

MICHAEL HEINRICHS



Part 2

Inside the CPU: the Unexpected Effects of Instruction Execution

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

The [first article](#) in this series about the effects of modern chip design on programming focused on the memory system, specifically the cache hierarchy. In this article, Part 2, I extend our investigation to the issues of multithreaded access. In Part 3, I will look at the internal workings of the CPU itself.

False Sharing

In our first experiment, we 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 we run two of the methods concurrently on different threads. In the first test run, we will execute the methods `modifyFarA()` and `modifyFarB()`, which modify two array elements that are 16 elements apart. In the second test run, we 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.**

```
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 percent faster. How can that be?

There are actually two puzzles to solve here. Why does it matter how far apart the elements are? And why do these

methods interfere with each other at all, even though they access different variables?

Watchful readers will notice 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 are great that they will end up in the same cache line. Is the observed variation in behavior related to cache lines?

Indeed, it is. The memory system has no understanding of Java. It does not know about Java arrays and Java variables. Its smallest unit is a cache line. If two threads modify the same cache line, they interfere with each other and it does not matter that we modified two different variables in the source code.

Why do two threads that modify the same cache line slow each other down? We have to fill out our 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. For example, a typical setup is that L2 cache is shared between adjacent cores while L3 cache is shared between all cores.)

As each core copies the cache line into its respective L1 cache and does the update to the variable, the core notifies the other cores of the update and tells them to freshen the L1 cache copy they own.

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, 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 our experiment. **Figure 1a** shows the state changes during the

The MESI protocol ensures that **the system never works with two different versions of the same cache line.**

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 tells us that we can update the cache line

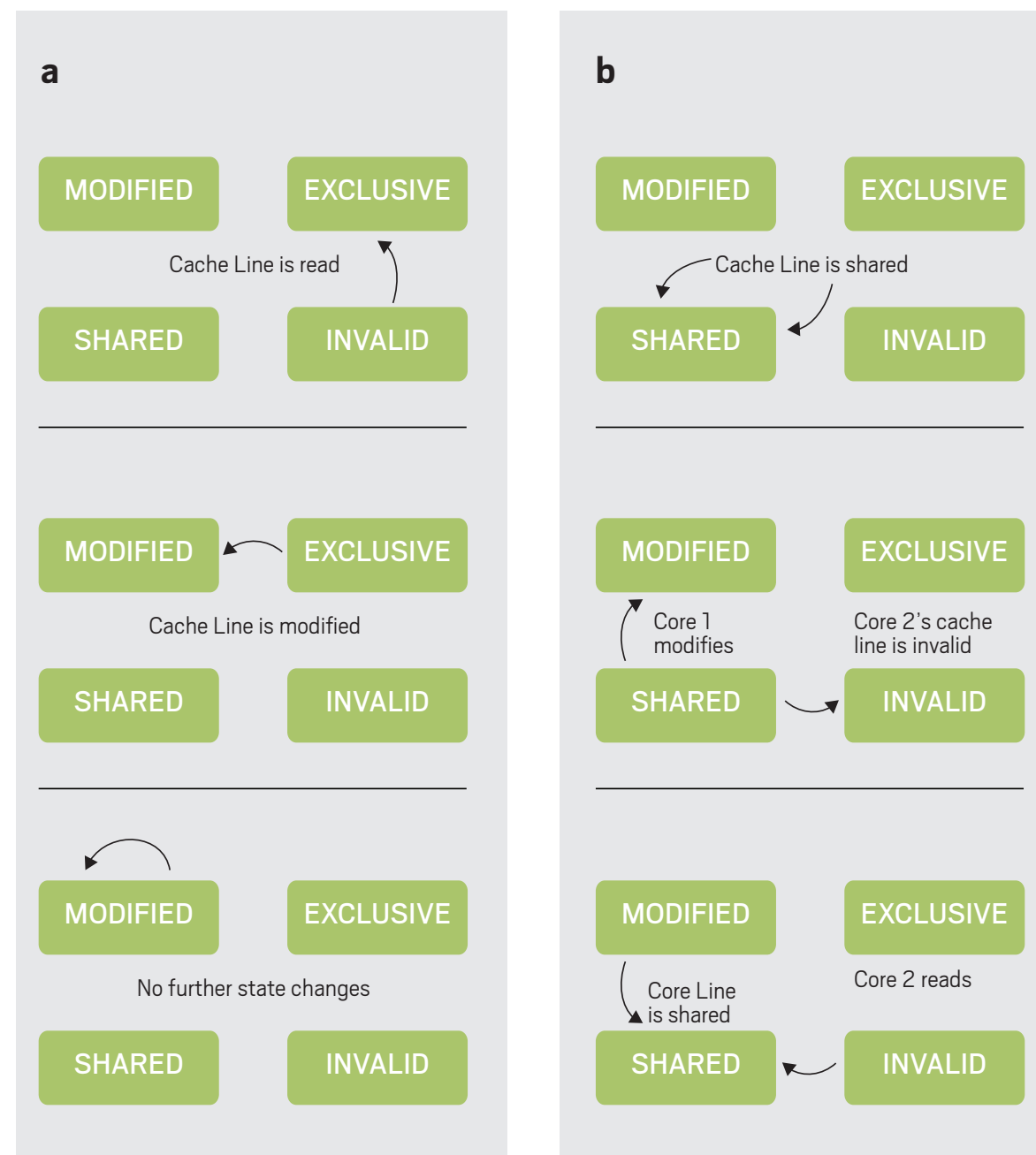


Figure 1: MESI state changes while modifying elements (1a: adjacent elements, 1b: distant elements)

formance numbers are roughly the same, but there is a significant difference in the number of *branch misses*. [Screen shots with full perf results are available in the [download area](#). —Ed.] The linear search had 346.9 million branch misses, while the binary search had 526.4 million. What is a branch miss? To understand branch misses, we need to take a step back and take a look at a mechanism called *pipelining*.

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.

The operations performed at each stage are quite different. The stages are implemented in different parts of the CPU. Thus, if a processor really worked as shown in **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 in **Figure 3**. 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.

Branch prediction. There is one case, however, in which pipelining becomes tricky: conditional jumps. Imagine that instruction A is a conditional jump and that, depending on the outcome, it may execute instruction B or jump directly to a distant instruction X. Should the CPU fetch instruction B or X while we decode A? It does not know the outcome of the con-

ditional 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 guesses

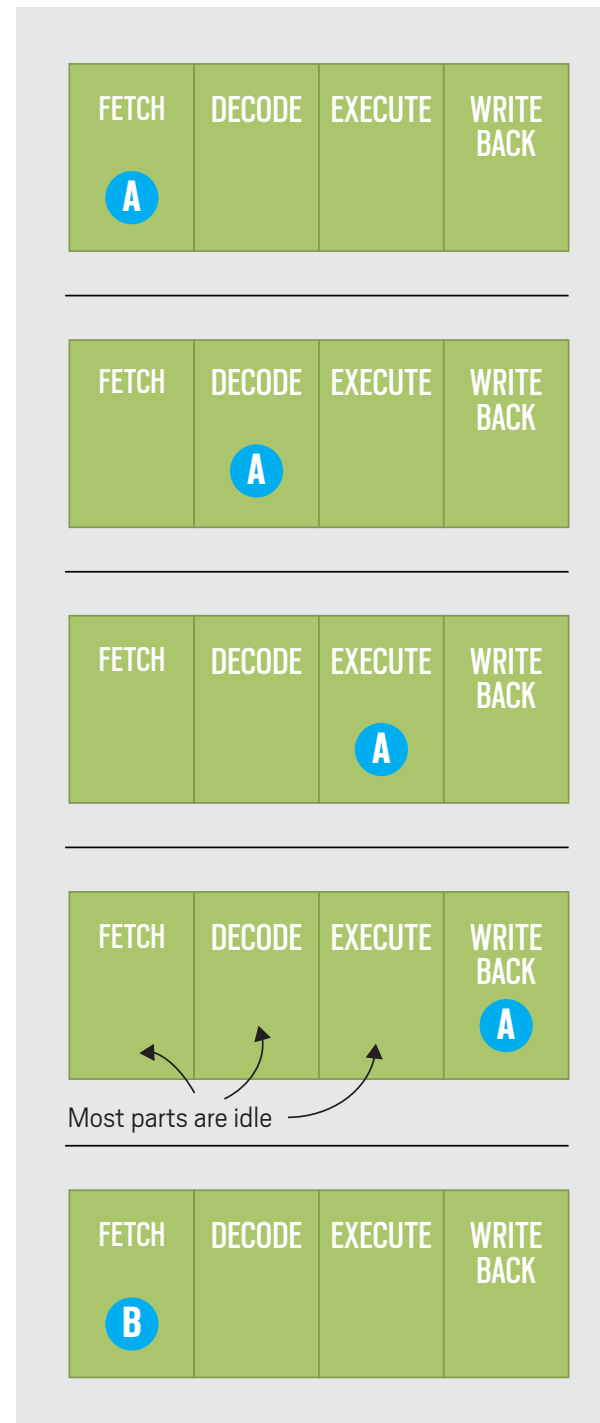


Figure 2: CPU execution without pipeline

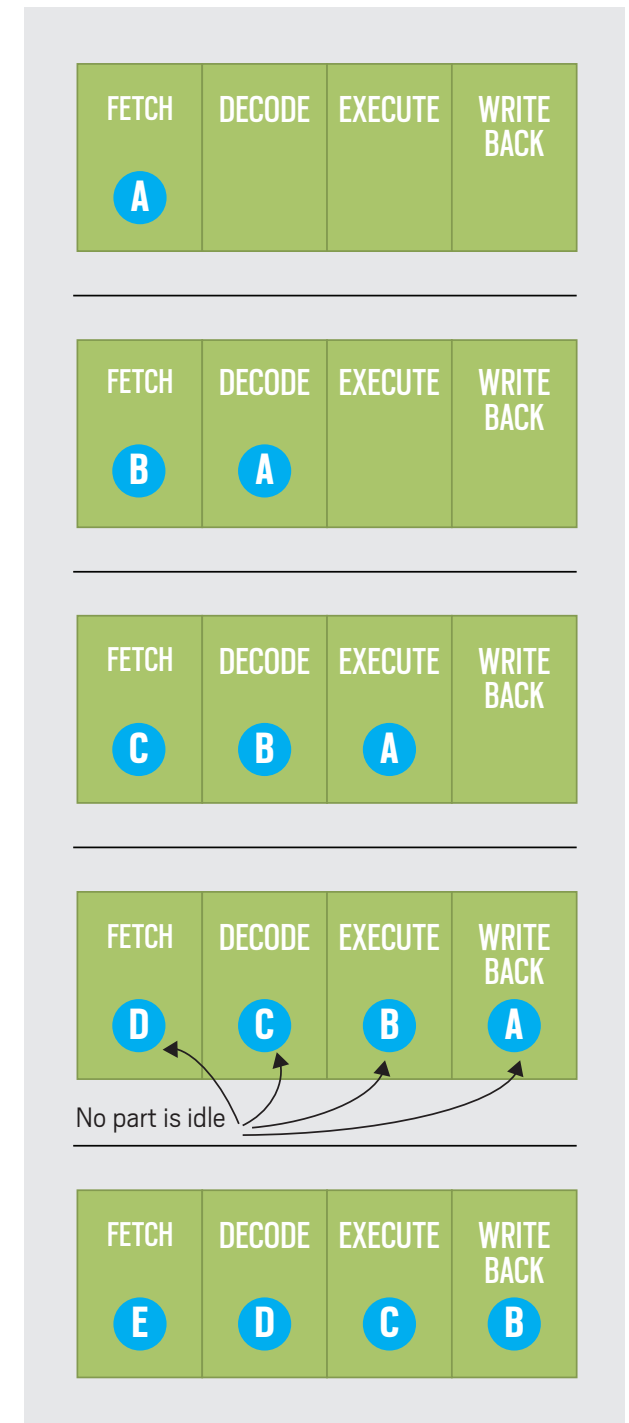


Figure 3: CPU execution with pipeline

In a typical program, the CPU's rate of **successful branch predictions** is well above **90 percent**.

which branch it should take. Even though the prediction algorithm has to be simple and fast, it is amazing how accurate it is. In a typical program, the rate of successful predictions is well above 90 percent.

What happens if the guess is wrong? The effect of a wrong guess and unnecessary execution is less dramatic than one might expect. Take a look at **Figure 4** 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 4 illustrates the effect of a wrong branch prediction. The diagram shows three stages of instruction execution across four pipeline stages: FETCH, DECODE, EXECUTE, and WRITE BACK.

- Stage 1:** Instruction A is in the EXECUTE stage.
- Stage 2:** Instruction B is in the EXECUTE stage, and Instruction A is in the WRITE BACK stage.
- Stage 3:** Instruction C is in the EXECUTE stage, Instruction B is in the WRITE BACK stage, and Instruction A is in the WRITE BACK stage.

An arrow points to the 'Wrong prediction' label, indicating that the CPU has determined that the branch prediction was incorrect and will discard instructions B and C.

Branch prediction while search-
ing. Nevertheless, branch
mispredictions are expensive,
because they nullify the per-
formance increase we gained
through pipelining our instruc-
tions. In the case of our two
search algorithms, the perfor-
mance loss is large enough to
make binary search the slower
alternative, even though it
requires fewer comparisons to
find the needle in the haystack.

The algorithm for 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 results in a conditional jump. Even though this is quite a few conditional jumps for such a short code segment, all of them are 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.

The algorithm for 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. The 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.

Conclusion

With the instruction pipeline and branch predictions, we have reached a level that you usually do not need to consider when optimizing your Java program. I have encountered maybe a handful of cases so far where avoiding branches actually did improve performance noticeably in very hot code segments. But pipelining and branch prediction are fascinating nevertheless. What this example shows 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. **</article>**

LEARN MORE

- Branch mispredicts using assembly on Intel processors
- "Eliminate False Sharing"
- Using padding in Java to avoid false sharing

Figure 4: Branch misprediction