;login:

THE MAGAZINE OF USENIX & SAGE

December 2000 • volume 25 • number 8

inside:

PROGRAMMING:

JAVA PERFORMANCE

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

java performance

Cache-Sensitive Java Programming

by Glen McCluskey

Glen McCluskey is a consultant with 15 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ performance, testing, and technical documentation areas.

<glenm@glenmccl.com>

One of the sayings you may have heard in connection with Java programming is "write once, run anywhere." The idea is that Java programs are compiled into an intermediate bytecode form, and then interpreted or dynamically compiled by the Java Virtual Machine (JVM). A further aspect of this idea is that a Java application is insulated from the underlying operating system and hardware.

This approach is a "good thing" and is at least partially true in practice. For example, it's possible to take a bytecode (.class) file produced by one Java compiler, and use that file with a JVM that comes from somewhere else.

But it's also interesting to explore the limitations of this approach. For example, if you're a Java programmer, do you ever need to worry about the characteristics of particular hardware that your application may run on? In this column, we'll look at a specific example, one that involves cache-sensitive programming.

CPU Caches

As CPUs get ever faster, more attention has been focused on the CPU cache, in particular how to keep the CPU supplied with instructions and data, even though it is running faster than memory. The idea is to store frequently accessed instructions and data in a high-speed cache, instead of having to access memory. For example, suppose your application spends most of its time executing tight loops in a few key functions; it would be desirable for the instructions that make up these loops not to be fetched from memory for each loop iteration.

A specific example of a cache architecture is the Pentium III, which has a 32K level-one cache, and a 256K level-two cache, both of which run at full CPU speed. Instructions and data are accessed from the cache whenever possible, in preference to memory. A typical cache replacement algorithm brings in 32 bytes (a "cache line fill"), aligned on a 32-byte boundary.

An Example

What does knowledge of CPU caches have to do with Java programming? Suppose you've got an application that uses a big array, and you're accessing array elements either sequentially or else in random order. A demo program that models this situation looks like this:

```
import java.util.Random;

public class CacheDemo {
    // size of array
    static final int N = 1*1024*1024;

    // random number generator with fixed seed
    static Random rn = new Random(0);

    // array of bytes to sum up
    static byte vec[] = new byte[N];

    // index giving order of access to vec elements
    static int index[] = new int[N];
```

```
        // fill the array with random bytes
        static void fill() {
            for (int i = 0; i < N; i++) {
                    vec[i] = (byte)(rn.nextInt() & 0x7f);
            index[i] = i;
            }
        }

        // sum the values of the array,
        // accessing elements according to the order in index
        static int getsum() {
            int sum = 0;
            for (int i = 0; i < N; i++)
                sum += vec[index[i]];
            return sum;
        }

        // shuffle the elements of the index
        static void shuffle() {
            for (int i = N - 1; i >= 1; i—) {
                 int j = (int)(rn.nextFloat() * (i + 1));
                int t = index[i];
                index[i] = index[j];
                index[j] = t;
            }
        }

        // driver
        public static void main(String args[]) {
            long start, elapsed1, elapsed2;
            int sum1 = 0, sum2 = 0;

        fill();

        // sum the array elements, accessing elements in sequential order
         start = System.currentTimeMillis();
        for (int i = 1; i <= 25; i++)
            sum1 = getsum();
         elapsed1 = System.currentTimeMillis() - start;

        // sum the array elements, accessing elements in random order
        shuffle();
         start = System.currentTimeMillis();
        for (int i = 1; i <= 25; i++)
            sum2 = getsum();
         elapsed2 = System.currentTimeMillis() - start;

        // sanity check
        if (sum1 != sum2)
            System.err.println("sum1 != sum2");

        // display results
        System.out.println("sequential order = " + elapsed1);
        System.out.println("random order = " + elapsed2);
        }
  }
```

The program creates an array and fills it with random values. Then another array is created to contain indices into the first array. We then do two timings, each of which goes through the array and adds up all of its elements. The first timing accesses the array in sequential order, the second in random order, using the index array to control the access order.

Timing Results

I tried this program on a 300MHz Pentium II, using a couple of different Java compilers, and the results in each case show that accessing array elements in random order is about 2.5 times slower than accessing them in sequential order.

It's possible to create other similar programs in Java or C that show the same behavior. For example, if you take a C array and sum its elements by accessing, say, every 128th element, in a sequence like this:

```
0, 128, 256, ...
1, 129, 257, ...
2, 130, 258, ...
```

you are likely to observe similar timing behavior.

These results are consistent with the way hardware caches work. If you access elements sequentially or access elements that are close together in memory at about the same time, then it's more likely that the elements will be in the cache, because of the way that memory locations are grouped together.

It's possible that you may observe very different behavior. For example, some old microprocessors might not have a cache, or their cache may run relatively slowly. There are also CPUs like the Pentium Xeon with caches as large as 2MB.

Should You Care?

Should you worry about cache-sensitive programming? Most of the time, no. A direct analogy can be made to programming in assembly language versus high-level languages like C. A skilled assembly-language programmer can typically turn out code that is faster than C code, but there's a cost to doing so – the programmer has to manage much more detail, and such programming is time-consuming and error-prone.

The same is true with Java programming. If you spend a lot of time worrying about cache behavior on the target machine, you may end up writing code that's convoluted or tricky to understand. And with Java code, which is supposed to be widely portable, picking a specific target architecture in the first place may be hard to do.

Having said this, understanding how CPU caches work is useful, and sometimes important in specific applications. The trend in hardware seems to be toward placing more emphasis on the cache, suggesting that this area will become more important as time goes by.

Further Reading

Jon Bentley discusses cache-sensitive programming in his *Programming Pearls* (Addison-Wesley, 2000). The Web site for the book is <*http://www.programmingpearls.com*>.