



MICHAEL HEINRICHS

BIO

Part 1

# The Quantum Physics of Java

An introduction to modern chip design and its effect on Java programs

Imagine you have an array with 67,000 integer elements and you run two loops over the array, as shown in **Listing 1**. Both loops multiply the elements of the array by three. However, while the first loop changes every element, the second loop modifies only every sixteenth element.

How much faster will the second loop be compared to the first? Take a guess!

The surprising answer is that if the code is executed on a typical laptop, both loops take the same amount of time. **Table 1** shows measurements from three computers. The difference is negligible. The second loop does only a fraction of the work, so how is it possible that the first loop runs as fast as the second?

To understand this behavior you have to consider how the CPU and the memory system work. On the lowest level, modern

computers can show surprising behavior, very much like quantum mechanics, that seems to contradict daily experience. But sometimes quantum mechanics has noticeable effects on our “real world.” And sometimes the effects of processes at the hardware level have a noticeable effect on our programs. This article takes a look at how modern computers work at the lowest level and explores the things that can affect performance.

### Instructions

If you try to imagine how the loops in **Listing 1** are executed, your initial interpreta-

tion might be that the array is stored in main memory and the CPU reads element after element, multiplies each element by three, and writes the result back, as you can see in **Figure 1**. This interpretation is useful for understanding the functionality of the loops, but it’s not what really happens inside a computer.

**Figure 2** shows a graph of relative performance improvements that CPUs and memory went through in recent decades. Memory performance improved steadily during the whole period, but that was nothing compared to the improvements in CPU speed, especially during the

1990s. In recent years, plain CPU speed hit a limit, but do not be fooled! The scale in **Figure 2** is logarithmic. Even

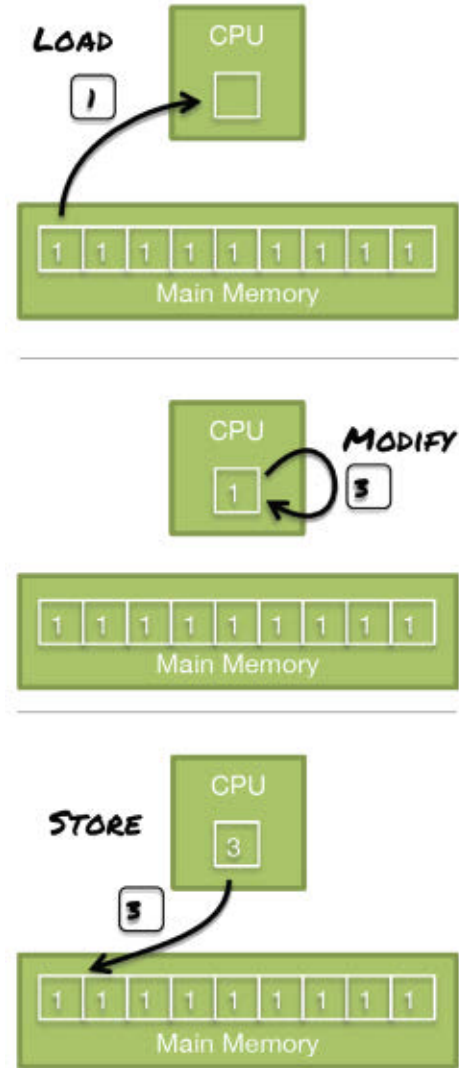


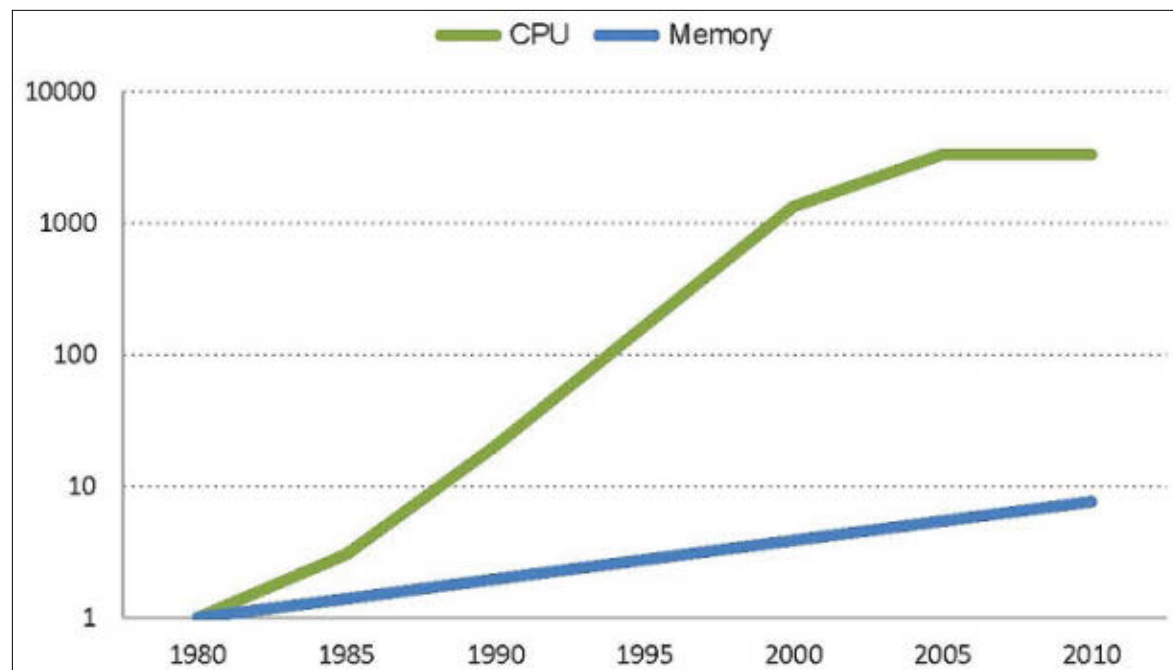
Figure 1

	SMALL STEPS EACH ELEMENT	LARGE STEPS EVERY 16TH ELEMENT
I7-4980HQ @ 2.8 GHZ, MAC OS X YOSEMITE	30.4 MS	29.7 MS
I7-3770 @ 3.4 GHZ, LINUX MINT 14	25.8 MS	26.1 MS
T7200 @ 2 GHZ, LINUX MINT 14	193.0 MS	184.2 MS

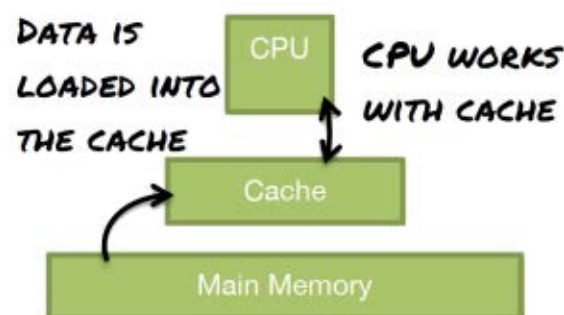
Table 1



PHOTOGRAPH COURTESY OF DEVOXX



## Figure 2



### Figure 3

though it might look as if memory performance is catching up, the gap is still huge.

For our example, this means that if a computer worked as we imagined in **Figure 1**, it would be terribly slow. The CPU would wait most of the time for the memory to deliver the next element. To overcome this bottleneck, processor designers added a cache between the CPU and main memory. The cache is a

smaller and much faster memory module, whose whole purpose is to mitigate the performance gap. **Figure 3** shows an improved model of the CPU and memory system.

Programs tend to access the same data and code several times within a short period of time (*temporal locality*), and memory access is often limited to small regions (*spatial locality*). This means that if you load all the data that you use into the cache, there is a high chance that you'll need it again later. And because the next time you need it the data is available in the cache, the performance of your programs increases tremendously.

Now you might wonder—if we can put faster memory between CPU and main memory, why can't

## LISTING 1

```
private static final int ARRAY_SIZE = 64 * 1024 * 1024;  
public int[] array = new int[ARRAY_SIZE];
```

```
for (int i = 0, n = array.length; i < n; i++) {
    array[i] *= 3;
}
```

```
for (int i = 0, n = array.length; i < n; i+=16) {
    array[i] *= 3;
}
```

 [Download all listings in this issue as text](#)

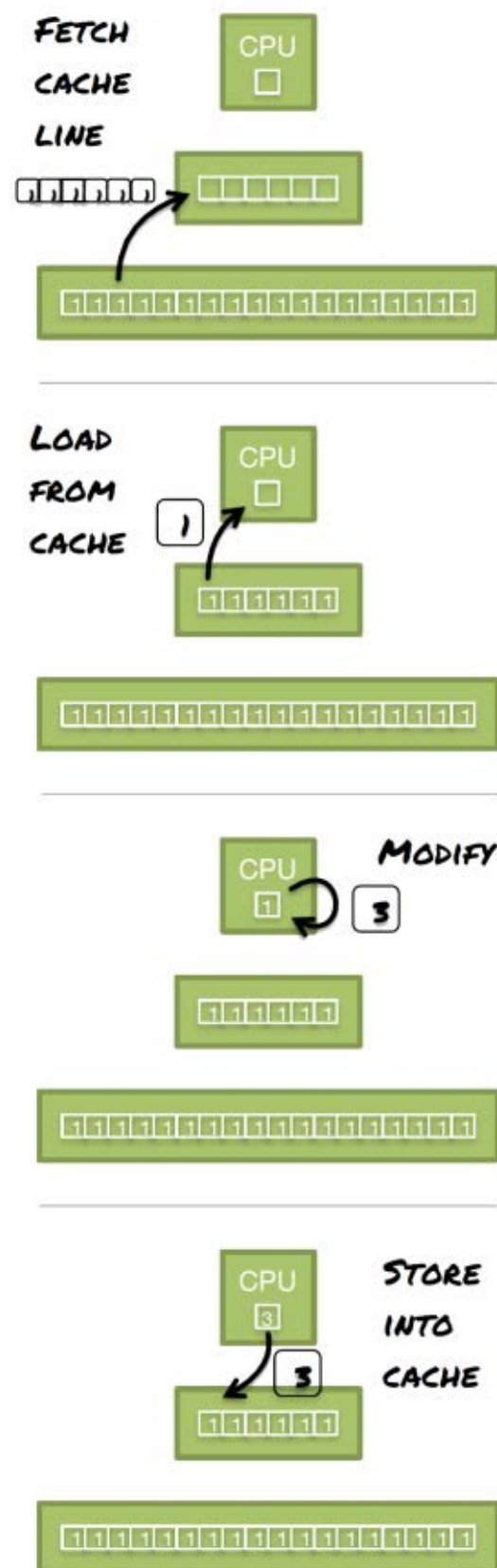
we just make the whole memory faster? There are mostly two reasons. For one, main memory is a lot larger than cache, and it simply takes more time to find the right address within 16 GB (the typical size of main memory, as of this writing) than to find the right address within 8 KB (the typical size of Level 1 [L1] cache). However, probably the more important reason is that the electronic components of cache (SRAM) are much more expensive than the ones used in main memory (DRAM) in terms of heat and space. Heat and space are the limiting factors in modern chip design.

To exploit spatial locality, the cache doesn't work with individual bytes but uses cache lines instead.

A *cache line* is an adjacent part of the memory, typically 64 bytes.

What really happens when you iterate over the large loops from **Listing 1** can be seen in **Figure 4**. The CPU loads a complete cache line from main memory into the cache and modifies the elements in the cache directly. The first loop modifies all elements in the cache line, while the second loop modifies only one element (16 integers, each 4 bytes long). The limiting factor in this setup is loading the cache line into the cache; it almost doesn't matter how many operations you execute on each cache line. This explains why the performance of both loops is roughly the same.

Counting instructions to estimate the performance of an algorithm is



### Figure 4

a useful approximation, because it's easy to assess and usually gives a good indication. But as this example shows, you have to keep in mind that it's just an approximation. In reality, the execution times of single instructions vary widely, and you can't rely on this number only.

## Data Size

Does the size of a data structure affect performance? To answer this question, run a small experiment using the code in **Listing 2**. Take the second loop from the first code example and run it repetitively. This time, change the size of the array and measure the average time to run a single loop iteration. The purpose of this experiment is to run a trivial algorithm over a data structure whose size you can control. Is there a relationship between the size of the array and the time needed to modify a single element?

Before you look at the results, consider briefly what you expect. Accessing a single array element requires constant time,  $O(1)$ . Thus, the inner part of the loop should be executed in constant time, too. That means for large enough arrays, you will hit an upper bound that is constant. But what happens before that? Will the execution time be constant all the way through?

**Figure 5** shows the dependency between array size and access time.


## LISTING 2

```
private static final int ARRAY_CONTENT = 777;
@Param({"1024", "2048", "4096", "8192", "16384", ..., "536870912"})
public int size;

public int[] array;
public int counter;
public int mask;

@Setup(Level.Iteration)
public void setUp() {
    final int elements = size / 4;
    final int indexes = elements / 16;
    mask = indexes - 1;
    array = new int[elements];
    Arrays.fill(array, ARRAY_CONTENT);
    counter = 0;
    for (int i = 0; i < indexes; i++) {
        seqIndex[i] = 16 * i;
    }
}

@Benchmark
public void benchLoop() {
    array[16 * counter] *= 3;
    counter = (counter + 1) & mask;
}
```

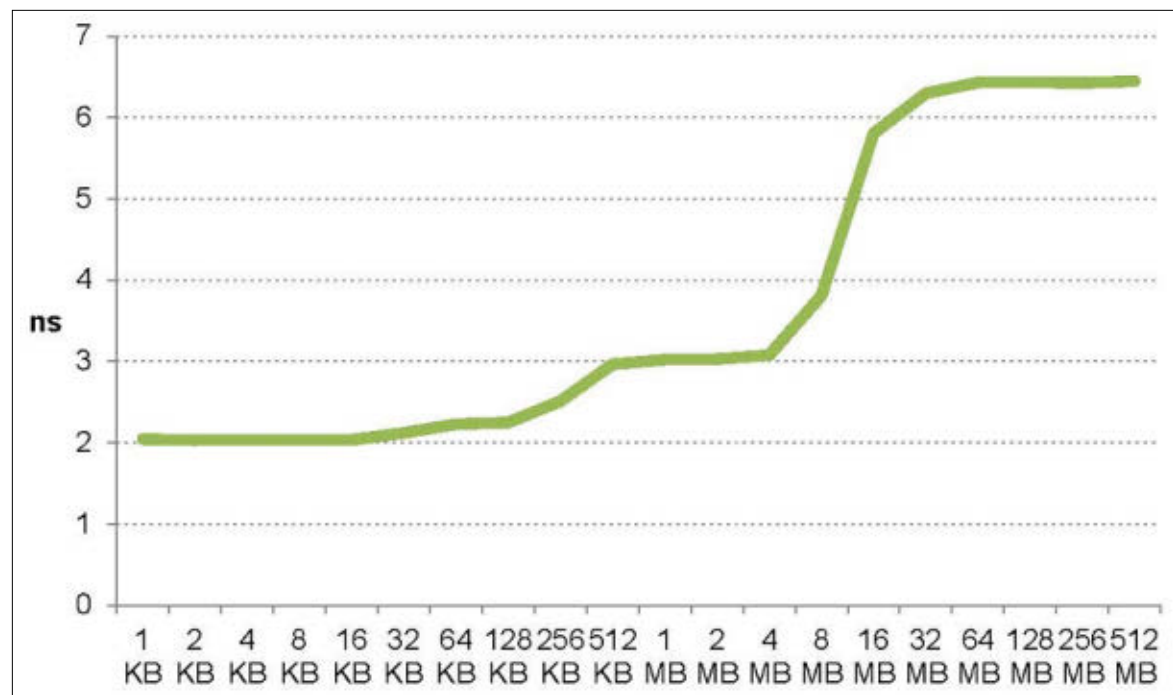
 [Download all listings in this issue as text](#)

As you can see, there is a relationship between these values. A single modification is faster if the array is small. But it's not that simple. The resulting curve looks like a staircase. The access time remains constant until the array size exceeds a specific threshold, and

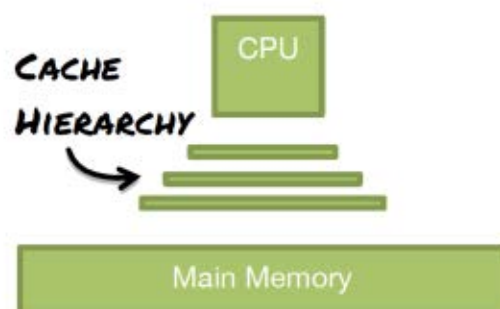
then it jumps to a new level where it remains until the next threshold is reached. Why is there a dependency at all, and where do these levels come from?

The cache is usually not a single unit, but consists of several levels with different sizes and access





### Figure 5



### Figure 6

times. L1 cache is the smallest and fastest. Current CPUs typically have three cache levels, with each level being slower and significantly larger than the level before. **Figure 6** shows an improved version of our model that contains the cache hierarchy.

How large are the performance gaps between the different cache levels? To explain this in a form

that is more accessible to human beings, my former colleague Richard Thompson came up with the beer cache hierarchy. Imagine that you're sitting in front of your TV watching your favorite team and you're thirsty.

- L1 cache is the bottle of beer in your hand. Access time is almost immediate ( $< 1$  ns), but the quantity is extremely limited (for example, 32 KB on my system).
- L2 cache is the cooler next to your sofa. Access time is still pretty low (7 ns), and the quantity is significantly larger (256 KB, which is equivalent to 8 bottles of beer).
- L3 cache is the fridge in the kitchen. Access time is noticeably

### LISTING 3

```
public int[] rndIndex;

@Setup(Level.Iteration)
public void setUp() {
    ...

    rndIndex = new int[indexes];
    final List<Integer> list = new ArrayList<>(indexes);
    for (int i=0; i<indexes; i++) {
        list.add(16 * i);
    }
    Collections.shuffle(list);
    for (int i=0; i<indexes; i++) {
        rndIndex[i] = list.get(i);
    }
}
```

 [Download all listings in this issue as text](#)

larger (25 ns), but the size is so large that the analogy falls apart (8 MB, which is equivalent to 256 bottles of beer).

- Main memory is the corner store. Access time is huge (100 ns), but the quantity of beer is probably more than enough for a lifetime (16 GB, which is equivalent to more than half a million bottles of beer).

Looking at these numbers, it becomes quite obvious why both loops in the initial example took the same amount of time. It doesn't really matter how many sips of beer you drink if you have to run to the corner store for each bottle.

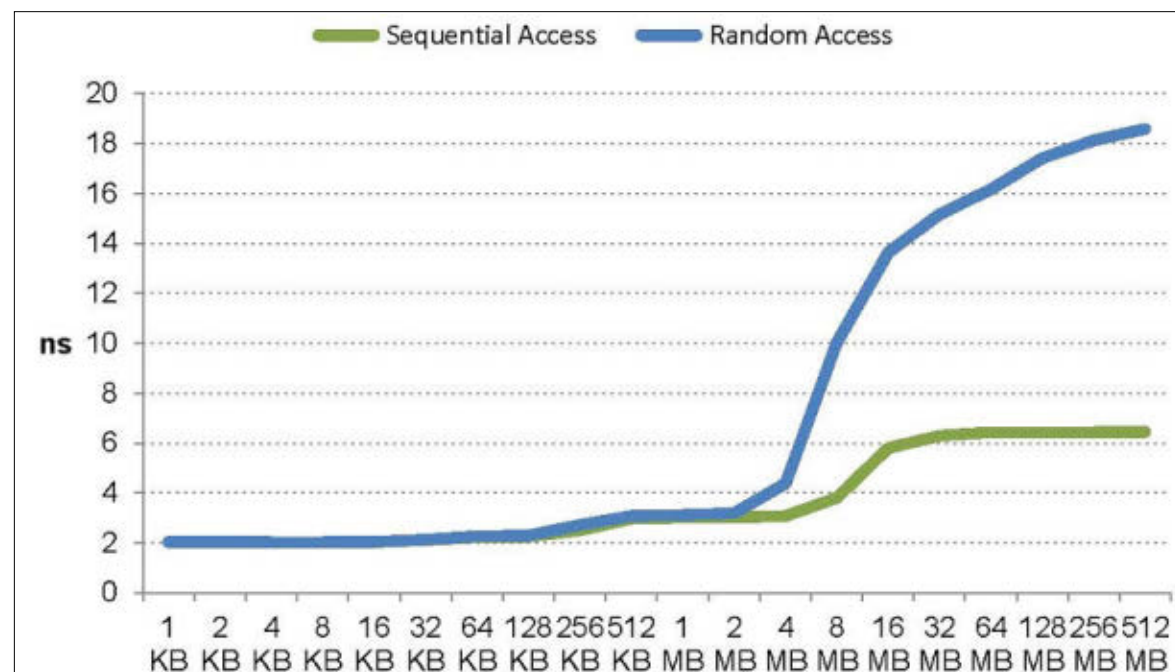
With the cache level hierarchy in mind, take a look at the graph in **Figure 5**. Each plateau in the graph corresponds to a level of the cache hierarchy. As long as the array fits into L1 and L2 cache, access time is very low. But as soon as the array becomes too large and has to be read from L3 cache, access time increases noticeably. And the same happens again as soon as the array does not fit into L3 cache and has to be read from main memory. If you look closely, you can even see the small jump between the L1 and L2 cache.

Size does matter. Even though memory is cheaper than ever

before, try to avoid wasting it. The smaller the size of the data you use, the higher the chance that it will fit into the cache, which can lead to significantly better performance.

## Access Patterns

So the size of data influences performance. Does the order in which you access your data—the data access pattern—have an influence, too? You can change the previ-



### Figure 7

```
michael@desktop ~/repos/quantum/target/classes $ perf stat -p 3195 -B -r 40 sleep 5

Performance counter stats for process id '3195' (40 runs):

    4997,277608 task-clock                #    0,999 CPUs utilized          ( +-  0,00% ) [100,00%]
         529 context-switches            #    0,106 K/sec                  ( +-  0,04% ) [100,00%]
          0 CPU-migrations                #    0,000 K/sec                  [100,00%]
          0 page-faults                  #    0,000 K/sec                  ( +- 35,51% )
19.429.967.708 cycles                    #    3,888 GHz                    ( +-  0,00% ) [83,28%]
19.006.043.778 stalled-cycles-frontend  #   97,82% frontend cycles idle   ( +-  0,00% ) [83,31%]
18.296.349.545 stalled-cycles-backend   #   94,17% backend  cycles idle   ( +-  0,01% ) [66,74%]
  1.386.651.837 instructions              #    0,07 insns per cycle         ( +-  0,05% ) [83,37%]
                                           #   13,71 stalled cycles per insn ( +-  0,07% ) [83,37%]
    23.934.690 branches                  #    4,790 M/sec                  ( +- 11,00% ) [83,30%]
     23.139 branch-misses                #    0,10% of all branches
    5,000586757 seconds time elapsed      ( +-  0,00% )
```

### Figure 8

ous experiment slightly to find an answer. Instead of simply running an index through the array, create a second array that stores the access order. Access the array sequentially as before for one time, and then access it randomly and measure the difference. You can see the code for both experiments in **Listing 3**.

If you run the experiment with different array sizes and plot the result in a graph, you get two curves, as shown in **Figure 7**. Not surprisingly, you can see the already familiar staircase pattern. Both curves show similar access times on the lower two levels, which correlate to the L1 and L2 caches. But on the third level, the performance of the sequential access pattern is noticeably better. The fourth level shows a significant difference. Why

is the access order insignificant for small arrays, but plays a major part for large arrays?

## A Valuable Tool

To get a better understanding of what is going on inside the computer, you can use the Linux profiler tool `perf`, which collects and prints out events generated by the CPU and the memory system while a program is executed. The command-line interface for `perf` is similar to that of `git`. You call `perf` with the command you want to execute:

■ perf COMMAND [ARGS]

To get a list of all commands, use `perf -help`. To get help for a specific command, you can run:

## perf help COMMAND

The most useful command is `stat`. It allows `perf` to run another program and tracks hardware events during execution:

■ perf stat [ARGS] PROGRAM

Without any arguments, this command starts **PROGRAM**, tracks some general events, and prints out the statistics as soon as the program ends. You can see a typical output in **Figure 8**. You can

specify which events should be tracked by using the `-e` option. To get a list of supported events, run the command:

perf list

In Java, you usually don't want to track the whole program, because this would include bootstrapping the VM, JIT compilation, and so on, which typically we're not interested in. However, with `perf` you can hook into a running process with the `-p` option. `Perf` is a little weird when used with the `-p` option, because you still have to specify a program that will be executed, and the measurement ends once this program ends. Typically you'd use the `sleep` command, which enables you to specify the duration of the test.

First, measure both your loops using the default settings of `perf`.

This provides a good overview. The most-interesting results can be seen in **Table 2**. The number of stalled front-end and back-end cycles differs significantly between both runs. A stalled cycle means the CPU is idle and waiting for something. One of the most likely causes of a stalled front-end cycle is a cache miss, which results in the CPU waiting for data to arrive from main memory or a slower cache. To validate this assumption, you can run `perf` again, but this time to specifically measure cache loads and cache misses. You can see the result in **Table 3**.

The ratio between successful cache loads and cache misses differs tremendously. When accessing the array in sequential order, only about 6 percent of all memory loads result in a cache miss. But when accessing the

array in random order, two out of three loads result in a cache miss. The high number of cache misses is expected, because the array doesn't fit into the cache, and you have to load everything from main memory. But why are there almost no cache misses when you access the array sequentially?

Loading data from main memory into the cache is often a major bottleneck. For this reason, the CPU tries to help by guessing which data you'll use next and loading it into the cache in the background, as you can see in **Figure 9**. While you modify the elements of a cache line, functionality called the *prefetcher* loads the next cache line into the cache. Thus, when you need the data, it's already available in the cache.

The prefetcher is not particularly smart. It can guess the next memory location correctly only if the memory loads follow a regular pattern. In the first case, when you went through the array sequentially, guessing the next memory location was easy, and the prefetcher could mitigate a substantial part of the performance loss by prefetching the next memory location. But when you accessed the array randomly, guessing the next memory location correctly was impossible, and the algorithm had to

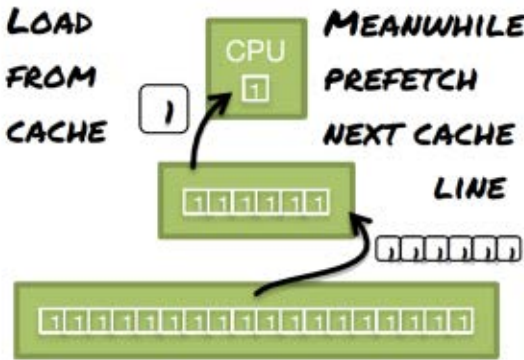


Figure 9

wait until data was loaded from main memory.

The access pattern has a significant influence on the performance of an algorithm, but the chances to use this knowledge within Java are limited. You have close to no control over how your data is arranged in memory. But there is hope. The proposed *value types* might provide this ability one day, because they allow you to arrange object-like structures sequentially in memory.

Another Valuable Tool

Microbenchmarking is another valuable tool for getting more insight into your programs. Probably the most important rule of microbenchmarking is to always use a tool that helps you to avoid some of the many pitfalls, such as the insufficient warmup of the VM, dead code elimination, and loop unrolling. The [Java Microbenchmark Harness \(JMH\)](#) from Oracle is probably the best harness available right now.

	SEQUENTIAL ACCESS		RANDOM ACCESS	
CYCLES	19,430,435,800		19,429,967,708	
STALLED FRONT-END CYCLES	7,217,361,632	(37.14%)	19,006,043,778	(97.82%)
STALLED BACK-END CYCLES	843,462,646	(4.34%)	18,296,349,545	(94.17%)

Table 2

	SEQUENTIAL ACCESS		RANDOM ACCESS	
L1 CACHE LOADS	5,758,001,370		170,655,221	
L1 CACHE MISSES	360,757,378	(6.27%)	365,959,699	(214.44%)

Table 3



ANNOTATION	DESCRIPTION
@BENCHMARKMODE	SPECIFIES WHAT SHOULD BE MEASURED—FOR EXAMPLE, THROUGHPUT OR AVERAGE TIME.
@OUTPUTTIMEUNIT	SPECIFIES THE TIME UNIT USED IN THE OUTPUT—FOR EXAMPLE, TimeUnit.MILLISECONDS.
@WARMUP	SPECIFIES THE WARMUP PHASE. YOU CAN SET THE NUMBER OF ITERATIONS AND THE DURATION OF A SINGLE ITERATION.
@MEASUREMENT	SPECIFIES THE MEASUREMENT PHASE AND IS SIMILAR TO @WARMUP.
@FORK	SPECIFIES HOW OFTEN YOU WANT TO FORK THE JAVA VIRTUAL MACHINE (JVM) AND RUN THE TESTS. YOU SHOULD ALWAYS DO RUNS IN SEVERAL FORKS.

Table 4

Tests written for JMH are similar to JUnit tests. The code you want to benchmark needs to be in a single method, which must be annotated with `@Benchmark`. The test can be configured with annotations at the class level. **Table 4** shows the most-important annotations and their meaning.

Conclusion

Most of the time, processes at the hardware level have no significant effect on programs, but sometimes they do. Therefore, it's useful to have a rough understanding of what goes on at the hardware level and to keep up with the latest developments. Besides being useful, this knowledge is also fascinating and a great way to impress your fellow developers.

This was the first part of a two-part series about the quantum physics of Java. This part

focused on memory and, specifically, the cache hierarchy. The second part looks a little more into memory and then takes a deep dive into the inner workings of a modern CPU. [</article>](#)

MORE ON TOPIC:



- LEARN MORE
- ["What Every Programmer Should Know About Memory"](#)
  - [Igor Ostrovsky Blogging](#)
  - [Martin Thompson's blog, Mechanical Sympathy](#)
  - ["Linux kernel profiling with perf" tutorial](#)

# 3 Billion Devices Run Java

ATMs, Smartcards, POS Terminals, Blu-ray Players, Set Top Boxes, Multifunction Printers, PCs, Servers, Routers, Switches, Parking Meters, Smart Meters, Lottery Systems, Airplane Systems, IoT Gateways, Programmable Logic Controllers, Optical Sensors, Wireless M2M Modules, Access Control Systems, Medical Devices, Building Controls, Automobiles...

