Mechanical Sympathy

Hardware and software working together in harmony

Thursday, 14 February 2013

CPU Cache Flushing Fallacy

Even from highly experienced technologists I often hear talk about how certain operations cause a CPU cache to "flush". This seems to be illustrating a very common fallacy about how CPU caches work, and how the cache sub-system interacts with the execution cores. In this article I will attempt to explain the function CPU caches fulfil, and how the cores, which execute our programs of instructions, interact with them. For a concrete example I will dive into one of the latest Intel x86 server CPUs. Other CPUs use similar techniques to achieve the same ends.

Most modern systems that execute our programs are shared-memory multi-processor systems in design. A shared-memory system has a single memory resource that is accessed by 2 or more independent CPU cores. Latency to main memory is highly variable from 10s to 100s of nanoseconds. Within 100ns it is possible for a 3.0GHz CPU to process up to 1200 instructions. Each Sandy Bridge core is capable of retiring up to 4 instructions-per-cycle (IPC) in parallel. CPUs employ cache sub-systems to hide this latency and allow them to exercise their huge capacity to process instructions. Some of these caches are small, very fast, and local to each core; others are slower, larger, and shared across cores. Together with registers and main-memory, these caches make up our non-persistent memory hierarchy.

Next time you are developing an important algorithm, try pondering that a cache-miss is a lost opportunity to have executed ~500 CPU instructions! This is for a single-socket system, on a multi-socket system you can effectively double the lost opportunity as memory requests cross socket interconnects.

Memory Hierarchy

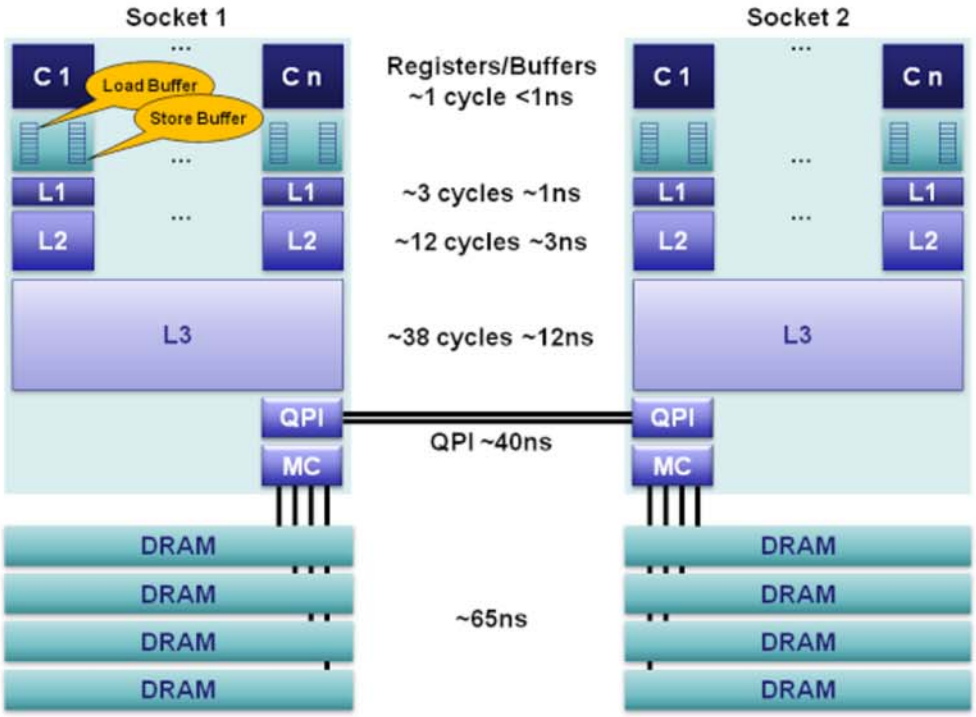
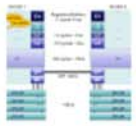


Figure 1.

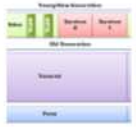
For the circa 2012 Sandy Bridge E class servers our memory hierarchy can be decomposed as follows:

1. **Registers**: Within each core are separate register files containing 160 entries for integers and 144 floating point numbers. These registers are accessible within a single cycle and constitute the fastest memory available to our execution cores. Compilers will allocate our local variables and function arguments to these registers. Compilers allocate to subset of registers know as the architectural registers, then the hardware expands on these as it runs instructions in parallel and out-of-order. Compilers are aware of out-of-order and parallel execution ability for given processor, and order instruction streams and register allocation to take advantage of this. When hyperthreading is enabled these registers are shared between the co-located hyperthreads.
2. **Memory Ordering Buffers (MOB)**: The MOB is comprised of a 64-entry load and 36-entry store buffer. These buffers are used to track in-flight operations while waiting on the cache sub-system as instructions get executed out-of-order. The store buffer is a fully associative queue that can be searched for existing store operations, which have been queued when waiting on the L1 cache. These buffers enable our fast processors to run without blocking while data is transferred to and from the cache sub-system. When the processor issues reads and writes they can can come back out-of-order. The MOB is used to disambiguate the load and store ordering for compliance to the published memory model. Instructions are executed out-of-order in addition to our loads and stores that can come back out-of-order from the cache sub-system. These buffers enable an ordered view of the world to be re-constructed for what is expected according to the memory model.

Popular Posts



CPU Cache Flushing Fallacy
Even from highly experienced technologists I often hear talk about how certain operations cause a CPU cache to "flush". This see...



Java Garbage Collection Distilled
Serial, Parallel, Concurrent, CMS, G1, Young Gen, New Gen, Old Gen, Perm Gen, Eden, Tenured, Survivor Spaces, Safepoints, and the hundreds ...

Memory Access Patterns Are Important
In high-performance computing it is often said that the cost of a cache-miss is the largest performance penalty for an algorithm. For many...

Native C/C++ Like Performance For Java Object Serialisation
Do you ever wish you could turn a Java object into a stream of bytes as fast as it can be done in a native language like C++? If you use s...

Compact Off-Heap Structures/Tuples In Java
In my last post I detailed the implications of the access patterns your code takes to main memory. Since then I've had a lot of quest...

Single Writer Principle
When trying to build a highly scalable system the single biggest limitation on scalability is having multiple writers contend for any item o...

False Sharing

3. **Level 1 Cache**: The L1 is a core-local cache split into separate 32K data and 32K instruction caches. Access time is 3 cycles and can be hidden as instructions are pipelined by the core for data already in the L1 cache.
4. **Level 2 Cache**: The L2 cache is a core-local cache designed to buffer access between the L1 and the shared L3 cache. The L2 cache is 256K in size and acts as an effective queue of memory accesses between the L1 and L3. L2 contains both data and instructions. L2 access latency is 12 cycles.
5. **Level 3 Cache**: The L3 cache is shared across all cores within a socket. The L3 is split into 2MB segments each connected to a ring-bus network on the socket. Each core is also connected to this ring-bus. Addresses are hashed to segments for greater throughput. Latency can be up to 38 cycles depending on cache size. Cache size can be up to 20MB depending on the number of segments, with each additional hop around the ring taking an additional cycle. The L3 cache is inclusive of all data in the L1 and L2 for each core on the same socket. This inclusiveness, at the cost of space, allows the L3 cache to intercept requests thus removing the burden from private core-local L1 & L2 caches.
6. **Main Memory**: DRAM channels are connected to each socket with an average latency of ~65ns for socket local access on a full cache-miss. This is however extremely variable, being much less for subsequent accesses to columns in the same row buffer, through to significantly more when queuing effects and memory refresh cycles conflict. 4 memory channels are aggregated together on each socket for throughput, and to hide latency via pipelining on the independent memory channels.
7. **NUMA**: In a multi-socket server we have non-uniform memory access. It is non-uniform because the required memory maybe on a remote socket having an additional 40ns hop across the QPI bus. Sandy Bridge is a major step forward for 2-socket systems over Westmere and Nehalem. With Sandy Bridge the QPI limit has been raised from 6.4GT/s to 8.0GT/s, and two lanes can be aggregated thus eliminating the bottleneck of the previous systems. For Nehalem and Westmere the QPI link is only capable of ~40% the bandwidth that could be delivered by the memory controller for an individual socket. This limitation made accessing remote memory a choke point. In addition, the QPI link can now forward pre-fetch requests which previous generations could not.

Associativity Levels

Caches are effectively hardware based hash tables. The hash function is usually a simple masking of some low-order bits for cache indexing. Hash tables need some means to handle a collision for the same slot. The associativity level is the number of slots, also known as ways or sets, which can be used to hold a hashed version of an address. Having more levels of associativity is a trade off between storing more data vs. power requirements and time to search each of the ways.

For Sandy Bridge the L1D and L2 are 8-way associative, the L3 is 12-way associative.

Cache Coherence

With some caches being local to cores, we need a means of keeping them coherent so all cores can have a consistent view of memory. The cache sub-system is considered the "source of truth" for mainstream systems. If memory is fetched from the cache it is never stale; the cache is the master copy when data exists in both the cache and main-memory. This style of memory management is known as write-back whereby data in the cache is only written back to main-memory when the cache-line is evicted because a new line is taking its place. An x86 cache works on

Memory is stored within the cache system in units know as cache lines. Cache lines are a power of 2 of contiguous bytes which are typically...

Memory Barriers/Fences

In this article I'll discuss the most fundamental technique in concurrent programming known as memory barriers, or fences, that make the...

Simple Binary Encoding

Financial systems communicate by sending and receiving vast numbers of messages in many different formats. When people use terms like "...

Fun with my-Channels Nirvana and Azul Zing

Since leaving LMAX I have been neglecting my blog a bit. This is not because I have not been doing anything interesting. Quite the opposi...

Blog Archive

- ► 2014 (1)
- ▼ 2013 (5)
  - ► August (1)
  - ► July (1)
  - ► June (1)
  - ▼ February (1)
    - CPU Cache Flushing Fallacy
  - ► January (1)
- ► 2012 (7)

blocks of data that are 64-bytes in size, known as a cache-line. Other processors can use a different size for the cache-line. A larger cache-line size reduces effective latency at the expense of increased bandwidth requirements.

To keep the caches coherent the cache controller tracks the state of each cache-line as being in one of a finite number of states. The protocol Intel employs for this is MESIF, AMD employs a variant know as MOESI. Under the MESIF protocol each cache-line can be in 1 of the 5 following states:

1. **Modified**: Indicates the cache-line is dirty and must be written back to memory at a later stage. When written back to main-memory the state transitions to Exclusive.
2. **Exclusive**: Indicates the cache-line is held exclusively and that it matches main-memory. When written to, the state then transitions to Modified. To achieve this state a Read-For-Ownership (RFO) message is sent which involves a read plus an invalidate broadcast to all other copies.
3. **Shared**: Indicates a clean copy of a cache-line that matches main-memory.
4. **Invalid**: Indicates an unused cache-line.
5. **Forward**: Indicates a specialised version of the shared state i.e. this is the designated cache which should respond to other caches in a NUMA system.

To transition from one state to another, a series of messages are sent between the caches to effect state changes. Previous to Nehalem for Intel, and Opteron for AMD, this cache coherence traffic between sockets had to share the memory bus which greatly limited scalability. These days the memory controller traffic is on a separate bus. The Intel QPI, and AMD HyperTransport, buses are used for cache coherence between sockets.

The cache controller exists as a module within each L3 cache segment that is connected to the on-socket ring-bus network. Each core, L3 cache segment, QPI controller, memory controller, and integrated graphics sub-system are connected to this ring-bus. The ring is made up of 4 independent lanes for: *request*, *snoop*, *acknowledge*, and 32-bytes *data* per cycle. The L3 cache is inclusive in that any cache-line held in the L1 or L2 caches is also held in the L3. This provides for rapid identification of the core containing a modified line when snooping for changes. The cache controller for the L3 segment keeps track of which core could have a modified version of a cache-line it owns.

If a core wants to read some memory, and it does not have it in a Shared, Exclusive, or Modified state; then it must make a read on the ring bus. It will then either be read from main-memory if not in the cache sub-systems, or read from L3 if clean, or snooped from another core if Modified. In any case the read will never return a stale copy from the cache sub-system, it is guaranteed to be coherent.

Concurrent Programming

If our caches are always coherent then why do we worry about visibility when writing concurrent programs? This is because within our cores, in their quest for ever greater performance, data modifications can appear out-of-order to other threads. There are 2 major reasons for this.

Firstly, our compilers can generate programs that store variables in registers for relatively long periods of time for performance reasons, e.g. variables used repeatedly within a loop. If we need these variables to be visible across cores then the updates must not be register allocated. This is achieved in C by qualifying a variable as "`volatile`". Beware that C/C++ `volatile` is inadequate for telling the compiler not to reorder other instructions. For this you

need memory fences/barriers.

The second major issue with ordering we have to be aware of is a thread could write a variable and then, if it reads it shortly after, could see the value in its store buffer which may be older than the latest value in the cache sub-system. This is never an issue for algorithms following the Single Writer Principle. The store buffer also allows a load instruction to get ahead of an older store and is thus an issue for the likes of the Dekker and Peterson lock algorithms. To overcome these issues, the thread must not let a sequential consistent load get ahead of the sequentially consistent store of the value in the local store buffer. This can be achieved by issuing a fence instruction. The write of a `volatile` variable in Java, in addition to never being register allocated, is accompanied by a full fence instruction. This fence instruction on x86 has a significant performance impact by preventing progress on the issuing thread until the store buffer is drained. Fences on other processors can have more efficient implementations that simply put a marker in the store buffer for the search boundary, e.g. the Azul Vega does this.

If you want to ensure memory ordering across Java threads when following the Single Writer Principle, and avoid the store fence, it is possible by using the `j.u.c.Atomic(Int|Long|Reference).lazySet()` method, as opposed to setting a `volatile` variable.

The Fallacy

Returning to the fallacy of "flushing the cache" as part of a concurrent algorithm. I think we can safely say that we never "flush" the CPU cache within our user space programs. I believe the source of this fallacy is the need to flush, mark or drain to a point, the store buffer for some classes of concurrent algorithms so the latest value can be observed on a subsequent load operation. For this we require a memory ordering fence and not a cache flush.

Another possible source of this fallacy is that L1 caches, or the TLB, may need to be flushed based on address indexing policy on a context switch. ARM, previous to ARMv6, did not use address space tags on TLB entries thus requiring the whole L1 cache to be flushed on a context switch. Many processors require the L1 instruction cache to be flushed for similar reasons, in many cases this is simply because instruction caches are not required to be kept coherent. The bottom line is, context switching is expensive and a bit off topic, so in addition to the cache pollution of the L2, a context switch can also cause the TLB and/or L1 caches to require a flush. Intel x86 processors require only a TLB flush on context switch.

26 comments:

Terry A. Davis 14 February 2013 14:14

SparrowOS is always identity mapped so never has memory map penalties.

Reply

Evi1M4chine 14 February 2013 14:52

What about DMA? Is the L3 cache notified if a modification of RAM by another device occurs? (Because if not, coherency would obviously be lost.)

Reply

Replies

Martin Thompson 14 February 2013 23:37

Depending on the hardware the IO device may be able to notify the cache controller of the impending write operation. If this is not available the operating system can provide support which a device driver developer will be aware of.

Sandy Bridge E class servers as discussed in the article introduce DDIO which allows direct transfer from a network card to the L3 cache for reduced latency.

Reply

Candy 14 February 2013 16:16

X86 processors only do a limited TLB flush, the pages marked as global are kept. Depending on your OS' use of this flag that could mean they're not flushed at all.

Reply

vemv 14 February 2013 16:42

As someone not particularly experienced at concurrent programming, it is still unclear to me what the fallacy is about. I never heard about "flushing the cache" - any examples of such incorrect reasoning? Finally, what are the implications of this fallacy being, well, a fallacy?

Reply

Replies

Martin Thompson 14 February 2013 23:24

I've been meaning to write a blog about x86 cache sub-systems for some time. Lately I've heard a few people say things that made me think caches are not that well understood. The fallacy is that many assume they know how the cache sub-system works. With a better understanding of how hardware works we can all write code that shows more sympathy and thus executes more efficiently. At the very least we can all gain an appreciation for the good work our hardware friends are putting in.

I find my own knowledge is expanded by writing and the subsequent helpful feedback it brings.

Understanding the memory hierarchy is key to getting the best out of tools like taskset and numactrl for an example of implications.

Reply

translatedcode 14 February 2013 19:26

"ARM does not use address space tags on TLB entries" -- are you sure that's correct? ARM supports ASIDs (address space IDs) on TLB and icache entries. I admit I haven't looked at the Linux kernel to see what operations it's actually doing on a context switch, though.

Reply

Replies

Martin Thompson 14 February 2013 21:22

Looks like you are right and ASIDs came in with ARMv6. I've obviously not kept up sufficiently with ARM. I have noticed that Samsung suggest Linux still flushes on a context switch but I've not been able to verify this.

http://opensrc.sec.samsung.com/document/uc-linux-04_sait.pdf

cheriff 17 February 2013 21:50

That pdf talks about running uClinux on an ARM9 (ie ARMv5) processor that indeed does not have a ASID-tagged TLB; requiring a full flush on context switch. Additionally, those processors did the cache lookup based on the virtual address (I presume so HW could avoid blocking on TLB-lookup or do it in parallel) so to avoid cross address space contamination you had to[1] flush+invalidate on switch.

Theses days on ARMv6, v7 and the upcoming v8 (ie ARM11, CortexA8/A9, etc) TLB entries are tagged by ASID and the caches are indexed by physical address so much of this becomes less relevant.

[1] If an OS could guarantee processes exist in non-overlapping virtual address spaces (as does uClinux), this could be relaxed., and there were HW tricks which could assist with this. Search for FASS (Fast Address Space Switching) by Wiggins, and the use of PID Relocation to generate globally unique Modified Virtual Addresses as an alternate technique to uClinux for achieving this.

Reply

Riss 14 February 2013 20:28

When changing rounding modes, as specified in IEEE-754, on x86 the FPU pipeline is unfortunately flushed.

Reply

Matt Godbolt 15 February 2013 12:41

Thanks for a great write up. I have a couple of minor comments:

My understanding is that the L1 cache _is_ flushed on some context switches. As the L1 cache circuitry basically runs at main core speed, it is indexed not by logical address, but by physical address. To put it behind the TLB would be too slow. As such when the page table is modified the L1 cache needs to be (partially) flushed too.

One other minor comment: you correctly state there are hundreds of physical registers on x86; however "Compilers will allocate our local variables and function arguments to these registers." is a little disingenuous to the on-chip scheduler/reservation station/flow analysis/renaming circuitry. The compiler allocates to the (considerably fewer) architectural registers, and then the on-chip wizardry allocates those to the available physical registers on demand.

Thanks again!

-- Matt

Reply

Replies

Matt Godbolt 15 February 2013 12:56

...I shouldn't post before coffee...

I meant to say the L1 is indexed by "logical, not by physical" address. My point's still valid I just misspoke :)

Martin Thompson 15 February 2013 20:33

Coffee fuelling the technology industry. I love the stuff too!

Thanks for the feedback. I agree the registers part could do with more description, in the reduction I may have over simplified this to the point of being misleading.

On the L1D, are you sure about this? My understanding is that it is physically tagged and the TLB lookup happens in parallel on the set with at most one in the set chosen. I'm confirming this with my hardware friends. Check out this paper for an overview in section 1.

http://www.cs.utah.edu/~zfang/islped12.pdf

Matt Godbolt 28 January 2014 12:51

So it took me a year to reply (sorry!) but I've been unable to find the reference I thought I had to the physical mapping of the L1 caches...apologies for tardiness of reply and for the confusing post!

Matt Godbolt 28 January 2014 13:02

I spoke too soon. A Google search (who'd've thought, eh?) turned up a StackOverflow question that has some

more information: http://stackoverflow.com/questions/19039280/physical-or-virtual-addressing-is-used-in-processors-x86-x86-64-for-caching-in-t

..."they're are physically tagged and virtually indexed" ... "L1 is 32k, 8-way associative, it means that it uses 64 sets, so you need only addr bits 6 to 11 in order to find the correct set. As it happens to be, virtual and physical addresses are the same in these bits, so you can lookup the DTLB in parallel with reading a cache set"

So, there's definitely a logical address component, but it doesn't really amount to anything as the physical address is still used to check if an entry is for the right address. As such, there's clearly no need to flush the L1 on context switches after all.

Reply

Thomas Sanchez 15 February 2013 18:40

Hi,
Short but very interesting post.
You may be interested in Herb Sutter video about memory model and concurrency, there is a part on the store buffer that is very good too:

http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

Reply

Brian Ouellette 20 February 2013 00:15

I've seen confusion between this and a pipeline flush (due to atomic instructions/memory barriers or branch prediction misses) as well.

Reply

Bill Broadley 21 February 2013 09:56

Heh, the QPI latency sounded high to me, so I tested:
TLB friendly, 64MB random read, numa local = 59.4ns
TLB friendly, 64MB random read, 100% remote = 105.6ns
Non-TLB friendly, 64MB random read, numa local = 98.3ns
Non-TLB friendly, 64MB random read, 100% remote = 168.5ns

Good overview and I agree that 40ns is a good number for the QPI latency.

One thing that surprised me for randomly reading main memory how much it helps to have additional threads. Each of 8 memory channels on a dual socket can handle a cache miss in approximately 65ns (agreeing with the original post).

So additional threads have the following benefits:
* with a random distribution each additional thread increases the chances
that all 8 memory channels are busy

* as the 8 memory channels get saturated the L3 cache misses queue
* any queued misses do NOT have to wait the full memory latency, but instead the Main memory - L3 latency. With the L3 around 1/4th of the latency to main memory this can be pretty helpful.

So with 1 thread I see a random cache line per 59.53ns.
8 threads 11.95ns.
12 threads 8.55ns
24 threads 5.25ns
48 threads (12 physical cores/24 threads) 5.32ns

Despite auditing the number of tlb misses and cache misses I had a hard time believing the 5.25ns number. But my main memory (59.5 ns) - L3 (16ns or so) =
43.5ns. You get that on each of 8 channels so 43.5/8 = 5.43 ns. I think the additional gains are for some additional pipe-lining, or maybe having more dram pages open.

Reply

Rational Root 4 April 2013 09:53

Regarding c# volatile, my understanding is that volatile implements a memory barrier on all reads and writes to that variable

See http://blogs.msdn.com/b/brada/archive/2004/05/12/volatile-and-memorybarrier.aspx

"This is more efficient than using volatile because a volatile field requires all accesses to be barriers and this effects some performance optimizations. "

Reply

Replies

Martin Thompson 9 April 2013 15:06

In the blog I reference C/C++ and not C# which has different behaviour. My understanding of C# is that all writes to fields follow StoreStore (i.e. stores are not reordered with other stores, aka TSO like x86) semantics as a software, and not hardware, memory barrier. To achieve a hardware memory barrier or full fence you would need to use Thread.MemoryBarrier().

http://msdn.microsoft.com/en-us/magazine/cc163715.aspx#S4

Martin Thompson 9 April 2013 15:11

Also the C# volatile access come with a full fence to ensure sequential consistency like in Java and C/C++ 11.

Reply

Gavin Lowe 2 January 2014 13:06

I'm trying to reconcile this blog with other seemingly definitive articles (such as http://gee.cs.oswego.edu/dl/cpj/jmm.html and http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html#synchronization) that explicitly talk about flushing the cache.

Reply

Replies

Martin Thompson 2 January 2014 13:15

As I mentioned even experts mistakenly discuss this subject :-) Maybe you should raise this subject on the concurrency interest list and see how people respond.

http://altair.cs.oswego.edu/mailman/listinfo/concurrency-interest

J.J 23 January 2014 15:38

The cookbook http://gee.cs.oswego.edu/dl/jmm/cookbook.html also mentioned: "holds stores waiting to be flushed to memory".

And especially the StoreLoad Barriers part :

"ensures that Store1's data are made visible to other processors (i.e., flushed to main memory) "

Interesting, i guess author sometimes might think this might be more easier to "understand" but it is really confusing sometimes also.

Martin Thompson 23 January 2014 20:29

I know, I've read that. As I said even experts are misquoting on this :-)

J.J 24 January 2014 02:47

I am reading a paper about Memory Barrier and CPU Coherence Protocol, write to memory happened when "Write Back', but so many blog/article talked about "Flush to main memory" when "Barrier", this really confused me.

This blog is great, It made my day.

Reply

Enter your comment...

Comment as: Select profile...

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)