JAVA SE

Java and the modern CPU, Part 2: The unexpected effects of instruction execution

How false sharing and branch misprediction can have unwanted effects on your code's performance

by Michael Heinrichs

February 12, 2021

[*Java Magazine* is pleased to republish a popular series on CPUs that was written in 2015. We have made only a few updates to the original articles. —*Ed*.]

The first article in this series, "Java and the modern CPU, Part 1: Memory and the cache hierarchy," covered the effects of modern chip design on programming and focused on the memory system, specifically the cache hierarchy. In this article, part 2, I conclude this investigation by examining the issues of multithreaded access.

You may wish to read or reread part 1 to familiarize yourself with caching (including the differences between the L1, L2, and L3 caches) and cache lines.

False sharing and the MESI cache protocol

In the first experiment, about false sharing, you will see a surprising effect that can occur when memory is accessed concurrently from different threads. The code in **Listing 1** contains an array of 17 integers and four methods that modify different elements of the array.

The experiment consists of two test runs in which you can run two of the methods concurrently on different threads. The first test run will execute the methods `modifyFarA()` and `modifyFarB()`, which modify two array elements that are 16 elements apart. The second test run will execute the methods `modifyNearA()` and `modifyNearB()`, which modify two

adjacent array elements. The only difference is how far apart the modified array elements are. How does that affect performance?

Listing 1. Code that runs methods concurrently on different threads

```
public final int[] array = new int[17];
@Benchmark
@Group("near")
public void modifyNearA() {
array[0]++;
}
@Benchmark
@Group("near")
public void modifyNearB() {
array[1]++;
}
@Benchmark
@Group("far")
public void modifyFarA() {
array[0]++;
}
@Benchmark
@Group("far")
public void modifyFarB() {
array[16]++;
}
```

On my laptop, a method call in the first test run takes about 3.6 ns, while in the second test run it takes 4.5 ns. In other words, if the array elements are farther apart, modifications are 25% faster. How can that be?

There are actually two puzzles to solve here.

- First, why does it even matter how far apart the elements are?
- Why do these methods interfere with each other at all, even though they access different variables?

You may have noticed the similarity to the initial example in the first article of this series. If elements of an integer array are 16 elements or more apart, they will be located in different cache lines. If they are closer to each other, chances become great that they will end up in the same cache line. Is the observed variation in behavior in this article's experiment also related to cache lines?

Indeed, it is. The computer's memory system has no understanding of Java. It does not know about Java arrays and Java variables; the smallest unit the memory system understands is a cache line. Therefore, if two threads modify the same cache line, those threads interfere with each other, slowing performance. It does not matter that you modified two different variables in the source code; this is a question of machine architecture, not software logic.

Why do two threads that modify the same cache line slow each other down? You have to fill out your model of the cache hierarchy to understand this effect. When two threads run in parallel long enough, they end up on different CPU cores. CPU cores do not share an L1 cache—each core has its own. (Note: How the L2 and L3 caches are shared between cores depends on the architecture. A typical setup is that the L2 cache is shared between *adjacent* cores while the L3 cache is shared between *all* cores.)

As each core copies the cache line into its respective L1 cache and executes the update to the variable, the core notifies the other cores of the update and tells them to refresh their L1 cache in case that L1 cache is out of sync.

Modern computers use a variant of the Modified Exclusive Shared Invalid (MESI) protocol to synchronize the L1 caches. The protocols used today have been improved over the years, but the basic principle remains the same.

In the MESI protocol, every cache line that was loaded into the cache is in one of four states: *Modified*, *Exclusive*, *Shared*, or *Invalid*:

- The Modified state means that the cache has an exclusive copy of the cache line and has modified it.
- If a cache line is in the Exclusive state, it also means that it is an exclusive copy, but it has not been modified yet.
- A cache line is in the Shared state if there are copies in two or more caches, but none of them has been modified.
- An Invalid cache line is no longer valid and cannot be used.

Figure 1 shows the state changes during this experiment.

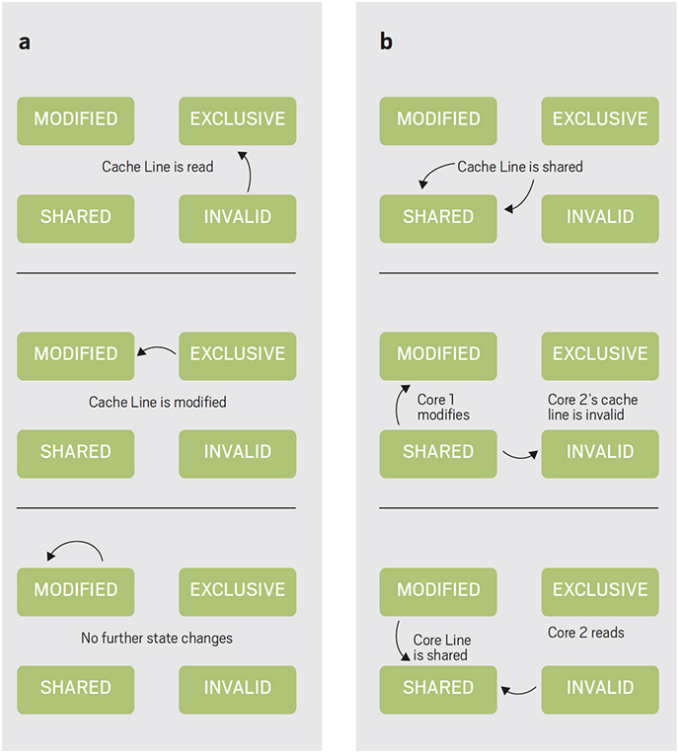

Figure 1. MESI state changes while modifying elements (section *a*: distant elements, section *b*: adjacent elements)

In **Figure 1**, section *a* shows the state changes during the first test run, when the array elements were farther apart and the threads accessed different cache lines.

When one of the threads loaded a cache line into the cache, the line was marked as being Exclusive. When the thread modified it, the state switched to Modified. From then on, the cache line remained in the Modified state. Access to the cache line was fast because the Modified state means the cache line can be updated directly without any further actions.

Things become more complicated during the second test run, as can be seen in section *b* of **Figure 1**. The MESI protocol ensures that the system never works with two different versions of the same cache line.

Both threads modified the same cache line; therefore, when the second thread requested the cache line, it had to load it from the L1 cache of the first core and both caches had to mark their respective cache lines as being in the Shared state. When one of the two threads tried to modify the cache line, it first had to signal the other cache that its cache line was in the Invalid state.

Only after that was successful and its cache line entered the Modified state was it able to do the modification. But when the other thread requested the cache line, both caches had to synchronize again and switch to the Shared state. Then the process had to start all over again. The constant need to mark the other thread as invalid and synchronize back again once

done was the reason why the second test ran slower than the first.

The problem of two seemingly independent variables affecting each other's performance is so common that it has its own name: *false sharing*. It is extremely tricky to detect, because false sharing is invisible if you look at Java code alone. One of the examples in the Java Microbenchmarking Harness gives a good overview of different techniques to avoid false sharing between two variables. The basic idea of all solutions is to insert other variables to ensure that the two variables potentially affected by false sharing are located in different cache lines.

Linear search and binary search show the cost of branch misses

At this point, I'll leave the memory system behind to take a closer look at the CPU itself. The next experiment will try to answer a simple question: If it is executed on a small array, which is faster: a linear search (as shown in **Listing 2**) or a binary search (as shown in **Listing 3**)?

Listing 2. Code for a linear search

```
public static boolean linearSearch(
int needle, int[] haystack) {
for (int current : haystack) {
if (current == needle) {
return true;
} else if (current > needle) {
return false;
}
}
return false;
}
```

Listing 3. Code for a binary search

```
public static boolean binarySearch(int needle
int[] haystack) {
int low = 0;
int high = haystack.length - 1;
while (low <= high) {
int mid = (low + high) >>> 1;
int midVal = haystack[mid];
if (midVal < needle)
low = mid + 1;
else if (midVal > needle)
high = mid - 1;
else
return true;
}
return false;
}
```

As you would expect by examining the algorithms, for large arrays, the binary search outperforms the linear search by a wide margin. The larger the array, the larger the performance gap. But for small arrays, such as one of 16 integers, the answer is not straightforward because the outcome depends on the CPU architecture.

On my laptop, a binary search takes 28.2 ns on average, whereas a linear search takes only 21.8 ns. This is counterintuitive because the binary search requires fewer iterations. Why is the linear search faster?

To get an initial idea, let's run `perf` on both algorithms. (For a quick introduction to using `perf`, see the first article in this series.) The performance numbers are roughly the same, but there is a significant difference in the number of *branch misses*. The linear search had 346.9 million branch misses, while the binary search had 526.4 million. That can't be good.

What is a branch miss? To understand branch misses, you need to take a step back and take a look at a mechanism called *pipelining*.

When the CPU processes an instruction, it actually has several steps to perform. The instruction needs to be fetched from memory and decoded. That is, the CPU must figure out what kind of instruction it is dealing with. After that, the instruction needs to be executed and the result has to be written back to memory. **Figure 2** shows the internal structure of a CPU and how a single instruction A steps through the different stages on a mythical CPU with no pipeline.

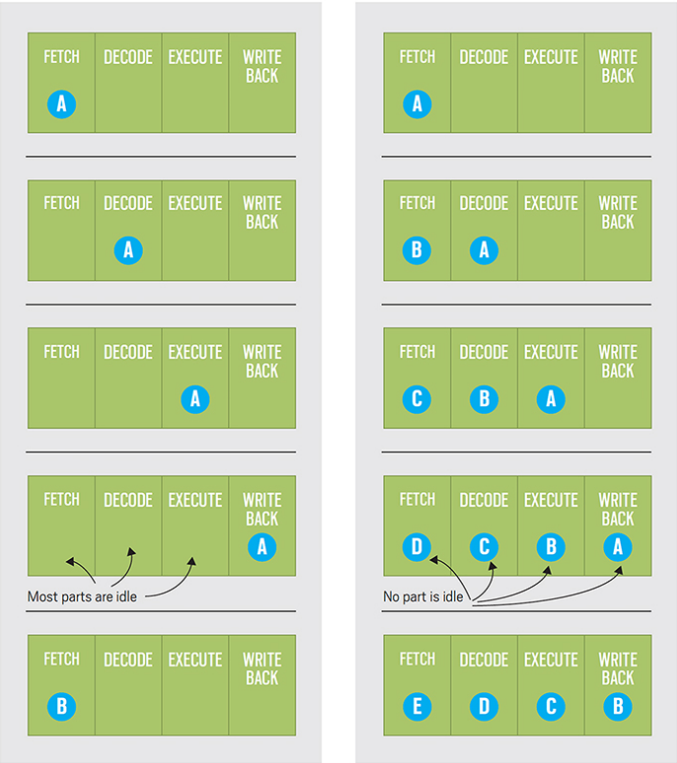

Figure 2. On the left, CPU execution without pipelining; on the right, CPU execution with pipelining

The operations performed at each stage are quite different, in part because the stages are implemented in different parts of the CPU. Thus, if a processor really worked as shown on the left side of **Figure 2**, it would be inefficient because most of its parts would be idle most of the time, waiting for the next instruction to arrive. Early on, chip designers thought about ways to keep all components busy and came up with a solution called pipelining.

The principle can be seen on the right side of **Figure 2**. Instruction A is fetched. While instruction A is decoded, the CPU fetches the next instruction, B. As it executes instruction A, the CPU decodes instruction B, fetches the third instruction C, and so on. Instead of processing one instruction after another, the CPU processes a stream of instructions: the instruction pipeline.

This works extremely well. A single instruction still needs four cycles to step through all four stages, but with pipelining the throughput increases from one instruction every four cycles to one instruction per cycle.

That leads to *branch prediction*: a situation where pipelining becomes tricky because of conditional jumps.

Imagine that instruction A is a conditional jump and that, depending on the outcome, the software may execute instruction B or jump directly to a distant instruction X. Should the CPU pipeline fetch instruction B or X while it's busy decoding A? The CPU does not know the outcome of the conditional jump yet, because that becomes clear only after instruction A has finished the execution stage.

Different processor architectures deal with this situation differently. One of the most common strategies is branch prediction: The CPU simply guesses which branch it should take. Even though the prediction algorithm has to be simple and fast, it is amazing how accurate those guesses are. In a typical program, the rate of successful predictions is well above 90%.

What happens if the guess is wrong? The effect of a wrong guess and unnecessary execution is less dramatic than you might expect.

Look at **Figure 3** and assume that instruction A is a conditional jump for which the CPU did a wrong guess. Once instruction A finishes the execution stage, the CPU knows that the branch prediction was wrong. It can simply drop instructions B and C now. At this point, they have only been fetched from memory and decoded. To the outside world, both operations have no noticeable effects. Therefore, the CPU can just throw out these instructions and continue fetching the first instructions of the correct branch.

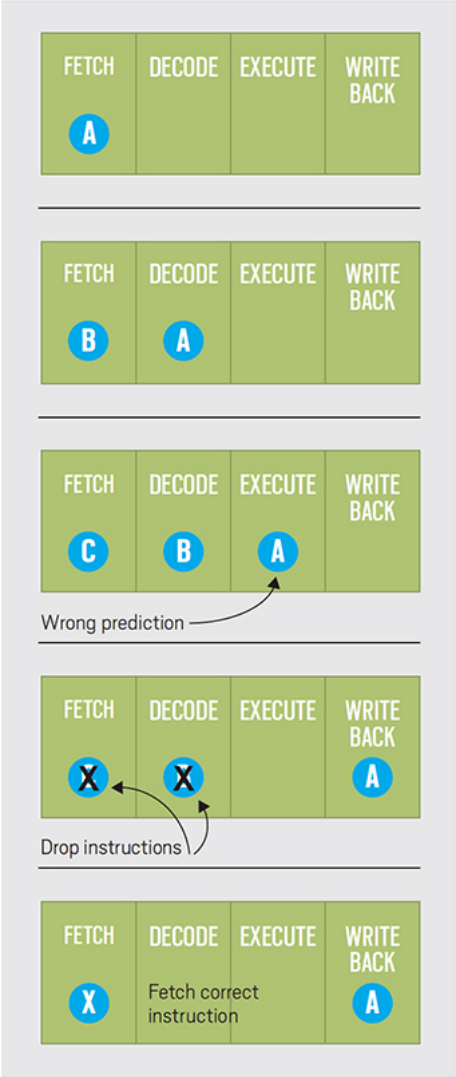

Figure 3. Branch misprediction

Even though they don't happen often, branch mispredictions are expensive. Very expensive, in fact, because they can nullify the performance increase gained through pipelining the instructions. In the case of the two search algorithms, the performance loss is large enough to make the binary search the slower alternative, even though it requires fewer comparisons to find the needle in the haystack.

The algorithm for the linear search contains three conditional jumps: the `for` loop; the check for whether the element was found; and the check on the range of possible indexes. Each of those are generally easy to predict correctly. The loop index is always smaller than the array bounds except at the very end, when the loop is exited. This means that once the loop is entered, every guess is likely to be right except the one at the very end. The same is true for the other two conditional jumps. If the CPU were to guess that the current element is not the element being searched for, it will always be right except once, when the element is found and the loop is exited. So, you see fast performance.

The algorithm for the binary search also contains three conditional jumps. Two of them are easy to guess right most of the time, for the same reason as described above. But in the loop, a decision must be made to continue in the upper half or lower half of the remaining array, and here test data is evenly distributed, which means chances are 50-50 that the code needs to go to one half or the other. In other words, it is impossible to guess the right branch at this point. No matter which branch is guessed, it will be wrong half the time. That slows the code down on fairly small arrays, as in this experiment.

Conclusion

The instruction pipeline and branch predictions are a level that most developers usually do not need to consider when optimizing Java programs.

I have encountered a handful of cases when optimizing the code to avoid branches actually did improve performance noticeably in very hot code segments. It's not common, but sometimes it can make a real difference.

What these examples show is that for very small data structures, it often makes sense to use the simplest algorithm possible. The positive effects of clever but complex algorithms can easily be nullified by other effects. Therefore, it makes even more sense to adhere to the most important rule in software engineering: If in doubt, follow the KISS principle—that is, keep it simple.

Dig deeper

- Java and the modern CPU, Part 1: Memory and the cache hierarchy
- Java and branch prediction
- Branch prediction — Everything you need to know
- Loop unrolling
- Branch mispredict
- What false sharing is and how JVM prevents it

Michael Heinrichs

Michael Heinrichs is co-founder of Karakun AG in Freiburg, Germany. He's a Java Champion as well as founder and leader of the Java user group in Freiburg. Follow him on Twitter at @net0pyr.