Sunday, 24 July 2011

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 memory state within a processor visible to other processors.

CPUs have employed many techniques to try and accommodate the fact that CPU execution unit performance has greatly outpaced main memory performance. In my “Write Combining” article I touched on just one of these techniques. The most common technique employed by CPUs to hide memory latency is to pipeline instructions and then spend significant effort, and resource, on trying to re-order these pipelines to minimise stalls related to cache misses.

When a program is executed it does not matter if its instructions are re-ordered provided the same end result is achieved. For example, within a loop it does not matter when the loop counter is updated if no operation within the loop uses it. The compiler and CPU are free to re-order the instructions to best utilise the CPU provided it is updated by the time the next iteration is about to commence. Also over the execution of a loop this variable may be stored in a register and never pushed out to cache or main memory, thus it is never visible to another CPU.

CPU cores contain multiple execution units. For example, a modern Intel CPU contains 6 execution units which can do a combination of arithmetic, conditional logic, and memory manipulation. Each execution unit can do some combination of these tasks. These execution units operate in parallel allowing instructions to be executed in parallel. This introduces another level of non-determinism to program order if it was observed from another CPU.

Finally, when a cache-miss occurs, a modern CPU can make an assumption on the results of a memory load and continue executing based on this assumption until the load returns the actual data.

Provided “program order” is preserved the CPU, and compiler, are free to do whatever they see fit to improve performance.

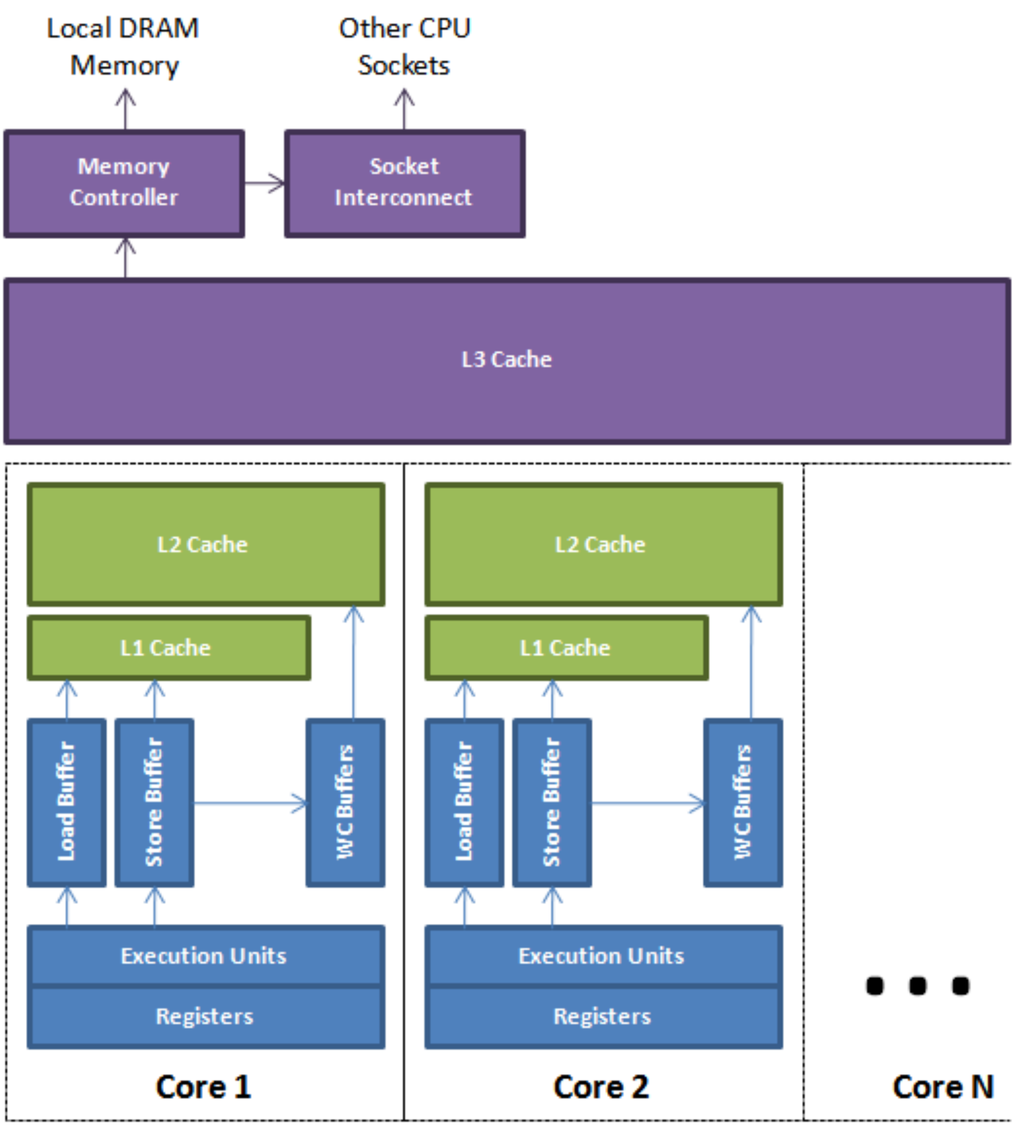

Figure 1.

Loads and stores to the caches and main memory are buffered and re-ordered using the load, store, and write-combining buffers. These buffers are associative queues that allow fast lookup. This lookup is necessary when a later load needs to read the value of a previous store that has not yet reached the cache. Figure 1 above depicts a simplified view of a modern multi-core CPU. It shows how the execution units can use the local registers and buffers to manage memory while it is being transferred back and forth from the cache sub-system.

In a multi-threaded environment techniques need to be employed for making program results visible in a timely manner. I will not cover cache coherence in this article. Just assume that once memory has been pushed to the cache then a protocol of messages will occur to ensure all caches are coherent for any shared data. The techniques for making memory visible from a processor core are known as memory barriers or fences.

Memory barriers provide two properties. Firstly, they preserve externally visible program order by ensuring all instructions either side of the barrier appear in the correct program order if observed from another CPU and, secondly, they make the memory visible by ensuring the data is propagated to the cache sub-system.

Memory barriers are a complex subject. They are implemented very differently across CPU architectures. At one end of the spectrum there is a relatively strong memory model on Intel CPUs that is more simple than say the weak and complex memory model on a DEC Alpha with its partitioned caches in addition to cache layers. Since x86 CPUs are the most common for multi-threaded programming I'll try and simplify to this level.

Store Barrier

A store barrier, "*sfence*" instruction on x86, forces all store instructions prior to the barrier to happen before the barrier and have the store buffers flushed to cache for the CPU on which it is issued. This will make the program state visible to other CPUs so they can act on it if necessary. A good example of this in action is the following simplified code from the BatchEventProcessor in the Disruptor. When the sequence is updated other consumers and producers know how far this consumer has progressed and thus can take appropriate action. All previous updates to memory that happened before the barrier are now visible.

```
private volatile long sequence = RingBuffer.INITIAL_CURSOR_VALUE;

// from inside the run() method

T event = null;
long nextSequence = sequence.get() + 1L;
while (running)
{
    try
    {
        final long availableSequence = barrier.waitFor(nextSequence);

        while (nextSequence <= availableSequence)
        {
            event = ringBuffer.get(nextSequence);
            boolean endOfBatch = nextSequence == availableSequence;
            eventHandler.onEvent(event, nextSequence, endOfBatch);
            nextSequence++;
        }

        sequence.set(nextSequence - 1L);
        // store barrier inserted here !!!
    }
    catch (final Exception ex)
    {
        exceptionHandler.handle(ex, nextSequence, event);
        sequence.set(nextSequence);
        // store barrier inserted here !!!
        nextSequence++;
    }
}
```

Load Barrier

A load barrier, “*lfence*” instruction on x86, forces all load instructions after the barrier to happen after the barrier and then wait on the load buffer to drain for that CPU. This makes program state exposed from other CPUs visible to this CPU before making further progress. A good example of this is when the BatchEventProcessor sequence referenced above is read by producers, or consumers, in the corresponding barriers of the Disruptor.

Full Barrier
A full barrier, "*mfence*" instruction on x86, is a composite of both load and store barriers happening on a CPU.

Java Memory Model
In the Java Memory Model a *volatile* field has a store barrier inserted after a write to it and a load barrier inserted before a read of it. Qualified *final* fields of a class have a store barrier inserted after their initialisation to ensure these fields are visible once the constructor completes when a reference to the object is available.

Atomic Instructions and Software Locks
Atomic instructions, such as the “*lock* ...” instructions on x86, are effectively a full barrier as they lock the memory sub-system to perform an operation and have guaranteed total order, even across CPUs. Software locks usually employ memory barriers, or atomic instructions, to achieve visibility and preserve program order.

Performance Impact of Memory Barriers
Memory barriers prevent a CPU from performing a lot of techniques to hide memory latency therefore they have a significant performance cost which must be considered. To achieve maximum performance it is best to model the problem so the processor can do units of work, then have all the necessary memory barriers occur on the boundaries of these work units. Taking this approach allows the processor to optimise the units of work without restriction. There is an advantage to grouping necessary memory barriers in that buffers flushed after the first one will be less costly because no work will be under way to refill them.

Martin Thompson at 19:24

Share G+1 25

39 comments:

Xin Wang 3 August 2011 at 03:39

Hi Martin,

I read the artical serval times but I still feel I didn't fully understand when we need a memory barrier. Let's take simple example (single reader single writer circular buffer), can you please help me identify where and why we need memory barriers?

```
static const std::size_t N = 1024;
static const uint64_t read = 0;
static const uint64_t write = 0;
T buffer[N];

//writer:
while(true) {
do {
//W1
} while (write - read == N);
produce(buffer[write & N]);
//W2
++write;
//W3
}

//reader:
while(true) {
do {
//R1
} while (read==write);
consume(buffer[read & N]);
//R2
++read;
//R3
}
```

I am wondering if we need memory barrier on:
//W1 to make sure we get the latest value of variable "read"?
//W2 to make sure that the value in buffer[write & N] is seen by reader before the variable "write" is updated?
//W3 to make the variable "write" observable by reader?
How about reader?

Thank you!

Reply

Martin Thompson 3 August 2011 at 08:07

Hi,

On x64 you need an mfence after, or better still do a "lock addq" on, the read and write variables when incrementing. That is at point W3 and R3 in your code. Note R3 needs to be a mfence or sfence and not a lfence.

You need this because you want to ensure the "produce(buffer[write & N])" happens before updating the counter to make it visible. By the way N needs to be N - 1 for the mask to work in your code.

If you are on a weaker memory model than x64 you would need a load barrier at W1 and R1.

Hope this helps.

Martin...

Reply

Xin Wang 3 August 2011 at 12:26

Thanks Martin.
"You need this because you want to ensure the "produce(buffer[write & N])" happens before updating the counter to make it visible."
Are you talking about mem barrier at W2 and R2?

Reply

Martin Thompson 3 August 2011 at 13:59

You need W3, don't need W2, on the producer side to make both the production of the item in the buffer and the write sequence increment visible in the correct order.

You need the R3 to be a store barrier to ensure the producer can see the progress of the consumer without over running it.

W2 and R2 are not needed on any architecture. W1 and R1 might be needed as load barriers on some architectures.

Reply

Replies

johanh 13 April 2012 at 18:47

Hi Martin,

I don't understand why W2 wouldn't be necessary in Xin Wang's above.

My understanding is (and it could very well be wrong) that the sfence instruction makes store instructions preceding the sfence globally visible before store instructions that follow the sfence. In Xin's example, we have to be sure that the produced data is globally visible before the incremented counter is visible. If we don't put an sfence between these stores, isn't there a risk that the reader is seeing the incremented counter before the new data in the buffer? Putting the fence after counter is incremented only makes sure that the counter is visible before any stores following the counter increment, but that is not really what we need here.

Martin Thompson 14 April 2012 at 14:31

You are correct, my mistake.

It should be noted that on x86_64 there is no need for a hardware memory barrier at all because the code respects the Single Writer Principle. Other platforms are different!

However this does not make any guarantees about the compiler when we go with C/C++. This code needs the synchronisation variables to be declared volatile and access to be wrapped using functions like the following:

```
/*
* Load a sequence and ensure instructions after, happen after.
*/
static inline int64 load_acquire(volatile int64* sequencePtr)
{
int64 sequence = *sequencePtr;
__asm__ __volatile__("" ::: "memory");
return sequence;
}

/*
* Store a sequence and ensure previous stores, happen before.
*/
static inline void store_release(volatile int64* sequencePtr, int64 sequence)
{
__asm__ __volatile__("" ::: "memory");
*sequencePtr = sequence;
}
```

chop suey 28 January 2013 at 19:35

Is x86 hotspot using the above fences? Looks to me more like a xchgq http://hg.openjdk.java.net/jdk7/jdk7/hotspot/file/4fc084dac61e/src/os_cpu/linux_x86/vm/orderAccess_linux_x86.inline.hpp

Martin Thompson 29 January 2013 at 14:21

The release_store options are just memory stores if you look at the code. Depending on what compiler and setting are used this may be an issue. The XCHG(implicit lock instruction) is not used in the release_store methods in the linked file. The store_release will simply generate a MOV instruction which is sufficient for the hardware memory model. The ASM example I give above is a directive to GCC to not reorder the code.

Reply

Xin Wang 5 August 2011 at 04:10

Hi Martin, I am still a little confused:
To ensure "produce(buffer(write & (N-1))" happens before the updated counter is visible, shouldn't we insert a store barrier between producer() and ++write?
Can you please recommend some articles or books if I want to learn the memory model/memory barrier?

Reply

Martin Thompson 5 August 2011 at 07:08

It is not necessary to have a memory barrier on every line. Memory barriers are used to create reference points when you can determine an order. The memory model is different for most CPUs. The link below is for the Intel x64 processors which is one of the more simple to understand. Others such as the Dec Alpha, on which the Linux memory model is based, are much more complicated. The more complex models are usually a super set of the less complex models.

http://www.multicoreinfo.com/research/papers/2008/damp08-intel64.pdf

The CPU manufactures publish their memory models so compiler writers can understand them.

Reply

Siarhei Kuryla 19 January 2012 at 09:09

Hi Martin,

Regarding your comment:

"You need this because you want to ensure the "produce(buffer[write & N])" happens before updating the counter to make it visible. By the way N needs to be N - 1 for the mask to work in your code."

The Intel x86 memory orderding is saying that "Writes by a single processor are observed in the same order by all processors". I am not sure how to interpret this, but I would think that it's guaranteed that these two writes are observed in the same order by other processers on Intel x86 without a fence:

```
produce(buffer[write & N]);
++write;
```

I have spent quote some time reading Intel x86 documentation and but a bit confused. Because of all guarantee that Intel x86 gives us I don't see applicability of sfence except for the case to prevent reordering loads with older stores.

Thank you!

Reply

Replies

Martin Thompson 19 January 2012 at 10:03

Siarhei,

You are correct in that a fence is not required on x86 if you are sure a variable only has a single writer. It is required if you have multiple writers. This is because the store buffer needs flushed so you see the latest value and not your own last write. Other processors don't have such a strong memory model, e.g. ARM, MIPS and ALPHA. From Java you have limited options to make sure a variable is not register allocated. If it is register allocated in a loop, then it may not be made visible. To get around this the variable needs to be declared volatile, therefore generating a lock instruction with a fence. There is a trick that works with Atomic*.lazySet() but this is not officially defined in the Java memory model.

Siarhei Kuryla 19 January 2012 at 18:27

Martin,

My understanding is that the Atomic*.lazySet() only prevents a variable from being register allocated and does not use a fence. Is it correct?

Thank you!

Martin Thompson 20 January 2012 at 09:32

Siarhei,

Atomic*.lazySet() not only prevents the variable being register allocated it also imposing ordering. This ordering is only at a software level as there is no explicit hardware fence. If you follow the code down into unsafe it calls putOrdered*().

Jimmy Zhang 9 March 2012 at 05:59

Hi Martin,
as you said above about multiple writers,only the barrier-w3 can ensure "buffer[write & N])" and "++write" be executed in a atomic way? I think the 2 lines are in a Critical Section.

Martin Thompson 12 March 2012 at 00:59

Jimmy,

The above algorithm will only work with single producer and single consumer. Multiple writers of the "++write" would result in a corrupt sequence.

Reply

Changgeng 6 March 2013 at 02:49

Hi Martin,

Does the barrier just work on a specific field(by volatile modifier), or all the memory in cache?

"A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking. " (http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html)

So it says here it can only guarantee the visibility of the volatile fields.

Also I'm reading the white paper of Distruptor, it seems it just rely on the ProducerBarrier and ConsumerBarrier to guarantee the visibility on ValueEntry.
Just wonder if my understanding to the description in javadoc is wrong, or this is an implementation specific behavior of oracle jdk.

Reply

Replies

Martin Thompson 7 March 2013 at 11:40

The barrier is associated with release and subsequent acquire of a specific field. What this means is that when you write the volatile field then all writes before that write in program order are visible to any thread that observes the write when they read the field. This is defined in the Java Memory Model for all JVM from Java 5 onwards.

Reply

ParadoxG 18 April 2013 at 01:32

Fantastic article! I have read a few of your articles. They are high quality and great for learning about low level architectures.

Reply

Mcoolive 15 June 2013 at 13:56

Hi, thank you for this article. I think you succeed to explain simply these complex things. It reminded me some details I forgot because I don't use them in my daily work.

Reply

David Sinclair 5 May 2014 at 15:27

Really enjoy your articles Martin. What is the cost associated with a sfence instruction? The reason I am asking is because I try to make most of my objects immutable, i.e. final fields. But if there is a performance penalty and the object is not published to multiple threads, does it make more sense to make these fields non-final? I realize this would be a micro-optimization, but for certain system every nano-second counts.

thanks!

Reply

Replies

Martin Thompson 5 May 2014 at 17:01

On x86 no hardware fence is required for final fields. The x86 memory model is Total Store Order (TSO) and provided the compiler does not reorder the stores then no fence is required. Use your final fields without any performance worries!

David Sinclair 5 May 2014 at 17:32

Thanks for the reply Martin.

Can you explain what you mean when you say this then? "Qualified final fields of a class have a store barrier inserted after their initialisation to ensure these fields are visible once the constructor completes when a reference to the object is available." Is this only when the compiler reorders the stores?

Martin Thompson 5 May 2014 at 17:41

If a processor is not TSO, e.g. ARM, then a hardware fence is required. On x86 no hardware fence is required because it is TSO. However the compiler requires a software fence to ensure ordering is preserved of the stores.

If you are on x86 their is no performance penalty.

Reply

Nadeem 6 August 2014 at 21:43

Hello Martin,
Thanks for this nice article. I have a question.
Why do we need a load barrier if the store barrier alone exposes the data to the cache subsystem, and "cache coherence" makes sure the data is synchronized between different CPUs ?

For example,

```
int value = 0;
public void A()
{
value = 123;
// Sore barrier. Flush store buffer.

}

public void B()
{
// Load barrier. Why do I need a Load barrier if the store barrier above makes sure the cache is up to date?
Print (value);
}
```

Reply

Replies

Martin Thompson 6 August 2014 at 21:46

Say you wanted to read value in a loop and always get the latest value, or you want a load to be ordered before other loads?

Nadeem 6 August 2014 at 22:09

Actually, I read that barriers have to be paired in order to work correctly and that a barrier in one CPU does not affect the other CPUs. That is what I don't understand. For example, the original code that confuses me is the following C# code which uses full barriers:

Taken from O'Reilly's C# in a Nutshell:

```
class Foo
{
int _answer;
bool _complete;
void A()
{
_answer = 123;
Thread.MemoryBarrier(); // Barrier 1
_complete = true;
Thread.MemoryBarrier(); // Barrier 2
}
void B()
{
Thread.MemoryBarrier(); // Barrier 3
if (_complete)
{
Thread.MemoryBarrier(); // Barrier 4
Console.WriteLine (_answer);
}
}
}
```

The author says: "Barriers 1 and 4 prevent this example from writing “0”. Barriers 2 and 3 provide a freshness guarantee: they ensure that if B ran after A, reading _complete would evaluate to true."

These are full barriers. I understand that Barrier 1 and 4 are needed for order, that is to make sure that answer is written to before _complete. While barrier 4 ensures that the read from _complete happens-before the read from _answer.

Now I don't understand why both Barrier 2 and 3 are needed. Isn't one of them enough ? Let's say barrier 2 flushes the store buffer, then barrier 3 is redundant.

Martin Thompson 8 August 2014 at 16:59

Barrier 2 is required for sequential consistency which can be achieved by waiting on the store buffer to drain. If B() was called in a loop then barrier 3 prevents the read of _complete being hoisted outside the loop which would be possible after inlining.

Nadeem 8 August 2014 at 18:46

I understand that for a loop Barrier 3 is needed, but there is no loop in B(). Assuming there is no loop, is barrier 3 necessary?

My guess and I could be wrong is that cache coherency is not atomic and there is a delay because there exists a queue for cache delivery between CPUs. When the store buffer is flushed out in Barrier 2, _complete will be present in that CPU cache only, but it is not immediately present for the other CPU cache running B(). So Barrier 3 will flush the caching queue. Is this possible ?

Martin Thompson 8 August 2014 at 22:44

You cannot make any assumptions regarding the context in which B() will be called if you designed it as a library.

I do not believe Barrier 3 is about flushing caching queues. Barriers/fences are for ordering and not flushing queues/buffers. Barrier 3 ensures the load of _complete is not ordered back in the stream from its intended position.

Nadeem 9 August 2014 at 03:38

I read in different places that barriers not only are for ordering, but also for flushing. See http://en.wikipedia.org/wiki/MESI_protocol
"A store barrier will flush the store buffer, ensuring all writes have been applied to that CPU's cache. A read barrier will flush the invalidation queue, thus ensuring that all writes by other CPUs become visible to the flushing CPU."

Martin Thompson 13 August 2014 at 08:26

The Wikipedia article is generic. It is not necessary to implement invalidate queues for MESI, and x86 does not. If you look at what x86 assembly instructions get generated for a volatile load in Java you will see it is just a simple MOV instruction. For the normal write back memory, to achieve sequential consistency, x86 only needs a fence for preventing younger loads passing older stores due to the store buffer. We have far more need for soft fences to prevent compiler re-orderings.

Nadeem 14 August 2014 at 21:18

"The Wikipedia article is generic. It is not necessary to implement invalidate queues for MESI, and x86 does not."
This code may not run on x86. What if it runs on a CPU that implements invalidate queues. It's safe to have a barrier then.

Martin Thompson 15 August 2014 at 09:32

If you write code in a language that has a memory model, then the compiler will generate the appropriate ordering instructions for the processor it runs on.

For example a Java load of a volatile field on x86 is just a simple MOV instruction but on other processors, such as Power and ARM, it has to generate additional fences. You don't change the code you write in Java.

Nadeem 15 August 2014 at 16:41

Yes, I think it's the JIT not the compiler that generates ordering instructions and fences. Same thing in .Net. But the fields are not volatile in this case. That's why the barriers are needed. (I think also Java memory model is stricter than .NET)

For example, a volatile on Intel x86 is only a hint for the JIT not to optimize the variable since Intel x86 has a strong memory model (with the exception of write-read reorder).

Anyway, the store buffer is still an issue even on x86.

Martin Thompson 15 August 2014 at 18:02

My understanding of the .Net Memory Model is that it is stronger than the Java Memory Model, particularly for field access - writes to a field have StoreStore ordering.

http://msdn.microsoft.com/en-us/magazine/jj863136.aspx

BTW volatile is not a "hint" to JIT. The interpreter, and code generated by the JIT complier, must produce in very specific behaviour for this synchronising action. In general, memory barriers are way more significant to the compiler optimisation than the hardware.

Nadeem 15 August 2014 at 18:23

By "hint to JIT", I mean the CPU does not do anything with the volatile keyword. It's the JIT interpreter that needs to generate instructions, fences, etc.

"As an interesting side note, the Java programming language takes a different approach. The Java memory model has a slightly stronger definition of “volatile” that doesn’t permit store-load reordering, so a Java compiler on the x86 will typically emit a locked instruction after a volatile write." http://msdn.microsoft.com/en-us/magazine/jj883956.aspx

Reply

francesco 25 July 2015 at 12:08

Hi Martin,
I've read of lot of articles (on blogs & papers too) + Doug Lea 's Cookbook (and the preogrammer's view one edited by Gil Tene) and i'm really confused by the ratio behind the JMM (becouse i don't catch it!)...
I feel that i need to start from the basis to understand these concepts and use it in the right way while programming! Do you suggest a good approach or sequence of lectures to master the JMM concepts (from the high level POWs of volatile,atomics etc to the low level of memory barriers...)? I hope that it wouldn't be necessary to simply memorized all the "rules" but that exist a logic thought that could be applied to deduce all the expected behaviours of the compiler (at least)!

Regards,
Francesco

Reply

Replies

Martin Thompson 25 July 2015 at 14:02

There is a lot to learn. To help I offer courses on this subject.

http://real-logic.co.uk/training.html

Reply

Add comment

About Me

Martin Thompson
London, United Kingdom

Technology geek exploring the capabilities of modern hardware. Available for development, training, performance tuning, and consulting services via Real Logic Limited. Twitter: @mjpt777

View my complete profile