Most developers who deal with web development don’t think how their programs behave on the low-level with hardware specific stuff. Today we will take a look. Half a year ago, I surfed the insides of JDK and found some curious things. I will discuss them in more detail further on (as you guessed, it’s related to the subject of this article).

Code which Java developers produce lives in a multiprocessor environment. CPUs use cache for reducing the cost to access data from the main memory (aka L2/L3). This well-known schema is illustrated in the picture below:

The single cache line has a size of 64 bytes on Intel’s architecture (on different platforms it ranges from 32 bytes to 128), x86–64 machines invalidate this chunk of memory atomically. Different cores can write data from the main memory to the same cache. There are no problems with cache coherency during concurrent reading, but writing is a more sophisticated operation - it makes a high load on coherency protocol. **False sharing** is when the memory cells are different but physically fall on the same cache line. False sharing can be a source of descending performance (to make false sharing reproducible, threads must resist on different physic CPUs, but in a high load apps it’s not a problem — OS planners gather threads on them). Suddenly, the simple writing operation to the local variable can be costly.

The first trick that comes to mind is displacing the remaining memory in the cache line. For the situation where a single long variable is actively in use (variable usefulVal in the code block below), let's create 7 long variables (t1, t2... t7) and fill the remaining 56 bytes of the cache by making artificial padding. It looks like this:

public class SomePopularObject {

public volatile long usefulVal;

public volatile long t1, t2, t3, t4, t5, t6, t7;

}

You can tell that allocation of an excessive 56 bytes is an awful idea, but the benchmarks listed below refute this. However, this code will hold some surprises for the programmer during code review (especially if he doesn’t know this trick) and utils for code analysis (find bugs, sonar, etc.) will behave wildly.

This approach applied in the older versions of Lmax Disruptor (high-performance library for concurrency) is the [PaddedAtomicLong](https://github.com/jbrisbin/disruptor/blob/master/src/main/java/com/lmax/disruptor/util/PaddedAtomicLong.java" \l "L25" \t "_blank) class (this class was removed in the latest versions).

Developers familiar with this topic will know that JVM prevents usage of dead code (compilers have methods for analyzing code on the usage of variables like [Reaching definition](https://en.wikipedia.org/wiki/Reaching_definition)). As a consequence, variables t1, t2 … t7 will be eliminated by the compiler thanks to [dead code elimination](https://en.wikipedia.org/wiki/Dead_code_elimination) (enforced with changes to javac in Java 7). But we can cheat — let’s make an “artificial” usage of these variables:

public class SomePopularObject {

public volatile long usefulVal;

public volatile long t1, t2, t3, t4, t5, t6, t7 = 1L;

public long preventOptmisation(){

return t1 + t2 + t3 + t4 + t5 + t6 + t7;

}

}

Apparently, this technique is naive. Java 8 introduced a different way of preventing false sharing with the ***@Contended*** annotation in [JEP 142](http://openjdk.java.net/jeps/142), which allows you to get rid of declaring dead variables. Under the hood it works with the following strategy — JVM allocates 128 bytes of space after each annotated field. The attentive reader will ask, 'why does JVM allocate 128 bytes while our cache line is only 64?' And the answer is …..[instruction prefetcher](https://en.wikipedia.org/wiki/Instruction_prefetch)*.* According to Wikipedia, "Prefetching occurs when a processor requests an instruction or data block from main memory before it is actually needed." Prefetching can occupy two cache lines simultaneously, as a result of this we must redouble the padding size.

Let’s check how ***@Contended***annotation affects an object’s layout in the memory. To check this influence, I used a [JOL](http://openjdk.java.net/projects/code-tools/jol/)(Java Object Layout tool) to observe the structure of the SomePopularObject class:

public class SomePopularObject {

@Contended

public volatile long usefulVal;

public volatile long anotherVal;

}

The result of layout of the mentioned class:

// -XX:-RestrictContended

SomePopularObject object internals:

OFFSET SIZE TYPE DESCRIPTION VALUE

0 12 (object header) N/A

12 4 Main SomePopularObject.this$0 N/A

16 8 long SomePopularObject.anotherVal N/A

24 128 (alignment/padding gap)

152 8 long SomePopularObject.usefulVal N/A

Instance size: 160 bytes

Space losses: 128 bytes internal + 0 bytes external = 128 bytes total

// -XX:-RestrictContended -XX:ContendedPaddingWidth=256

SomePopularObject object internals:

OFFSET SIZE TYPE DESCRIPTION VALUE

0 12 (object header) N/A

12 4 Main SomePopularObject.this$0 N/A

16 8 long SomePopularObject.anotherVal N/A

24 256 (alignment/padding gap)

280 8 long SomePopularObject.usefulVal N/A

Instance size: 288 bytes

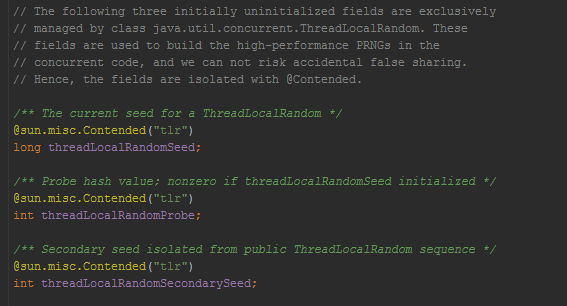
Space losses: 256 bytes internal + 0 bytes external = 256 bytes total

-XX:-RestrictContended  should be used in order to set the result of *@Contended*annotation. Another flag -XX:ContendedPaddingWidth allows you to control the padding size — in the first case, the default option is 128, while in the second it's 256. As a result, there is a difference in 128 bytes.

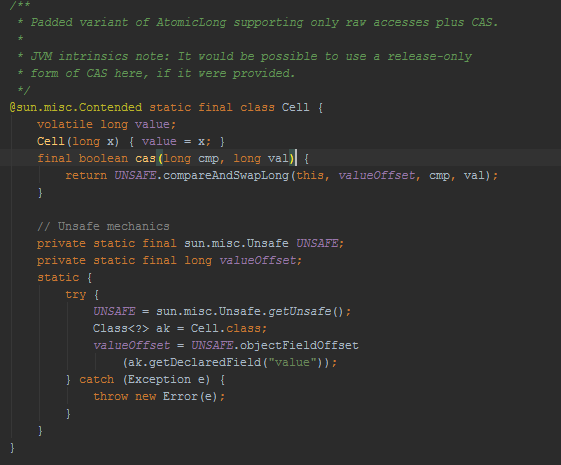
In the beginning of the article, I mentioned the relationship of web development with hardware - it’s time to explain what I had in mind. Most of the JVM-based web frameworks use the thread pool for an invocation specific job. One of them is **ForkJoinPool**(which appeared in Java 7), and it is a widely used tool inside the Java Web framework. You can ForkJoinPool in different places, like the Play Framework (pure web framework), Akka, Scala’s implicit executor service, concurrent implementations of fork-join algorithms (quick sort, algorithms on graphs), Clojure, X10, etc.

The efficiency of ForkJoinPool can be explained by taking a look at the processes under the hood. A key concept is the [**Work stealing**](https://en.wikipedia.org/wiki/Work_stealing)**concurrent pattern,** which is a base of ForkJoinPool. This pattern of concurrent programming brings efficiency to execution time, memory usage, and inter-processor communication. Jobs for “stealing” and further execution are stored in the **WorkQueue class**(nested class in ForkJoinPool). As a result, WorkQueue (and ForkJoinPool) is a potential candidate for false sharing. The solution of this issue was pretty simple — it’s the usage of ***@Contended***annotation under them.

Image titleThe use of this annotation is not limited only to ForkJoinPool. This annotation is used in other places in JDK, for instance, high-performance PRNG managed by **java.util.concurrent.ThreadLocalRandom.**To get rid of accidental false sharing, three seed fields are marked with the ***@Contended***annotation (see picture below). Note that seed fields are located in the java.lang.Thread class, not in ThreadLocalRandom.



In Java 8, we saw changes in atomics — a padded variant of AtomicLong — LongAdder. All of them delegate to a **Cell class**(which was introduced in JSR-166e) — see the picture the below. A cell class is nested in **java.util.concurrent.atomic.Striped64**. LongAdder aims to improve the performance of operations like **CAS**and **lazySet**due to the fact that threads can perform operations on dedicated cells instead of a single annotation susceptible to false sharing. The new atomic API, as most current operations, collaborates with **sun.misc.Unsafe**.



Finally, I would like to provide some evidence for the superiority of padded objects over unpadded objects. I used JMH for this:

import org.openjdk.jmh.annotations.\*;

import sun.misc.Contended;

@Fork(value = 1, jvmArgsPrepend = "-XX:-RestrictContended")

@Warmup(iterations = 10)

@Measurement(iterations = 30)

@Threads(2)

public class ContendedBenchmarks {

@State(Scope.Group)

public static class Unpadded {

public long a;

public long b;

}

@State(Scope.Group)

public static class Padded {

@Contended

public long a;

public long b;

}

@Group("unpadded")

@GroupThreads(1)

@Benchmark

public long updateUnpaddedA(Unpadded u) {

return u.a++;

}

@Group("unpadded")

@GroupThreads(1)

@Benchmark

public long updateUnpaddedB(Unpadded u) {

return u.b++;

}

@Group("padded")

@GroupThreads(1)

@Benchmark

public long updatePaddedA(Padded u) {

return u.a++;

}

@Group("padded")

@GroupThreads(1)

@Benchmark

public long updatePaddedB(Padded u) {

return u.b++;

}

}

The result of this benchmark proves that JVM effectively solves the false sharing problem with the ***@Contended***annotation.

Full results of benchmarks:

Benchmark Mode Cnt Score Error Units

ContendedBenchmarks.padded thrpt 30 236434399.252 ± 9209297.004 ops/s

ContendedBenchmarks.padded:updatePaddedA thrpt 30 114409851.688 ± 4803042.336 ops/s

ContendedBenchmarks.padded:updatePaddedB thrpt 30 122024547.564 ± 5108447.473 ops/s

ContendedBenchmarks.unpadded thrpt 30 63941093.638 ± 2478065.296 ops/s

ContendedBenchmarks.unpadded:updateUnpaddedA thrpt 30 32050855.915 ± 1371581.512 ops/s

ContendedBenchmarks.unpadded:updateUnpaddedB thrpt 30 31890237.723 ± 1344691.282 ops/s

In symmetric multiprocessor (SMP) systems, each processor has a local cache. The memory system must guarantee cache coherence. False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces an update, which hurts performance. This article covers methods to detect and correct false sharing.

This article is part of the larger series, "[Intel Guide for Developing Multithreaded Applications](https://software.intel.com/en-us/articles/intel-guide-for-developing-multithreaded-applications)," which provides guidelines for developing efficient multithreaded applications for Intel® platforms.

Background

False sharing is a well-known performance issue on SMP systems, where each processor has a local cache. It occurs when threads on different processors modify variables that reside on the same cache line, as illustrated in Figure 1. This circumstance is called false sharing because each thread is not actually sharing access to the same variable. Access to the same variable, or true sharing, would require programmatic synchronization constructs to ensure ordered data access.

The source line shown in red in the following example code causes false sharing:

|  |  |
| --- | --- |
| 01 | double sum=0.0, sum\_local[NUM\_THREADS]; |
| 02 | #pragma omp parallel num\_threads(NUM\_THREADS) | |

|  |  |
| --- | --- |
| 03 | { |
| 04 | int me = omp\_get\_thread\_num(); | |

|  |  |  |
| --- | --- | --- |
| 05 | sum\_local[me] = 0.0; | |
| 06 |  |

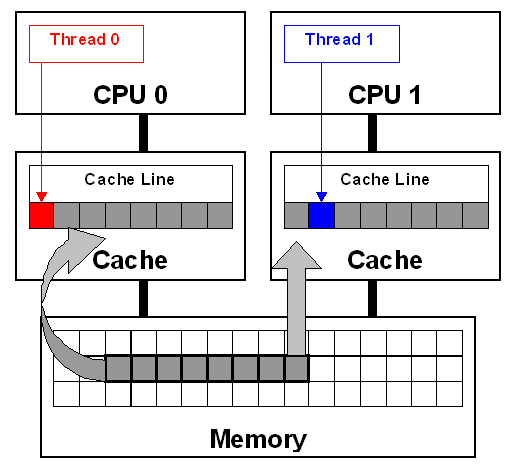
|  |  |
| --- | --- |
| 07 | #pragma omp for |
| 08 | for (i = 0; i < N; i++) | |

|  |  |  |
| --- | --- | --- |
| 09 | sum\_local[me] += x[i] \* y[i]; | |
| 10 |  |

|  |  |
| --- | --- |
| 11 | #pragma omp atomic |
| 12 | sum += sum\_local[me]; | |

|  |  |
| --- | --- |
| 13 | } |

There is a potential for false sharing on array sum\_local. This array is dimensioned according to the number of threads and is small enough to fit in a single cache line. When executed in parallel, the threads modify different, but adjacent, elements of sum\_local (the source line shown in red), which invalidates the cache line for all processors.

  
**Figure 1.** False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces a memory update to maintain cache coherency.

In Figure 1, threads 0 and 1 require variables that are adjacent in memory and reside on the same cache line. The cache line is loaded into the caches of CPU 0 and CPU 1 (gray arrows). Even though the threads modify different variables (red and blue arrows), the cache line is invalidated, forcing a memory update to maintain cache coherency.

To ensure data consistency across multiple caches, multiprocessor-capable Intel® processors follow the MESI (Modified/Exclusive/Shared/Invalid) protocol. On first load of a cache line, the processor will mark the cache line as ‘Exclusive’ access. As long as the cache line is marked exclusive, subsequent loads are free to use the existing data in cache. If the processor sees the same cache line loaded by another processor on the bus, it marks the cache line with ‘Shared’ access. If the processor stores a cache line marked as ‘S’, the cache line is marked as ‘Modified’ and all other processors are sent an ‘Invalid’ cache line message. If the processor sees the same cache line which is now marked ‘M’ being accessed by another processor, the processor stores the cache line back to memory and marks its cache line as ‘Shared’. The other processor that is accessing the same cache line incurs a cache miss.

The frequent coordination required between processors when cache lines are marked ‘Invalid’ requires cache lines to be written to memory and subsequently loaded. False sharing increases this coordination and can significantly degrade application performance.

Since compilers are aware of false sharing, they do a good job of eliminating instances where it could occur. For example, when the above code is compiled with optimization options, the compiler eliminates false sharing using thread-private temporal variables. Run-time false sharing from the above code will be only an issue if the code is compiled with optimization disabled.

Advice

The primary means of avoiding false sharing is through code inspection. Instances where threads access global or dynamically allocated shared data structures are potential sources of false sharing. Note that false sharing can be obscured by the fact that threads may be accessing completely different global variables that happen to be relatively close together in memory. Thread-local storage or local variables can be ruled out as sources of false sharing.

The run-time detection method is to use the Intel® VTune™ Performance Analyzer or Intel® Performance Tuning Utility (Intel PTU, available at [/en-us/articles/intel-performance-tuning-utility/](https://software.intel.com/en-us/articles/intel-performance-tuning-utility/)). This method relies on event-based sampling that discovers places where cacheline sharing exposes performance visible effects. However, such effects don’t distinguish between true and false sharing.