Mechanical Sympathy

Hardware and software working together in harmony

Saturday, 30 July 2011

False Sharing

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 32-256 in size. The most common cache line size is 64 bytes. False sharing is a term which applies when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line. Write contention on cache lines is the single most limiting factor on achieving scalability for parallel threads of execution in an SMP system. I've heard false sharing described as the silent performance killer because it is far from obvious when looking at code.

To achieve linear scalability with number of threads, we must ensure no two threads write to the same variable or cache line. Two threads writing to the same variable can be tracked down at a code level. To be able to know if independent variables share the same cache line we need to know the memory layout, or we can get a tool to tell us. Intel VTune is such a profiling tool. In this article I'll explain how memory is laid out for Java objects and how we can pad out our cache lines to avoid false sharing.

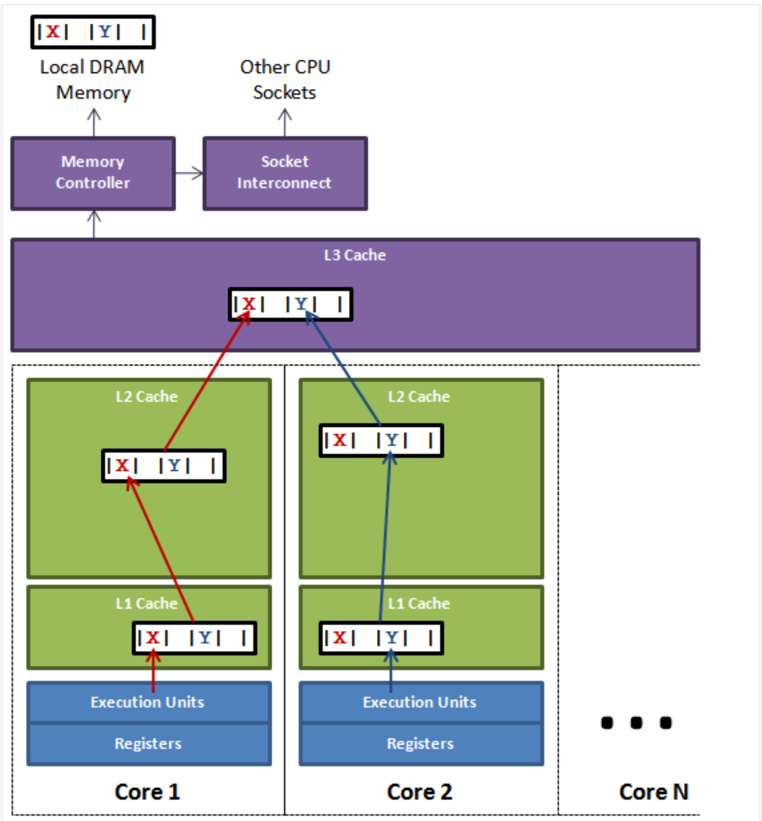


Figure 1.

Figure 1. above illustrates the issue of false sharing. A thread running on core 1 wants to update variable X while a thread on core 2 wants to update variable Y. Unfortunately these two hot variables reside in the same cache line. Each thread will race for ownership of the cache line so they can update it. If core 1 gets ownership then the cache sub-system will need to invalidate the corresponding cache line for core 2. When Core 2 gets ownership and performs its update, then core 1 will be told to invalidate its copy of the cache line. This will ping pong back and forth via the L3 cache greatly impacting performance. The issue would be further exacerbated if competing cores are on different sockets and additionally have to cross the socket interconnect.

Java Memory Layout

For the Hotspot JVM, all objects have a 2-word header. First is the "mark" word which is made up of 24-bits for the hash code and 8-bits for flags such as the lock state, or it can be swapped for lock objects. The second is a reference to the class of the object. Arrays have an additional word for the size of the array. Every object is aligned to an 8-byte granularity boundary for performance. Therefore to be efficient when packing, the object fields are re-ordered from declaration order to the following order based on size in bytes:

1. doubles (8) and longs (8)
2. ints (4) and floats (4)
3. shorts (2) and chars (2)

Search This Blog

Search

Discussion Group

https://groups.google.com/forum/#!forum/mechanical-sympathy

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

Training & Consulting

www.real-logic.co.uk

Next public course

Conferences & Workshops

- QCon London March
- JavaLand Brühl - March
- Craft Conf Budapest - April
- JOnTheBeach Málaga - May
- Goto Amsterdam - June
- JCrete - July

Popular Posts

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 ...

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...

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...

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

4. booleans (1) and bytes (1)
5. references (4/8)
6. <repeat for sub-class fields>

With this knowledge we can pad a cache line between any fields with 7 longs. Within the Disruptor we pad cache lines around the RingBuffer cursor and BatchEventProcessor sequences.

To show the performance impact let's take a few threads each updating their own independent counters. These counters will be volatile longs so the world can see their progress.

```
public final class FalseSharing
    implements Runnable
{
    public final static int NUM_THREADS = 4; // change
    public final static long ITERATIONS = 500L * 1000L * 1000L;
    private final int arrayIndex;

    private static VolatileLong[] longs = new VolatileLong[NUM_THREADS];
    static
    {
        for (int i = 0; i < longs.length; i++)
        {
            longs[i] = new VolatileLong();
        }
    }

    public FalseSharing(final int arrayIndex)
    {
        this.arrayIndex = arrayIndex;
    }

    public static void main(final String[] args) throws Exception
    {
        final long start = System.nanoTime();
        runTest();
        System.out.println("duration = " + (System.nanoTime() - start));
    }

    private static void runTest() throws InterruptedException
    {
        Thread[] threads = new Thread[NUM_THREADS];

        for (int i = 0; i < threads.length; i++)
        {
            threads[i] = new Thread(new FalseSharing(i));
        }

        for (Thread t : threads)
        {
            t.start();
        }

        for (Thread t : threads)
        {
            t.join();
        }
    }

    public void run()
    {
        long i = ITERATIONS + 1;
        while (0 != --i)
        {
            longs[arrayIndex].value = i;
        }
    }

    public final static class VolatileLong
    {
        public volatile long value = 0L;
        public long p1, p2, p3, p4, p5, p6; // comment out
    }
}
```

Results

Running the above code while ramping the number of threads and adding/removing the cache line padding, I get the results depicted in Figure 2. below. This is measuring the duration of test runs on my 4-core Nehalem.

a native language like C++? If you use s...

Simple Binary Encoding

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

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...

Memory Barriers/Fences

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

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...

False Sharing

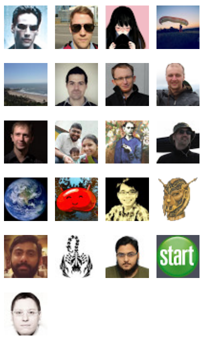
Memory is stored within the cache system in units know as cache lines. Cache lines are a power of 2 of contiguous bytes which are typically...

Blog Archive

- ▶ 2014 (1)
- ▶ 2013 (5)
- ▶ 2012 (7)
- ▼ 2011 (19)
  - ▶ December (1)
  - ▶ November (3)
  - ▶ October (1)
  - ▶ September (3)
  - ▶ August (4)
  - ▼ July (7)
    - False Sharing
    - Memory Barriers/Fences
    - Processor Affinity - Part 1
    - LMAX Architecture - by Martin Fowler
    - Let The Caller Choose
    - Write Combining
    - Why Mechanical Sympathy?

Followers

Followers (711) Next



Follow

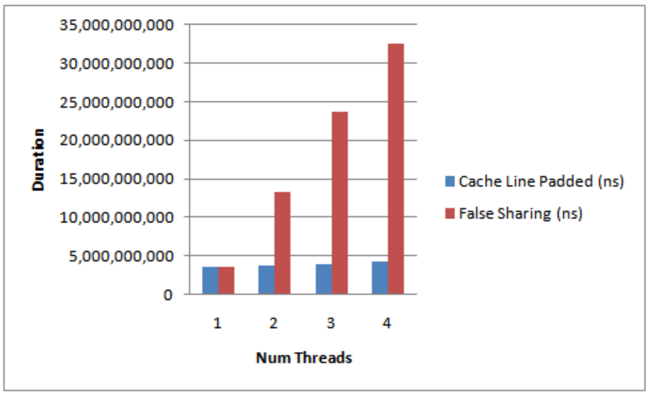


Figure 2.

The impact of false sharing can clearly be seen by the increased execution time required to complete the test. Without the cache line contention we achieve near linear scale up with threads.

This is not a perfect test because we cannot be sure where the VolatileLongs will be laid out in memory. They are independent objects. However experience shows that objects allocated at the same time tend to be co-located.

So there you have it. False sharing can be a silent performance killer.

Note: Please read my further adventures with false sharing in this follow on blog.

Posted by Martin Thompson at 09:34

Labels: False Sharing, Java, Performance, Threads

Location: London, UK

34 comments:

mbien 30 July 2011 at 20:10

nice. I didn't thought that it would be reproducible in a microbenchmark. Sounds like a good opportunity for an jvm optimisation in C2. The main issue would be to detect this situation *before* the method runs hot. (but even if this wouldn't be feasible the GC could still "fix" it in the next full GC)

Reply

Martin Thompson 31 July 2011 at 11:11

Thanks. Writing micro benchmarks are very easy to get wrong. It would be nice if Java could provide a hint to cache align an object. Kind of similar to the support in C# and C++0x, e.g. [[align(CACHE_LINE_SIZE)]] T then add padding. It is usually clear what variables need padded because they are the concurrent state for signalling between threads. I plan to write a blog showing what can happen with more subtle false sharing that does not result from planned concurrent state. It is a shame the JVM runtime will not give profiling information for what lines of code are causing L2 cache issues.

Reply

David Yu 31 July 2011 at 12:53

So I guess we're left with micro-benchmarks to figure out cache alignment. Didn't know C# has cache alignment. Would really be nice if its available on java :/

Anyway, great read! Thanks for sharing.

Reply

Olivier Deheurles 31 July 2011 at 14:46

In C# we can define the memory layout only with structs, not with classes.

Trying to add padding in a class does not work: the .NET JIT reorder fields very often so you never know what will be the order of fields in memory once Jitted (I checked with Son Of Strike. http://msdn.microsoft.com/en-us/library/bb190764.aspx)

There is something else which is impossible to do in .NET and Java: controling dynamically the size of the padding which means the code is not portable from one architecture to another. Maybe using weaving but this sound nasty...

If you want more info about false sharing I would strongly suggest to have a look here:
- Eliminating false sharing (Herb Sutter): http://drdobbs.com/go-parallel/article/showArticle.jhtml?articleID=217500206
- Scott Meyers video here:http://skillsmatter.com/podcast/home/cpu-caches-and-why-you-care and slides here:http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf
- Ulrich Drepper describes the problem in his white paper page 65: http://www.inf.udec.cl/~leo/cpumemory.pdf

Olivier

Reply

Replies

Wil 28 February 2012 at 18:34

Olivier, I believe you are mistaken about the layout. You can provide layout information for both classes and structs. The difference is that depending on the specified option and the type of fields/properties in the class/struct, layout may be different in the managed vs unmanaged environment.

You can also dynamically apply padding BTW, albeit in a roundabout way. I do it by creating the structure (all blittable tyes, Layout.Sequential or Layout.Explicit), acquiring native memory myself (Marshal.AllocHGlobal), using a pointer with array subscripts. This allows controlling the layout as well as the padding.

Wil 28 February 2012 at 18:45

Should be "blittable types".

Also, array subscripts if the padding is on the class/struct, otherwise, pointers since the padding is dynamically "added" by the memory allocation.

Reply

Matt Fowles 10 August 2011 at 00:26

Objects in the JVM have a header that includes things like the lock info for the object and magic bits used for GC bookkeeping. I suspect you do not need all 7 longs to make your object a cache line long.

Reply

Martin Thompson 10 August 2011 at 07:08

Matt,

I've considered that question many times myself :-) You are right in that Hotspot uses 2 words for the object header. The second is the class pointer and therefore pretty dormant. The first word, the "mark" word, is not so dormant. It can contain the hashcode plus some flags. It can also be swapped for the object lock. Mostly the flags are used for GC housekeeping. To be safe I've chosen to not use it as I don't know the frequency with which it will be updated by the Runtime with any certainty. Other JVMs such as JRockit also play the first two words of an object header.

It is probably safe to assume it as effective padding but you should measure for your own application, and JVM combination, and let the results inform you.

Martin...

Reply

Matt Fowles 10 August 2011 at 19:02

Looking at Sequence[1] in google code, you already do share a cache line with the object header (well most of the time, I suppose the object could end up perfectly align so that the header is in one line and the rest is in another). It seems like you should either add 7 longs of padding before the volatile or trim out 1 or 2 longs (64 vs 128 bit headers on 32 vs 64 bit VMs). I agree that the results should be the ultimate arbiter (and concede that this is probably needless micro-optimization).

[1] http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/Sequence.java

Reply

Martin Thompson 11 August 2011 at 07:42

Matt,

From my tests I can trim one long and false sharing does not occur. Trim 2 and some occurs.

Measurement is the key :-)

Also one should never synchronize on the Sequence object to prevent its own mark word being used.

Martin...

Reply

Guðmundur Georgsson 12 August 2011 at 17:08

Matt

Have you tested the padding with jre 1.7? I have, and the results indicate that the padding is not working as expected, i.e. no near linear scale up with threads. (And yes I see the expected scale up when switching back to jre 1.6).

Gum

Reply

Martin Thompson 13 August 2011 at 09:19

Gum,

I had not tested this on Java 7. I just ran a test and get the same slow down so false sharing might be occurring. I'll investigate. I'm also seeing similar slow down in the Disruptor performance tests.

Thanks,
Martin...

Reply

Martin Thompson 13 August 2011 at 10:16

I've posted a new article illustrating a more robust technique that works for Java 7.

http://mechanical-sympathy.blogspot.com/2011/08/false-sharing-java-7.html

Reply

Ming Fang 23 August 2011 at 17:02

How would we pad two int fields? For example,
class Foo{
int a;
long p1,p2,p3,p4,p5
int b;
}
With the re-ording rules, all the pads will be moved to the front leaving a next to b.

Reply

Martin Thompson 23 August 2011 at 17:33

Try using an AtomicIntegerArray for each of "a" and "b" like in the pattern described from my follow on blog post.

http://mechanical-sympathy.blogspot.com/2011/08/false-sharing-java-7.html

You need to make the size of the array larger to reflect the 4-byte integer values to pad out for 64 bytes.

Reply

Martin Thompson 24 August 2011 at 07:44

BTW why are you trying to pad "a" and "b" when they are not volatile? Are they shared across threads?

Reply

sergey 31 August 2011 at 18:46

Thanks for such a great article!

On my 64-bit i7 padding works with 6 longs in total. It also work with 5 longs and one int or byte, because of the object alignment. But if I add one more additional byte, false sharing takes place:

public final static class VolatileLong {
public volatile long value = 0L;
public long p1, p2, p3, p4;
byte b1, b2;
}

But I would not expect that. According to my estimation sizeof(VolatileLong) = 2*8 + 5*8 +2*1 = 58. Maybe you have an idea what is wrong with my estimation...

Thanks!
Siarhei

Reply

Martin Thompson 31 August 2011 at 18:51

Sergey,

What JVM and OS are you using?

Reply

sergey 31 August 2011 at 19:17

Sorry... I am using Hotspot JVM 1.6.0_26 and Ubuntu 10.04

Reply

Martin Thompson 31 August 2011 at 19:17

Sergey,

Seems that the JVM will optimise out the value when you add more than a size unless they are used. Try the following on 64-bit 1.6.0_27 on Windows 7.

public final static class VolatileLong
{
public volatile long value = 0L;
public long p1, p2, p3, p4, p5, p6 = 3L;
}

Try setting the constant to 0, 1, 2, 3, etc. and see what happens.

This is why I recommend the new technique in my follow on article.

http://mechanical-sympathy.blogspot.com/2011/08/false-sharing-java-7.html

Reply

Martin Thompson 31 August 2011 at 21:20

After a good discussion with Jeff Hain we have discovered the JVM cannot be trusted when it comes to re-ordering code. What happens before the volatile seems to be safe but what happens after is not reliable. We have a better solution by replacing the VolatileLong with the following and using the normal methods of AtomicLong.

```
static class PaddedAtomicLong extends AtomicLong
{
public volatile long p1, p2, p3, p4, p5, p6, p7 = 7L;
}
```

Oh how I wish the Java powers the be would get serious about this stuff and add some intrinsics that are cache line aligned and padded. This have been the biggest source or performance bugs in the Disruptor.

Reply

Javin Paul 5 September 2011 at 15:17

fantastic article man, you have indeed covered the topic quite well with code and graphics. very low level information.

Thanks
How HashMap works in Java

Reply

krosh 10 May 2012 at 17:05

This is off-topic but I noticed a curious thing running the test under 64-bit v1.6.0_31.

```
runTest();
System.out.println("duration = " + (System.nanoTime() - start));
```

prints:

```
duration = 5618002000
```

but

```
runTest();
System.out.println("duration = " + (System.nanoTime() - start) / 1000000);
```

prints:

```
duration = 22418
```

Any ideas why it supposedly takes four times longer?

I do not think it is because of

```
27: ldc2_w #10; //long 1000000l
30: ldiv
```

Besides, it executes after

```
22: invokestatic #3; //Method java/lang/System.nanoTime:()J
```

By the way,

```
runTest();
System.out.println((System.nanoTime() - start));
```

prints:

```
22824718000
```

Reply

Replies

Martin Thompson 16 May 2012 at 12:40

I cannot reproduce this finding. The results I get are:

```
System.out.println("duration = " + (System.nanoTime() - start));
duration = 4873935430

System.out.println("duration = " + (System.nanoTime() - start) / 1000000);
duration = 4722
```

Reply

Alex Koturanov 18 September 2013 at 03:45

Martin, thank you for the great article. I would also recommend to use free CPUID PerfMonitor utility to monitor L2/L3 cache hits: http://www.cpuid.com/downloads/perfmonitor2/2.02_x64.zip (Intel VTune is an overkill for simple scenarios).

Reply

netrikare 13 January 2016 at 14:50

Hello Martin,

how does Hyper Threading affect results? In c you can set thread affinity to a specific core (or logical processor). Do you have any benchmarks of 2 threads executing on the same core but on different logical processors vs same core, 1 logical processor?

Reply

Replies

Martin Thompson 6 March 2016 at 13:39

If two OS threads map to hyper threads sharing the same core then there is no false sharing as the cache lines are in the L1. Try benchmarking and setting the thread affinity to find out.

Reply

DeepChahal 17 January 2017 at 14:40

public long p1, p2, p3, p4, p5, p6;

should not be above padding having p7 also?

Reply

DeepChahal 17 January 2017 at 14:40

public long p1, p2, p3, p4, p5, p6; , I think we should have 7 long padding variable

Reply

Replies

Martin Thompson 17 January 2017 at 15:11

Have you considered the object header and the value itself?

Reply

leo lin 26 February 2017 at 18:55

hi,Martin , i doubt that ,what if following situation happened

| * | * | * | * | OH | OH | P | v | ------ line 1
| P | P | P | P | P | P | P | * | ------ line 2

that is what the 2 cache line looks like.i assume that first 4 bytes of " line 1 "has a volatile value too.
is that cause False Sharing?

Reply

Replies

Martin Thompson 26 February 2017 at 19:06

The example is just for illustration. It is properly addressed using a technique like the following:
https://github.com/real-logic/Agrona/blob/master/agrona/src/main/java/org/agrona/concurrent/AbstractConcurrentArrayQueue.java#L38

leo lin 27 February 2017 at 02:36

sorry,i made a mistake,it is the first 4 slots(8bytes per slot).....not first 4 bytes

leo lin 27 February 2017 at 03:07

The example you offered, would cost lot of memory.....
but i understand what you mean ,i can pad head and tail to avoid the situation i presented right? thanks a lot martin

Reply

Enter your comment...

Comment as: fangsboyfriend(Sign out

Publish Preview Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Simple theme. Powered by Blogger.