

Java Memory Model Under The Hood

10 Feb 2014

tech

[java](/tag/java.html)

[\(/tag/java.html\)](/tag/java.html)

[membar](/tag/membar.html)

[\(/tag/membar.html\)](/tag/membar.html)

[openjdk](/tag/openjdk.html)

[\(/tag/openjdk.html\)](/tag/openjdk.html)

[internals](/tag/internals.html)

[\(/tag/internals.html\)](/tag/internals.html)

[volatile](/tag/volatile.html)

[\(/tag/volatile.html\)](/tag/volatile.html)

[jmm](/tag/jmm.html)

[\(/tag/jmm.html\)](/tag/jmm.html)

[happens-before](/tag/happens-before.html)

[\(/tag/happens-before.html\)](/tag/happens-before.html)

There are many sources where you can get an idea of what JMM is about, but most of them still leave you with lots of unanswered questions. How does that happens-before thing work? Does using `volatile` result in caches being dropped? Why do we even need a memory model in the first place?

This article is intended to give the readers a level of understanding which allows them to answer all of these questions. It will consist of two large parts; the first of them being a hardware-level outline of what's happening, and the second is indulging in some digging around OpenJDK sources and experimenting. Thus, even if you're not exactly into Java, the first part might still be of interest to you.

🔗 The Hardware-Related Stuff

The engineers that create hardware are working hard on optimizing their products ever further, enabling you to get more and more performance units out of your code. However, it does come at a price of counter-intuitive execution scenarios that your code may display when it is run. There are countless hardware details obscured from our view by abstractions. And abstractions tend to get leaky. (<http://www.joelonsoftware.com/articles/LeakyAbstractions.html>).

🔗 Processor Caches

A request to the main memory is an expensive operation, which can take hundreds of nanoseconds to execute, even on modern hardware. The execution time of other operations, however, has grown considerably smaller over the years, unlike the main memory access. This problem is commonly named as the Memory Wall (http://www.eecs.ucf.edu/~lboloni/Teaching/EEL5708_2006/slides/wulf94.pdf), and the obvious workaround for this is introducing caches. To put it simply, the processor has local copies of the contents of the main memory that it frequently uses. You can read further on different cache structures here (<http://arstechnica.com/gadgets/2002/07/caching/2/>), while we shall move on to the problem of keeping the cached values up to date.

Although there is apparently no problem when you have only one execution unit (referred to as **processor** from now on), things get complicated if you have more than one of those.

How does processor **A** know that processor **B** has modified some value, if **A** has it cached?

Or, more generally, how do you ensure **cache coherency**?

To maintain a consistent view on the state of the world, processors have to communicate with each other. The rules of such communication are called a **cache coherency protocol**.

🔗 Cache Coherency Protocols

There are numerous different protocols, which vary not only from one hardware manufacturer to another, but also constantly develop within a single vendor's product line. In spite of all this variety, most of the protocols have lots of common aspects. Which is why we will take a closer look at **MESI**. It does not give the reader a full overview of all the protocols out there, however. There are some (e.g. Directory Based (http://courses.cs.washington.edu/courses/cse471/00au/Lectures/luke_directories.pdf)) protocols that are absolutely different. We are not going to look into them.

In MESI, every cache entry can be in one of the following states:

- **Invalid**: the cache does not have such entry
- **Exclusive**: the entry resides in this cache only, and has not been modified

- **Modified:** the processor has modified a value, but has not yet written it back to the main memory or sent to any other processor
- **Shared:** more than one processor has the entry in its cache

Transitions between states occur via sending certain messages that are also a part of the protocol. The exact message types are not quite relevant, so they are omitted in this article. There are many other sources which you can use to gain insight into them. [Memory Barriers: a Hardware View for Software Hackers](http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf) (<http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf>) is the one that I would recommend.

It is ironical that deep down, messaging is used to change states concurrently. Problem, Actor Model haters?

🔗 MESI Optimizations And The Problems They Introduce

Without going into details, we will say that it takes time for messages to be delivered, which introduces more latency into state switching. It is also important to understand that some state transitions require some special handling, which might stall the processor. These things lead to all sorts of scalability and performance problems.

🔗 Store Buffers

If you need to write something to a variable that is **Shared** in the cache, you have to send an **Invalidate** message to all its other holders, and wait for them to acknowledge it. The processor is going to be stalled for that duration. Which is a sad thing, seeing as the time required for that is typically several orders of magnitude higher than executing a simple instruction needs.

In real life, cache entries do not contain just a single variable. The established unit is a cache line, which usually contains more than one variable, and is typically 64 bytes in size.

It can lead to interesting side effects, e.g. [cache contention](http://openjdk.java.net/jeps/142) (<http://openjdk.java.net/jeps/142>).

To avoid such a waste of time, **Store Buffers** are used. The processor places the values which it wants to write to its buffer, and goes on executing things. When all the **Invalidate Acknowledge** messages are received, the data is finally committed.

One can expect a number of hidden dangers here. The easy one is that a processor may try to read some value that it has placed in the store buffer, but which has not yet been committed. The workaround is called **Store Forwarding**, which causes the loads to return the value in the store buffer, if it is present.

The second pitfall is that there is no guarantee on the order in which the stores will leave the buffer. Consider the following piece of code:

```
void executedOnCpu0() {
    value = 10;
    finished = true;
}
```

```
void executedOnCpu1() {
    while(!finished);
    assert value == 10;
}
```

Suppose that when the execution starts, **CPU 0** has `finished` in the **Exclusive** state, while `value` is not installed in its cache at all (i.e. is **Invalid**). In such scenario, `value` will leave the store buffer considerably later than `finished` will. It is entirely possible that **CPU 1** will then load `finished` as `true`, while `value` will not be equal to `10`.

Such changes in the observable behavior are called **reorderings**. Note that it does not necessarily mean that your instructions' places have been changed by some malicious (or well-meaning) party.

It just means that some other CPU has *observed* their results in a different order than what's written in the program.

🔗 Invalidate Queues

Executing an invalidation is not a cheap operation as well, and it costs for the processor applying it. Moreover, it is no surprise that Store Buffers are not infinite, so the processors sometimes have to wait for **Invalidate Acknowledge** to come. These two can make performance degrade considerably. To counter this, **Invalidate Queues** have been introduced. Their contract is as follows:

- For all incoming **Invalidate** requests, **Invalidate Acknowledge** messages are immediately sent
- The **Invalidate** is not in fact applied, but placed to a special queue, to be executed when convenient
- The processor will not send any messages on the cache entry in question, until it processes the **Invalidate**

There, too, are cases when such optimization will lead to counter-intuitive results. We return to our code, and assume that **CPU 1** has `value` in the **Exclusive** state. Here's a diagram of a possible execution:

#	CPU 0: operations	CPU 0: value	CPU 0: finished	CPU 1: operations	CPU 1: value	CPU 1: finished
0		0 (Shared)	false (Exclusive)		0 (Shared)	(Invalid)
1	<div> <code>value = 10;</code> <code>- store_buffer(value)</code> <code>← invalidate(value)</code> </div>	0 (Shared) 10 (in store buffer)	false (Exclusive)			
2				<div> <code>while (!finished);</code> <code>← read(finished)</code> </div>	0 (Shared)	(Invalid)
3	<div> <code>finished = true;</code> </div>	0 (Shared) 10 (in store buffer)	true (Modified)			
4				<div> <code>→ invalidate(value)</code> <code>← invalidate_ack(value)</code> <code>- invalidate_queue(value)</code> </div>	0 (Shared) (in invalidation queue)	(Invalid)
5	<div> <code>→ read(finished)</code> <code>← read_response(finished)</code> </div>	0 (Shared) 10 (in store buffer)	true (Shared)			
6				<div> <code>→ read_response(finished)</code> </div>	0 (Shared) (in invalidation queue)	true (Shared)
7				<div> <code>assert value == 10;</code> </div>	0 (Shared) (in invalidation queue)	true (Shared)
				Assertion fails		
N				<div> <code>- invalidate(value)</code> </div>	(Invalid)	true (Shared)

Concurrency is simple and easy, is it not? The problem is in steps (4) — (6). When **CPU 1** receives an **Invalidate** in (4), it queues it without processing. Then **CPU 1** gets **Read Response** in (6), while the corresponding **Read** has been sent earlier in (2). Despite this, we do not invalidate `value`, ending up with an assertion that fails. If only operation (N) has executed earlier. But alas, the damn optimization has spoiled everything! On the other hand, it grants us some significant performance boost.

The thing is that hardware engineers cannot know in advance when such an optimization is allowed, and when it is not. Which is why they leave the problem in our capable hands. They also give us a little something, with a note attached to it: "It's dangerous to go alone! Take this!"

🔗 Hardware Memory Model

The Magical Sword that software engineers who are setting out to fight Dragons are given, is not quite a sword. Rather, what the hardware guys have given us are the Rules As Written. They describe which values a processor can observe given the instructions this (or some other) processor has executed. What we could classify as Spells would be the Memory Barriers. For the MESI example of ours, they would be the following:

- **Store Memory Barrier** (a.k.a. ST, SMB, `smp_wmb`) is the instruction that tells the processor to apply all the **stores** that are already in the **store buffer**, before it applies any that come after this instruction
- **Load Memory Barrier** (a.k.a. LD, RMB, `smp_rmb`) is the instruction that tells the processor to apply all the **invalidates** that are already in the **invalidate queue**, before executing any loads

So, these two Spells can prevent the two situations which we have come across earlier. We should use it:

```
void executedOnCpu0() {
    value = 10;
    storeMemoryBarrier(); // Mighty Spell!
    finished = true;
}
```

```
void executedOnCpu1() {
    while(!finished);
    loadMemoryBarrier(); // I am a Wizard!
    assert value == 10;
}
```

Yay! We are now safe. Time to write some high-performance *and* correct concurrent code!

Oh, wait. It doesn't even compile, says something about missing methods. What a mess.

🔮 Write Once @ Run Anywhere

All those cache coherency protocols, memory barriers, dropped caches and whatnot seem to be awfully platform-specific things. Java Developers should not care for those at all. Java Memory Model has no notion of reordering, after all.

If you do not fully understand this last phrase, you should not continue reading this article. A better idea would be to go and learn some JMM instead. A good start would be this FAQ (<http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html>).

But there *are* reorderings happening on deeper levels of abstractions. Should be interesting to see how JMM maps to the hardware model. Let's start with a simple class ([github \(https://github.com/gvsmirnov/java-perv/blob/master/labs/src/main/java/ru/gvsmirnov/perv/labs/concurrency/TestSubject.java\)](https://github.com/gvsmirnov/java-perv/blob/master/labs/src/main/java/ru/gvsmirnov/perv/labs/concurrency/TestSubject.java)):

```
1. public class TestSubject {
2.
3.     private volatile boolean finished;
4.     private int value = 0;
5.
6.     void executedOnCpu0() {
7.         value = 10;
8.         finished = true;
9.     }
10.
11.    void executedOnCpu1() {
12.        while(!finished);
13.        assert value == 10;
14.    }
15.
16. }
```

There are many venues we could follow to understand what's going on: the `PrintAssembly` fun, checking out the interpreter's doings, asking someone, mysteriously.

(http://lesswrong.com/lw/iu/mysterious_answers_to_mysterious_questions/) saying that the caches are being dropped, and many more. I have decided to stick with looking at the C1 (a.k.a. the client compiler) of OpenJDK. While the client compiler is barely used in real applications, it is a good choice for educational purposes.

I have used jdk8 at revision 933:4f8fa4724c14 (<http://hg.openjdk.java.net/jdk8/jdk8/file/4f8fa4724c14>) . Things may be different in other versions.

If you have never before dugged through the sources of OpenJDK (and even if you have, for that matter), it could be hard to find where the things that interest you lie. An easy way to narrow down the search space is getting the name of the bytecode instruction that interests you, and simply look for it. Alright, let's do that:

```
$ javac TestSubject.java & javap -c TestSubject
void executedOnCpu0();
  Code:
    0: aload_0          // Push this to the stack
    1: bipush           10 // Push 10 to the stack
    3: putfield         #2 // Assign 10 to the second field(value) of this
    6: aload_0          // Push this to the stack
    7: iconst_1         // Push 1 to the stack
    8: putfield         #3 // Assign 1 to the third field(finished) of this
   11: return

void executedOnCpu1();
  Code:
    0: aload_0          // Push this to the stack
    1: getfield         #3 // Load the third field of this(finished) and push it to the stack
    4: ifne             10 // If the top of the stack is not zero, go to label 10
    7: goto            0 // One more iteration of the loop
   10: getstatic        #4 // Get the static system field $assertionsDisabled:Z
   13: ifne             33 // If the assertions are disabled, go to label 33(the end)
   16: aload_0          // Push this to the stack
   17: getfield         #2 // Load the second field of this(value) and push it to the stack
   20: bipush           10 // Push 10 to the stack
   22: if_icmpeq        33 // If the top two elements of the stack are equal, go to label 33(the end)
   25: new              #5 // Create a new java/lang/AssertionError
   28: dup              // Duplicate the top of the stack
   29: invokespecial    #6 // Invoke the constructor (the <init> method)
   32: athrow           // Throw what we have at the top of the stack (an AssertionError)
   33: return
```

You should not try to predict the performance (or even low-level behavior) of your program by looking at the bytecode. When the JIT Compiler is through with it, there will not be much similarities left.

We are only doing this because we need to know who the assassins were working for.

There are two things of interest here:

1. Assertions are disabled by default, as many people tend to forget. Use `-ea` to enable them.
2. The names that we were looking for: `getfield` and `putfield`.

Ah, the Field brothers. I knew it was them. It is not long before I put them behind bars for good, now.

🔗 Down The Rabbit Hole

As we can see, the instructions used for loading and storing are the same for both `volatile` and plain fields. So, it is a good idea to find where the compiler learns whether a field is `volatile` or not. Digging around a little, we end up in `share/vm/ci/ciField.hpp`. The method of interest is

```
177. bool is_volatile    () { return flags().is_volatile(); }
```

So, what we now are tasked with is finding the methods that handle loading and storing of fields and use investigate all the codepaths conditional on the result of invoking the method above. The Client Compiler processes them on the **Low-Level Intermediate Representation (LIR)** stage, in the file `share/vm/c1/c1_LIRGenerator.cpp`.

🔗 C1 Intermediate Representation

Let's start with the stores. The method that we are looking into is

`void LIRGenerator::do_StoreField(StoreField* x)`, and resides at lines 1658:1751. The first remarkable action that we see is

```

1724. if (is_volatile && os::is_MP()) {
1725.     __ membar_release();
1726. }

```

Cool, a memory barrier! The two underscores are a macro that expand into `gen()->lir()->`, and the invoked method is defined in `share/vm/c1/c1_LIR.hpp`:

```

2089. void membar_release() { append(new LIR_Op0(lir_membar_release)); }

```

So, what happened is that we have appended one more operation, `lir_membar_release`, to our representation.

```

1737. if (is_volatile && !needs_patching) {
1738.     volatile_field_store(value.result(), address, info);
1739. }

```

The invoked method has platform-specific implementations. For x86 (`cpu/x86/vm/c1_LIRGenerator_x86.cpp`), it's fairly simple: for 64-bit fields, we dabble in some Dark Magics to ensure write atomicity. Because the [spec says so](http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.7) (<http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.7>). This is a bit outdated, and may be reviewed in [Java 9](http://openjdk.java.net/jeps/188) (<http://openjdk.java.net/jeps/188>). The last thing that we want to see is one more memory barrier at the very end of the method:

```

1749. if (is_volatile && os::is_MP()) {
1750.     __ membar();
1751. }

```

```

2087. void membar() { append(new LIR_Op0(lir_membar)); }

```

That's it for the stores.

The loads are just a bit lower in the source code, and do not contain anything principally new. They have the same Dark Magic stuff for the atomicity of `long` and `double` fields, and add a `lir_membar_acquire` after the load is done.

Note that I have deliberately left out some of the things that are going on, e.g. the GC-related instructions.

☞ Memory Barrier Types And Abstraction Levels

By this time, you must be wondering what the **release** and **acquire** memory barriers are, for we have not yet introduced them. This is all because the **store** and **load** memory barriers which we have seen before are the operations in the MESI model, while we currently reside a couple of abstraction levels above it (or any other Cache Coherency Protocol). At this level, we have different terminology.

Given that we have two kinds of operations, **Load** and **Store**, we have four ordered of pairs of them: **LoadLoad**, **LoadStore**, **StoreLoad** and **StoreStore**. It is therefore very convenient to have four types of memory barriers with the same names.

If we have a **XY** memory barrier, it means that **all X operations that come before the barrier** must complete their execution before **any Y operation after the barrier** starts.

For instance, **all Store operations before a StoreStore barrier** must complete earlier than **any Store operation that comes after the barrier** starts. The [JSR-133 Cookbook](http://g.oswego.edu/dl/jmm/cookbook.html) (<http://g.oswego.edu/dl/jmm/cookbook.html>) is a good read on the subject.

Some people get confused and think that memory barriers take a variable as an argument, and then prohibit reorderings of the variable stores or loads across threads.

Memory barriers work within one thread only. By combining them in the right way, you can ascertain that when some thread loads the values stored by another thread, it sees a consistent picture. More generally, all the abstractions that JMM goes on about are granted by the correct combination of memory barriers.

Then there are the **Acquire** and **Release** semantics. A **write** operation that has **release** semantics requires that all the memory operations that come before it are finished before the operation itself starts its execution. The opposite is true for the **read-acquire** operations.

One can see that a **Release Memory Barrier** can be implemented as a `LoadStore|StoreStore` combination, and the **Acquire Memory Barrier** is a `LoadStore|LoadLoad`. The `StoreLoad` is what we have seen above as `lir_membar`.

🔗 Emitting Assembly Code

Now that we have sorted out the IR and its memory barriers, we can get down to the native level. All the emission happens in the `share/vm/c1/c1_LIRAssembler.cpp` file:

```
682. case lir_membar_release:
683.     membar_release();
684.     break;
```

The memory barriers are platform-specific, so for x86 we are looking into the `cpu/x86/vm/c1_LIRAssembler_x86.cpp` file. Seeing as x86 is an architecture with a rather strict memory model, most of the memory barriers are no-ops.

```
3987. void LIR_Assembler::membar_acquire() {
3988.     // No x86 machines currently require load fences
3989.     // __ load_fence();
3990. }
3991.
3992. void LIR_Assembler::membar_release() {
3993.     // No x86 machines currently require store fences
3994.     // __ store_fence();
3995. }
```

Not all of them, however:

```
3982. void LIR_Assembler::membar() {
3983.     // QQQ sparc TSO uses this,
3984.     __ membar( Assembler::Membar_mask_bits(Assembler::StoreLoad));
3985. }
```

(which we follow into `cpu/x86/vm/assembler_x86.hpp`)

```
3982. // Serializes memory and blows flags
3983. void membar(Membar_mask_bits order_constraint) {
3984.     if (os::is_MP()) {
3985.         // We only have to handle StoreLoad
3986.         if (order_constraint & StoreLoad) {
3987.             // ALL usable chips support "locked" instructions which suffice
3988.             // as barriers, and are much faster than the alternative of
3989.             // using cpuid instruction. We use here a locked add [esp],0.
3990.             // This is conveniently otherwise a no-op except for blowing
3991.             // flags.
3992.             // Any change to this code may need to revisit other places in
3993.             // the code where this idiom is used, in particular the
3994.             // orderAccess code.
3995.             lock();
3996.             addl(Address(rsp, 0), 0); // Assert the lock# signal here
3997.         }
3998.     }
3999. }
```

So, for every `volatile` write we have to use the relatively expensive `StoreLoad` barrier in the form of `lock addl $0x0, (%rsp)`. It forces us to execute all the pending stores, and effectively ensures that other threads see the fresh values quickly. And for `volatile` read, we emit no additional barriers. One should not think that volatile reads are as cheap as regular reads are (<http://brooker.co.za/blog/2012/09/10/volatile.html>), however.

It should be clear that while the barriers may emit no assembly code, they are still there in the IR. If they were ignored by the components that can modify the code (say, the compiler), there would be bugs like [this one](https://bugs.openjdk.java.net/browse/JDK-7170145) (<https://bugs.openjdk.java.net/browse/JDK-7170145>).

Sanity Checks

While making up theories by looking at the sources of OpenJDK is all nice and good, all the *real* scientists go out there and test their theories. Let us not get too out of the loop and try it as well.

Java Concurrency Stress Fun

The first thing we want to check is that things will actually get bad if we remove `volatile` from our code. The problem with demonstrating such a “reordering” is that the prior probability of it happening is fairly low. And on some architectures, the HMM prohibits it altogether. So, we have to rely on the compiler, and also try it a lot of times.

The good thing is that we have no need to invent the wheel, as there’s the

[jctestress](<http://openjdk.java.net/projects/code-tools/jctestress/>) (<http://openjdk.java.net/projects/code-tools/jctestress/>) tool that executes the code lots of times and keeps an aggregated track of the outcomes. It also very conveniently does all the dirty work for us, including the dirty work we did not even suspect we had to do.

Moreover, jctestress already has the very `test` (<http://hg.openjdk.java.net/code-tools/jctestress/file/dd797d922f1c/tests-custom/src/main/java/org/openjdk/jctestress/tests/fences/UnfencedAcquireReleaseTest.java>) that we need:

```
37. static class State {
38.     int x;
39.     int y; // acq/rel var
40. }
41.
42. @Override
43. public void actor1(State s, IntResult2 r) {
44.     s.x = 1;
45.     s.x = 2;
46.     s.y = 1;
47.     s.x = 3;
48. }
49.
50. @Override
51. public void actor2(State s, IntResult2 r) {
52.     r.r1 = s.y;
53.     r.r2 = s.x;
54. }
```

We have one thread executing stores, and another thread doing reads, and then reporting the observed states. The framework aggregates the results for us, and then matches it against some certain rules (<http://hg.openjdk.java.net/code-tools/jctestress/file/ecfe9e112bc6/tests-custom/src/main/resources/org/openjdk/jctestress/desc/fences.xml>). We are interested in two possible observations made by the second thread: `[1, 0]` and `[1, 1]`. In these two cases, it has loaded `y == 1`, but has either failed to see any writes to `x`, or the loaded version was not the most recent one at the time `y` was written. According to our theory, such events should happen without the `volatile` modifier. Let’s see:

```
$ java -jar tests-all/target/jctestress.jar -v -t ".*UnfencedAcquireReleaseTest.*"
...
```

Observed state	Occurrence	Expectation	Interpretation
[0, 0]	32725135	ACCEPTABLE	Before observing releasing write to, any value is OK for \$x.
[0, 1]	15	ACCEPTABLE	Before observing releasing write to, any value is OK for \$x.
[0, 2]	36	ACCEPTABLE	Before observing releasing write to, any value is OK for \$x.
[0, 3]	10902	ACCEPTABLE	Before observing releasing write to, any value is OK for \$x.
[1, 0]	65960	ACCEPTABLE_INTERESTING	Can read the default or old value for \$x after \$y is observed.
[1, 3]	50929785	ACCEPTABLE	Can see a released value of \$x if \$y is observed.
[1, 2]	7	ACCEPTABLE	Can see a released value of \$x if \$y is observed.

So, in 65960 cases out 83731840 ($\approx 0.07\%$) the second thread has observed `y == 1 && x == 0`, which confirms that the reorderings can indeed happen.

PrintAssembly Fun

The second thing we want to check is if we have correctly predicted the generated assembly code. So, we add lots of invocations of the required code, disable inlining for easier result interpretation, enable assertions and run in the client VM:


```
$ java -client -ea -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly -XX:MaxInlineSize=0 TestSubject
...
# {method} 'executedOnCpu0' '()'V in 'TestSubject'
...
0x00007f6d1d07405c: movl    $0xa,0xc(%rsi)
0x00007f6d1d074063: movb    $0x1,0x10(%rsi)
0x00007f6d1d074067: lock    addl $0x0,(%rsp)    ;*putfield finished
                                ; - TestSubject::executedOnCpu0@8 (line 15)
...
# {method} 'executedOnCpu1' '()'V in 'TestSubject'
...
0x00007f6d1d061126: movzbl 0x10(%rbx),%r11d    ;*getfield finished
                                ; - TestSubject::executedOnCpu1@1 (line 19)
0x00007f6d1d06112b: test    %r11d,%r11d
...
```

Yay, just as planned! Means that it's time to go wrap up.

Let me remind you the questions that you should be able to answer by now:

- How does that happens-before thing work?
- Does using `volatile` result in caches being dropped?
- Why do we even need a memory model in the first place?

Do you think you can answer these? Welcome to the comments in any case!

P.S. This is the translation of my earlier blog entry in Russian (<http://habrahabr.ru/post/209128/>), which has been reviewed by Aleksey Shipilëv (<https://twitter.com/shipilev>), so thank him.

17 Comments **Gleb Smirnov: Homesite**

 Login ▾

 Recommend 1  Tweet  Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



Joachim Dorn • 6 years ago

Nice introduction, however imho you stress the processor side (HMM) too much. Its also/more often the compilers that are aggressively reordering (so even x86's TMO doesnt help for reasoning).

From JSR-133 (new JMM) FAQ: "There are a number of cases in which accesses to program variables (object instance fields, class static fields, and array elements) may appear to execute in a different order than was specified by the program. The compiler is free to take liberties with the ordering of instructions in the name of optimization."

see:

- <http://www.cs.umd.edu/~pugh...>
- <http://www.infoq.com/presen...>
- <http://channel9.msdn.com/Sh...>
- (academic roots of 'happened before' <http://www.ics.uci.edu/~cs2...>)

3 ^ | ▾ • Reply • Share ▸



gysmirnov Mod → Joachim Dorn • 6 years ago

There was a warning that the compilers can also cause counter-intuitive behavior of one's code, unless explicitly prohibited from doing so:

- > It should be clear that while the barriers may emit no assembly code, they are still there in the IR.
- > If they were ignored by the components that can modify the code (say, the compiler), there would be ...

Moreover, in the jcstress example, it was indeed the compiler that has introduced the reordering.

However, I can't quite agree that the compilers are more often the culprit than the hardware is. While that may be true on x86, other architectures with more relaxed memory models can break way more things than the compilers do.

Also, the whole point of this article was to show how JMM abstractions mapped to the hardware.

^ | ▾ • Reply • Share ▸



Joachim Dorn → gysmirnov • 6 years ago

Agreed. The main reason for commenting was my feeling, that for readers new to the topic its easy to read over the points about the compiler's role in that game.

^ | ▾ • Reply • Share ▸

^ | v • Reply • Share ›



gvsimirnov Mod → Joachim Dorn • 6 years ago

Got it. Thanks for pointing it out. I do hope the people will also read the comments :)

^ | v • Reply • Share ›



Alexander Oksenenko • 6 years ago • edited

I know nothing about jmm and your article is awesome for someone like me :P
sorry for useless comment

3 ^ | v • Reply • Share ›



gvsimirnov Mod → Alexander Oksenenko • 6 years ago

Thanks. At least I know that these comments work! :)

^ | v • Reply • Share ›



Apurva Singh • 4 years ago

StoreLoad barrier means stores cannot be reordered with loads. Why is this needed for volatile writes? Store barrier is enough. Flush store buffers and make the volatile write visible?

1 ^ | v • Reply • Share ›



gvsimirnov Mod → Apurva Singh • 4 years ago • edited

Very glad someone asked that, thank you! An important reason for this is that volatile stores and loads are a part of the Synchronization Order (see <https://docs.oracle.com/jav...>), and it's a total order. Valid executions have to be consistent with program order, too, so we can't have a volatile store reordered with a volatile load. It would also be correct to emit this barrier before a volatile load, but volatile loads are much more frequent than stores, so it would hinder performance.

1 ^ | v • Reply • Share ›



Joachim Dorn • 6 years ago

By the way - I just learned about JITwatch (<https://github.com/AdoptOpenJDK/jitwatch>) by a post from Nitsan Wakart (<http://psy-lob-saw.blogspot...>). This tool looks great for helping your "PrintAssembly Fun" section

1 ^ | v • Reply • Share ›



gvsimirnov Mod → Joachim Dorn • 6 years ago

Right, I think I'll try it some time soon. Thanks!

^ | v • Reply • Share ›



Apurva Singh • 4 years ago

super thanks smirnov

^ | v • Reply • Share ›



Ami • 6 years ago

extremely helpful for novice programmer like me...eagerly waiting for second part of this article.

^ | v • Reply • Share ›



berkus • 6 years ago • edited

Yay, well written explanation. Store Buffers and Invalidate Queues are what was missing from my mental model of this whole thing. Thanks for clearing it up!

^ | v • Reply • Share ›



gvsimirnov Mod → berkus • 6 years ago

You're welcome. Just in case, though, let me remind you that there are numerous other hw/sw optimizations at play. Store Buffers and Invalidate Queues are just an example. Some architectures might not even have those.

^ | v • Reply • Share ›



berkus → gvsimirnov • 6 years ago

It's just some parts of the puzzle weren't clicking until I looked at it as store buffers/invalidate queues.

^ | v • Reply • Share ›



Guest • 6 years ago

Funny how the only comment is a useless one

^ | v • Reply • Share ›



Guest → Guest • 6 years ago

Now you have two comments that are useless. Or three after I press the button.

3 ^ | v • Reply • Share ›

ALSO ON GLEB SMIRNOV: HOMESITE

Building OpenJDK 8 on Mac OS X Mavericks

18 comments • 6 years ago



qu1j0t3 — Turns out the CGBase.h file is only present in the Xcode SDKs, not in /System/Library/Frameworks (!). Got past this one by changing -F to point into /Applications/Xcode.app/Contents/Develop..., in

I Am Now A Part Of Plumb


1 comment • 5 years ago



Ivan — >> I am very happy to finally leave Russia ... I am not exactly a russophobe, for I equally dislike idiots of any nationality, but ... it seems to me that there are much


Today I Learned: Volume 1

4 comments • 5 years ago

 gvmirnov — Right, it probably doesn't. However, other things like complex exhibited behavior, do. I just mentioned the axon size as a curious fact, not as an indication of intelligence. Thanks for the references.

Investigating "Mysterious" Hotspot Behavior

2 comments • 5 years ago

 gvmirnov — Definitely there is a lot of rationale behind this. First of all, if an application gets stuck in a metaspace expansion triggered GC loop, it is not the problem of metaspace, but of the application. With permgen, the



(<http://creativecommons.org/publicdomain/zero/1.0/>) Freely use anything from this website in any way you can imagine