Performance Tuning in Java

The following mantra was first stated about two decades ago in Jon Bentley's "Programming Pearls" column - defer optimization and get your code working first. This wisdom has been amplified by numerous writers on object-oriented design, coding, thinking, and more. The reigning philosophy has been stated as, "get it working first, then determine which areas are the critical ones and optimize only those."

Since 20% of the code is run 80% of the time, this seems like a reasonable idea. Bentley was an early advocate of the use of profiling tools that show which parts of the code are run the most, and targeting the most critical areas first. On the whole, this is good advice, but the lesson has been learned too well. Today it's common practice to ignore efficiency, scattering layers of unnecessary inefficiency everywhere without thought. This article shows that it's just as easy to write faster code without taking extra development time to do it, and teaches something about the way Java optimizes your code.

Method Calls

Object-oriented design is an organizational technique; at the individual method level, code has been written the same way for the past 40 years. The focus on organization merely breaks code down into small manageable units classes that contain a number of (typically quite small) methods. In a language such as C++ this is not a problem at all, as the language itself has extensive control over the cost of the method calls. In fact, with the inline directive, the cost of method execution in C++ can drop to zero. In Java, however, the default method definition checks to see what object it is and calls the appropriate method, which is equivalent to a virtual function in C++. This involves overhead: the program must first examine the object to determine its type, select the appropriate method, and then call it. Calling a method is quite slow compared to executing instructions within a method. In fact, as my benchmarks show, a loop that executes n times doing nothing but counting is 50 times faster than one that calls a method that does nothing. (See the benchmark on my Web site, www.righttrak.com/javaperformance/benchmarks.)

What is the cost of a method call and how can you reduce it? A static method costs about three units of time on my Pentium 4 PC, where a unit is defined by an empty counting loop. A final method costs roughly the same, and an ordinary nonfinal method is about three times as expensive. The conclusion is obvious: whenever possible, use the final or static qualifiers on methods (in other words, if you don't intend to override the method, say so).

Let's start by saying that if you want a program to run fast, get JDK 1.4 and run it with optimization turned on:

```
java -server MyClass
```

The -server option scans the entire loaded program as it's being run, eliminating methods by inlining them, turning methods into native assemblers, removing constant evaluations from loops, and other optimizations. It improves performance, often by a factor of 10 in CPU-intensive bits of code. It might surprise you to think about optimizing programs at runtime, but considering that Java runs on different machines, the only way to optimize for your particular processor is at runtime.

This feature is new in 1.4. There's a "bad feature" in 1.3 that tends to invoke the JIT compiler lazily, often too late. If you compile the following program and run it under 1.3, any code in main is optimized, but code in f() is optimized only after it's called once:

```
public static void f() { ... }
public static void main(String args[]) {
f();
}
```

In this case, since f() is called only once, it's obvious that the compiler had better optimize f before executing it, or not bother.
The compiler makes certain assumptions about what is worth inlining based on the fact that inlining code can take more memory if the code is big. If you have a big routine, it won't be inlined. This can be very unfortunate in some specialized cases, which is where human intelligence comes in.

Suppose you're writing f(), which calls g(), which calls h(). You're doing this just to break up the code and make it easier to read. However, even though f is the only function that calls g(), if g is big enough it won't be inlined. It doesn't matter if there's a big loop involved:

```
void f() { g(); }

void g() {
for (int i = 0; i < 100000000; i++)
...;
}
```

because the time it takes to perform the loop dwarfs the time it takes to get in and out, and so the percentage cost of the procedure call is tiny.
There is, however, a case to be made for applying human intelligence to inlining code. Frequently a method is large, but the first line of the method is a test that determines whether or not to execute the rest. Consider a logging routine. If debug mode is on, it should write out. But for all those cases where debug is false, why call the routine at all? Here's an example of this in action:

```
private static boolean debug;
```

In the first case, f7 calls the logging routine regardless of the state of the debugging flag:

```
public static int f7(int n) {
int sum = 0;
for (int i = 0; i < n; i++) {
log(i);
}
return sum;
}
```

In the second case, f8 calls the logging routine only if debug mode is on. At the cost of an extra statement every time you call the log routine, this code runs 25% faster.

```
public static int f8(int n) {
int sum = 0;
for (int i = 0; i < n; i++) {
if (debug)
log2(i);
}
return sum;
}
```

Multithreading and Synchronization
Since Java is a multithreaded language, the synchronized primitive is provided to make sure that multiple threads of execution do not destroy objects. When entering a synchronized method, it acquires a lock that's associated with the object and prevents any other synchronized method from entering. Acquiring such a lock is a slow machine language instruction. The result is that calling a synchronized method is three times as slow as an ordinary method, which in turn is three times as slow as a static or final method. The computer must first check whether someone else already has the lock, and if not, acquire the lock all in one atomic operation.

```
public synchronized void f() { ... }
```

Multithreading is a complex topic, and the reader is well advised to read one of the many books on the topic for a full understanding. However, a very quick overview of optimization should begin with the observation that since acquiring locks and managing them is a costly business, we should avoid it unless there is some real reason for their use. Furthermore, multithreading will only result in a gain of efficiency if we can overcome the immediate loss of efficiency that results from calling synchronized methods. In a great many situations, multithreading is not called for, and the simplest and best way to handle the situation is to say that the object is not thread-safe, and that programmers should never make two simultaneous calls to the same object.

On the other hand, if multithreading has a significant advantage, the best way to achieve it, where possible, is to have more than one object and give each thread its own object. As perfect examples of this, consider IO streams in a Web environment. While there may be many threads simultaneously writing Web pages, each one is writing to its own Web page. In such cases, the IO stream for servlets could be implemented as a fast, unsynchronized version of PrintStream.

Sometimes, however, there are applications (like a log) where it's vital that multiple threads be able to write to the same object. In such cases, synchronization is vital for correctness. While we can add a new class to the library to support unthread-safe IO, we must always continue to support thread-safe IO for those few cases where it's important.

If you're going to acquire a lock, do so only once. Planning how locks are acquired and released is not only good optimization practice, it's worth really thinking over as this is one of those tricky areas where badly thought-out designs are not only slow, but often don't work in very subtle, nonrepeatable ways. These are the hardest possible situations to debug. Because acquiring the lock means that no one else can enter, synchronized critical sections should:

1. Be as short as possible
2. Not call other synchronized routines (i.e., do whatever needs to be done in a single synchronized section if possible)
3. Never allow unsynchronized access to critical data
4. Never deadlock

Case Study

Simply removing all the synchronization from java.io.PrintWriter and writing a class that is functionally equivalent but not thread safe resulted in a 50% improvement in speed. Class PrintWriter contains synchronized methods that call other synchronized methods, in some cases three deep. The long chains of method invocation before getting to any actual code is a large part of what slows down IO.

Calling Native Methods

You might assume that if you really need speed, you can resort to linking in some C++ code and call that for the ultimate in performance. The answer may surprise you; it certainly surprised me. Even ignoring the obvious disadvantages of using C++ the lack of portability, requiring a shared library to deploy an application, etc. the simple fact is that calling a native method is twice as slow as an ordinary method call.

Having looked a bit at the implementation of the JDK, I can tell you that while it may be tweaked a bit, the reason is essentially sound to call a C++ routine, you must first make a native mode call (that's one) and then set up a call to the underlying C++ routine; twice as much work, twice as much time, right? And to communicate with anything in the Java environment takes further calls as well, so the only way you'll see a significant speed advantage is by staying in the C++ world for a while. In short, native methods seem to be totally outclassed at this point by a combination of increasingly good optimization in the Java world and the somewhat inefficient code involved in the communication between the two.

Creating Objects

As a C++ programmer originally, I assumed that the biggest cost I was likely to find was the synchronized method call. I was surprised the slowest operation by far was the creation of an object. In hindsight it makes perfect sense. Creating an object requires the allocation of memory, including all the overhead for identifying the class of the object, its lock, and the amount of memory being used. After using the object for a time and invoking methods, the garbage collector must eventually free the memory that has been allocated. The act of allocating the memory alone, even when optimized in JDK 1.4, is far more expensive than a synchronized method call. The overriding rule in Java code optimization is simple: don't create unnecessary temporary objects.

In the following example, the first version, which creates only a single object and repeatedly queries it, is 800ms versus 26,300ms, or more than 30 times faster than the second one, which repeatedly re-creates the object. This is an extreme example, of course, because what is being done is very simple compared to the object creation, but it gives an idea of just how costly object creation is.

```
public static int f10(int n) {
int sum = 0;
TempThing t = new TempThing(0);
for (int i = 0; i < n; i++)
sum += t.getV();
return sum;
}

public static int f11(int n) {
int sum = 0;
for (int i = 0; i < n; i++) {
TempThing t = new TempThing(0);
sum += t.getV();
}
return sum;
}
```

Case Study

While removing synchronization and streamlining the code path of PrintWriter resulted in a factor of two improvements in performance, eliminating the temporary string created in printing an int resulted in a sixfold performance improvement.

String Manipulation

Many programmers have seen the sequence:

```
String s = "a" + "b" + "c";
```

and know that StringBuffer is better:

```
StringBuffer b = new StringBuffer();
b.append("a").append("b").append("c");
```

This knowledge seems to break down after this point. If you're processing large strings in StringBuffers, don't then turn them back into strings to pass them to another routine unless you're worried about multithreading problems. As long as you're processing single threaded, you're better off continuing to append into the StringBuffer until you're done. The following routine:

```
public String getAsXML() {
StringBuffer b = new StringBuffer();
b.append(...);
return b.toString();
}
```

must make an unnecessary copy in order to turn the StringBuffer into a string. Then, if the caller is going to append more text, this string must be appended into yet another StringBuffer. This is a big waste. Instead, try:

```
public void getAsXML(StringBuffer buf) {
buf.append(...);
}
```

where the caller allocates the StringBuffer and passes it to the routine, which fills it. The caller can then continue processing. This approach has another advantage, namely that the caller usually has a much better idea of the total size of the StringBuffer at the end of processing. It is vastly more efficient, if you know how many characters are involved, to preallocate them rather than allow the StringBuffer to start at the default size of 16 and grow, which requires a lot of copying. For example, if you know the eventual size of the string will be as high as 2K, then:

```
StringBuffer buf = new StringBuffer(2048);
obj.getAsXML(buf);
```

will typically result in approximately 100% performance improvement over the original string code. It's far better to overallocate than to underallocate and require a grow operation. Remember, this works only if the string in question is not being assaulted by multiple threads.
Manipulating strings, even optimized ones, takes a fair amount of work and code, even if the string length is one. If you're processing a single character, using a char is much faster, so:

```
buf.append('\n');
```

is significantly faster than:

```
buf.append("\n");
```

Efficient Use of Lists
Java provides a fairly rich set of data structures. They're not all the same, and while they may work interchangeably, that doesn't mean they're all equally good in all circumstances. To build up a list in order, ArrayList is faster than LinkedList by a factor of two. LinkedList is substantially slower because each node requires the creation of an object. Vector is a close second in speed; it's slower because it's a synchronized data structure. However, in situations where values are to be inserted in the middle of the list (or worse still, the beginning), LinkedList is the best by orders of magnitude since it does not have to constantly copy elements to move them aside.

```
ArrayList v = new ArrayList(n);
for (int i = 0; i < n; i++)
v.add(new Integer(i));
return v.size();
```

While both Vector and ArrayList use a doubling algorithm that will adaptively grab larger and larger chunks every time the size is exceeded, each time they grow an enormous expense is incurred. As with StringBuffer, it's about twice as fast to preallocate as much space as you'll need than to grow later, even if you overallocate.
Last, remembering that object allocation is the slowest activity of all, you can easily see that this list, which must create object wrappers for each int, is vastly inefficient. The following code, using a list class written just for int elements (see my Web site for the code), runs a full four-and-a-half times faster than ArrayList.

```
IntArrayList a = new IntArrayList(n);
for (int i = 0; i < n; i++)
a.add(i);
```

For scanning through an existing list, ArrayList is the fastest of the JDK list classes; getting an element from an array is a trivial operation, so synchronization dominates the time. Here, LinkedList can be monstrously slow if you use it incorrectly. Since LinkedList is not a random-access data structure, calling get(i) means it must start from element 0 and scan forward until it reaches position i. A loop that scans through the entire list is therefore not an O(n) operation, but O(n2). For a list of 100,000 elements, my

computer performed the ArrayList traversal in 3.25 milliseconds. LinkedList traversal took an astounding 113,657 milliseconds, or 34,971 times slower.

```
LinkedList l = ll;
for (int i = 0; i < n; i++)
l.get(i);
```

The correct way to code traversal through a LinkedList is to use the iterator design pattern:

```
LinkedList l = ll;
for (Iterator i = l.listIterator(); i.hasNext(); )
i.next();
```

The lesson to be learned here is that it pays to understand your data structures well. Just choosing the right data structure for your situation can pay enormous dividends. And using one incorrectly, as in the case of LinkedList traversal, can be very costly.
Last, if you want to store a list of primitives, the best way would be to have classes designed for the purpose, like IntArrayList. No one wants to go to the expense of writing and maintaining all permutations of lists for all the primitive data types; this is one reason Java needs a high-quality template facility like C++. That's a topic for another day, but one that I hope to revisit in a future article. For now, a friend and I are proposing some primitive list classes to add to the Java library, because when you do want a list of primitives, there's no substitute for a decent data structure.

Maps
HashMap is quite a bit faster than the older Hashtable, mostly by virtue of not being synchronized. However, the algorithm used is still less than optimal. To analyze it further, you have to look into HashMap's source code, and know a bit about hashing algorithms. In general, a hash algorithm is fast because it "hashes" the key and turns it directly into the location of the bin where the value is stored, making it an O(1) operation. The problem comes when two different keys happen to hash to the same bin. Statistically, this happens fairly often, and it's the job of the writer of the HashMap to reduce it as much as possible.

Collisions cannot be totally eliminated in the general case, so the design of hash algorithms must allow for them. Therefore, each bin in the HashMap is essentially a linked list for all the keys and values that could hypothetically end up there. This means that every time you add an element to a HashMap, you're once again creating an object that holds the key, the value, and a reference to the next node in case any more values happen to land in the same bin.

Object creation is the most expensive operation possible, so I've tried a different approach and have on my site a couple of experimental classes that perform twice as fast as HashMap (FastHashMap) or four times as fast if your key is an int (FastIntHashMap). They do, however, achieve part of their spectacular speed by not checking the size each time a new element is added, so you must allocate the right size table in advance.

As with all other Java data structures, if you add too many elements to a Hashtable or HashMap, they grow. This is the worst thing you can do, since growing requires painfully reinserting every element. Hashing requires about 2530% more bins than there are elements for efficient operation. Always preallocate what you think is the right size for your Hashtable, be generous, and check at the end to be sure you were right and that the table did not have to grow.

Last, because the hash algorithm for strings looks at every character in the string, avoid hashing large strings if at all possible. The smaller the string, the faster the hash.

Strength Reduction
Turning slow machine language instructions into equivalent but faster ones is traditionally the job of a peephole optimizer in a compiler; the optimizer looks at a window of instructions coming out of the code generator and makes judicious substitutions. In the Java environment there are two stages at which

peephole optimizations can be done. One is during compile time when the source code is turned into JVM code; the other is when the code is run and the JIT turns JVM code into a native assembler. The latter is the approach chosen by Sun, because that way they can optimize code for the particular processor running the code.

Having admitted that most strength reductions are things compilers should do, if your compiler doesn't do them (and Java didn't used to), then it's up to you to do them yourself. In doing so, there are a number of issues: Will the resulting code be as simple as or simpler than the original? Gaining a little speed while losing understandability is not a great bargain. Will the resulting code be faster? Programmers often assume they're optimizing, when in fact they're doing the reverse. The kind of clock cycle counting is certainly better done by a compiler, with knowledge of the target CPU and environment if at all possible. The good news is JDK 1.4 now does some strength reduction. It's up to you to decide how much speed you need now.

First, what not to do. Multiplications by the constant power of 2 are automatically converted to shifts by the computer:

```
x * 2 x << 1
x * 16 x << 4
```

More complex, but not worth it, are multiplications by constants:

```
x * 10 (x << 3) + (x << 1)
```

Divisions are not supported at the moment, but will be soon. If you need the speed right now, the speed of the division itself is five or six times faster.

```
x / 2 x >> 1
```

A much more important strength reduction, and one that the JIT is not likely to detect in the near future, also involves division. Often, programmers want to go around a loop, but do something different every n times. One standard trick is to count and take the counter modulo n, as in the following example:

```
for (int i = 0; i < 100000; i++)
if (i%10 == 9) {
// do something every tenth time
}
```

This is slow; the following is four to five times faster:

```
for (int i = 0, j = 10; i < 100000; i++,j--) {
if (j == 0) {
// do something every tenth time
j = 10; // restart the count
}
```

Similarly, if you have code in a loop like:

```
j = (j + 1) % n;
// j should always end up between 0 and n-1
```

it's much faster to write:

```
if (++j == n)
j = 0;
```

In general, for any positive number *x*, *x* % *n* is equivalent to*x* & (*n*-1) if *n* is a power of two. So*x* % 8 == *x* & 7 as long as *x* is positive. Using the & operator is a lot faster.

Summary

All the performance enhancements in this article have involved the application of simple techniques to make individual sections of code faster. If you learn these tricks and apply them everywhere as a matter

of course, your code can get significantly faster without a lot of effort. These techniques, combined with JDK 1.4 and the next generation of Java compilers, are going to take us within a hair's breadth of being as fast as a well-written C++ application and most applications in C++ are not well written. The world will enjoy the resulting crisp handling of the programs to come. Get out there and write something great.