

1. L1-1 has the data in E or M state
2. L1-1 performs the write. All done
    1. No other cache has the data, so it is safe to write to it immediately
    2. The state of the cache-line is set to M, since it is now modified

### Local Cache Miss, Sibling Cache Hit

1. L1-1 has the data in S state
    1. This implies that another sibling cache might have the data
    2. This same flow is also used if L1-1 doesn't have the data at all
2. L1-1 sends a Request-For-Ownership to the L2 cache
3. L2 looks up its directory and sees that L1-2 currently has the data in S state
4. L2 sends a snoop-invalidate to L1-2
5. L1-2 marks its data as being Invalid (I)
6. L1-2 sends an Ack to L2
7. L2 sends an Ack, along with the latest data, to L1-1
    1. L2 keeps track of the fact that L1-1 has the data for this address in E state
8. L1-1 now has the latest data, as well as permission to enter E state
9. L1-1 performs the write, and changes the state of that data to M

## Memory Read

Now suppose a thread on core-2 wants to read from address 0xabcd. The following are some possible sequences of events.

### Cache Hit

1. L1-2 has the data in S or E or M state
2. L1-2 reads the data and returns it to the thread. All done

## Local Cache Miss, Parent Cache Miss

1. L1-2 has the data in I (invalid) state, meaning it's not allowed to use it
2. L1-2 sends a Request-for-Share to the L2 cache
3. L2 does not have the data either. It reads the data from memory
4. L2 gets back the data from memory
5. L2 sends this data to L1-2, along with permission to enter S state
    1. L2 keeps track of the fact that L1-2 has this data in S-state
6. L1-2 gets the data, stores it in its cache, and sends it to the thread

## Local Cache Miss, Parent Cache Hit

1. L1-2 has the data in I state
2. L1-2 sends a Request-for-S to the L2 cache
3. L2 sees that L1-1 has the data in S state
4. L2 sends an Ack to L1-2, along with the data, and permission to enter S state
5. L1-2 gets the data, stores it in its cache, and sends it to the thread

## Local Cache Miss, Sibling Cache Hit

1. L1-2 has the data in I state
2. L1-2 sends a Request-for-S to the L2 cache
3. L2 sees that L1-1 has the data in E (or M) state
4. L2 sends a snoop-share to L1-1
5. L1-1 downgrades its state to an S
6. L1-1 sends an Ack to L2, along with the modified data if applicable
7. L2 sends an Ack to L1-2, along with the data, and permission to enter S state
8. L1-2 gets the data, stores it in its cache, and sends it to the thread

# Variations

The above are just some of the possible scenarios that can occur. In reality, there are numerous variations of the above design, and no 2 implementations are the same. For example, some designs have an O/F state (https://en.wikipedia.org/wiki/MESIF_protocol). Some have write-back caches, whereas others use write-through (https://stackoverflow.com/questions/27087912/write-back-vs-write-through). Some use snoop-broadcasts, while others use a snoop-filter (https://en.wikipedia.org/wiki/Bus_snooping#Snoop_filter). Some have inclusive caches and others have exclusive caches (https://en.wikipedia.org/wiki/Cache_inclusion_policy). The variations are endless, and we haven't even discussed store-buffers (https://stackoverflow.com/questions/11105827/what-is-a-store-buffer)!

The above example also considers a simple processor with only 2 levels of caching, but note that this same protocol can also be applied recursively. You could easily add an L3 cache, which in turn coordinates multiple L2s, using the exact same protocol as above. You can also have a multi-processor system (https://software.intel.com/en-us/articles/how-memory-is-accessed), with "Home Agents" that coordinate across multiple L3 caches on completely different chips.

In each scenario, each cache only needs to communicate with its parent (to get data/permissions), and its children (to grant/revoke data/permissions). And all this can be accomplished in a manner that's invisible to the software thread. From the perspective of the software application, the memory subsystem appears to be a single, consistent, monolith … with *very* variable latencies.

## Why Synchronization Still Matters

One final word, now that we've discussed the awesome power and consistency of your computer's memory system. If caches are so consistent, why do we need volatiles at all in languages like Java (https://componenthouse.com/2016/12/28/comparing-the-volatile-keyword-in-java-c-and-cpp/)?

That's a very complicated question that's better answered elsewhere (https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html), but let me just drop one partial hint. Data that's read into CPU registers (https://en.wikipedia.org/wiki/Processor_register), is *not* kept in sync with data in cache/memory. The software compiler makes all sorts of optimizations when it comes to loading data into registers, writing it back to the cache (https://www.inf.ed.ac.uk/teaching/courses/copt/lecture-7.pdf), and even reordering of instructions (https://stackoverflow.com/questions/22106843/gccs-reordering-of-read-write-instructions). This is all done assuming that the code will be run single-threaded. Hence why any data that is at risk of race-conditions, needs to be manually protected through concurrency algorithms and language constructs such as atomics and volatiles.

In the case of Java volatiles, part of the solution is to force all reads/writes to bypass the local registers, and immediately trigger cache reads/writes instead (https://stackoverflow.com/questions/4633866/is-volatile-expensive). As soon as the data is read/written to the L1 cache, the hardware-coherency protocol takes over and provides guaranteed consistency across all global threads. Thus ensuring that if multiple threads are reading/writing to the same variable, they are all kept in sync with one another. And this is how you can achieve inter-thread coordination in as little as 1ns.

**Tagged:**
atomic,
cache,
concurrency,
CPU,
hardware,

lock,
programming,
volatile

## 11 thoughts on “Myths Programmers Believe about CPU Caches”

**Charles** says:
2018-04-30 at 10:29 am
In step 3 of “Local Cache Miss, Sibling Cache Hit”, you wrote:
> L2 sees that L1-1 has the data in E (or M) state

Can it really be in state M? L1-1 can’t just downgrade from M to S without writing back its modifications, can it?

☐ Reply

**OutlookZen** says:
2018-04-30 at 11:27 am
Thanks. I’ve updated the line you’ve quoted, to note that in this scenario, modified data will need to be sent back along with the Ack.

☐ Reply

**Charles** says:
2018-04-30 at 11:32 am
Thank you – I just wanted to check that I understood what’s going on!

Pingback: Newsy programistyczne 2018-05-06 – DevNation
Pingback: New top story on Hacker News: Myths Programmers Believe about CPU Caches – News about world
Pingback: New top story on Hacker News: Myths Programmers Believe about CPU Caches – World Best News

**giovanni deretta** says:
2018-08-02 at 11:39 am
Good article, but regarding volatile (and memory barriers in general), it is not just about bypassing registers, but also enforcing coherency of other internal CPU structures (on x86 that’s mostly the write buffer). You are completely right that volatile and barriers have nothing to do with caches.

☐ Reply

Pingback: Myths Programmers Believe about CPU Caches | Infozonic
Pingback: New top story on Hacker News: CPU cache misconceptions, and the MESI cache coherence protocol – ÇlusterAssets Inc.,

**Elena** says:
2018-11-19 at 11:38 am
Could you roughly specify the % of cache hit-rate in modern CPUs. What it depends on? What cache replacement algorithms are currently used? Thanks.

◻ Reply

**OutlookZen** says:
2018-11-19 at 4:44 pm
I don’t know this off the top of my head. It changes every few years, with new architectures. Cache hit rates depend dramatically on the exact applications being run, the cores that are assigned to run those threads, and the CPU’s cache size. You’ll have to dig into some benchmarks to find more precise numbers. The cache eviction algorithm is almost always a psuedo-lru, though the exact details very with architecture

◻ Reply