Common Java Performance Issues and Solutions

by Gordie Freedman, Intuitive Systems Inc.

Intuitive Systems, Inc.
555 N. Mathilda Avenue
Suite 22
Sunnyvale, CA 94086
optimizeit@intuisys.com

Table of Contents

Introduction: Software Development and Performance

Many projects suffer from performance problems, and this brings up a lot of questions that have to be answered in order to fix those problems. This is particularly true when you use development techniques and languages which put elegance, encapsulation, and all those other good design principles ahead of performance goals. In most cases, trying to blindly achieve optimizations in every bit of code you write can be detrimental, to the expense of developing straightforward and maintainable frameworks or applications, and from an economic standpoint. It's important to know what to optimize, where to put your effort, where to spend your development time.

It's also important to consider that many of the techniques that facilitate the development of well encapsulated and elegant code may seem to stand in the way of efficiency. This doesn't mean that clear design principles and good performance are exclusive, but in many cases if you focus on different design aspects and ignore performance considerations, you'll likely end up with performance problems. Achieving a balance, and understanding where it is most important to focus on performance issues, requires a deep understanding of the architecture and the implementation of the software. To do this requires looking inside a program or framework, at how the program is put together, and how the program operates when it is running.

One simple truth in software development is that you need to have the right tools for the right job. In relation to asking the right questions, it is much easier to do so when you have tools that help answer these questions. Debuggers are an obvious example, they answer questions like "what is this code doing", and "how did we get here".. In much the same way, tools for profiling are critical, to answer questions like "what is getting allocated when the program does this", "where is the time being spent in this program", and "why does the program slow down after it has been running for a while". An important thing to note is that besides perceptual problems with performance, the very problems that cause poor performance can also cause a program to stop functioning. If a program runs out of memory and stops working, or if a program runs too slowly to service requests and operate adequately, it becomes useless.

Common performance problems

To illustrate some of these points, we'll discuss three common performance issues: memory leaks, too much temporary allocation, and inefficient algorithms. We'll look at an example of each, discuss techniques to diagnose and pinpoint the causes, and describe how to solve each problem. The expectation here is not exhaustively describe every performance problem you'll ever encounter, but to form a basis for looking at a program with respect to performance issues, and to explore the steps to take in order to deal effectively with performance problems. The examples here will use Optimizeit, a Java profiler from Intuitive Systems, to demonstrate some of the techniques used in diagnosing performance problems.

A side note - during architecture stages early on in development, looking at performance can often be premature, unless you are developing something with hard performance constraints that are key to the operation of the program. This advice might sometimes be misinterpreted, unfortunately, to prevent people from dealing with performance issues as code is developed, after the architecture is for the most part worked out, but before the final stages of development. If you ignore any aspect of software development for too long, be it the architectural design, the documentation of the code, performance, or any of a number of things, you are going to have problems down the road. Balance is a key point, as well as the simple advice to pay attention to what you're working on. And to pay attention to software, you need to understand what that software is doing, what the effects of it's architecture are, and how it operates in the real world.

Memory leaks

Sometimes a framework may behave in unexpected ways, leading to performance issues that aren't obvious because they aren't directly in your code. This doesn't mean that the framework itself is doing anything wrong, but the combination of its behavior and the way the framework is being used could lead to performance problems. One example of this concerns memory leaks. A framework might hold onto a set of Objects until it is explicitly told to release them, or until a specific event or action takes place. If such actions don't occur, the framework may never release memory, causing a leak. A framework might also hold onto a set of Objects due to some expectation it has over intended usage. If the framework is used for a different purpose, this might not be necessary, and in fact may lead to a memory leak.

As an example, Java provides the Object Serialization Framework to read and write Objects using a stream. These can be used between processes communicating with a Socket, and to serialize Objects into a persistent store. This example looks at using the classes java.io.ObjectInputStream and java.io.ObjectOutputStream in order to send Objects between two processes. This is useful for a simple protocol that sends arguments in a request to another process (possibly an Array of Strings or a Hashtable of simple types), and receives a simple response, such as a String. In many cases, a simple protocol is all that is needed between two processes, so something more general like a Distributed Object Model may not be necessary.

It is straightforward to set up an ObjectInputStream and ObjectOutputStream pair on each side of a connection, and send messages bundling copies of Objects this way. It might have surprising results, however, concerning the Objects that are sent and returned in requests. In this example, a Hashtable is constructed for each request, and a unique String is returned. This is innocent enough, however, ObjectStreams are designed to handle the case of sending the same Object multiple times across a connection. This is not just for efficiency, it is also for correctness. If a graph of Objects contains the same Object at multiple nodes, a reconstruction of that graph at another side of a connection should also refer to only one Object at those nodes. The mechanism to do this may also be used to maintain uniqueness for Distributed Object Proxies across a connection, and to prevent infinite recursion in cyclic graphs.

Since our example sends and returns unique Objects, and because every Object sent and returned is held onto by the ObjectStreams, we leak every Object that is used for requests and responses. This leak may not be detected unless the developer is specifically looking at memory profiling information, and it is not obvious from the documentation for ObjectOutputStreams that the Objects will be held onto in this way.

Looking at the memory allocation as the program is running shows the count of Hashtables and Strings growing. Forcing a Garbage Collection does not reduce the number of these Objects. This is different than a problem with temporary allocation, as these Objects will not be freed when the garbage collector runs.

In this case, there is a simple solution to this problem. The ObjectOutputStream class has a reset() method, which will clear the list of Objects written to the Stream. A corresponding InputStream on the other side of a connection will then clear the list of Objects read from the Stream. When ObjectStreams are used in this way, the reset() method should be called periodically to clear the set of saved Objects. This can be done on every request/response, although this imposes some overhead, resulting in additional messages on the network. For optimal performance, using DataStreams (instead of ObjectStreams) and manually reading and writing simple Objects across the stream in terms of Strings or byte arrays will be more efficient.

Techniques used here:

- *Look for Classes whose instances keep growing.*
- *Run the garbage collector manually.*
- *Examine reference graph info to see where objects are allocated.*

Too much temporary allocation

A problem with Strings

Java provides a number of natural constructs for dealing with Strings, making it easy to construct and manipulate Strings. It isn't always apparent, however, how many Objects are being created when working with Strings. As an example, the addition ("+") operator can be used to concatenate Strings, allowing a simple technique to build up Strings on the fly. This seemingly innocuous operation can unfortunately hamper performance in ways that wouldn't be obvious if you didn't know what to look for. We're picking on String because it is a straightforward example of performance problems with temporary allocation, even though a lot of people may already know about these issues. The point here is to demonstrate the process of examining this kind of problem.

In this example, a program has a number of statements like these:

```
Trace.trace(Trace.Informational, "Fetched " + n + "records from database " +
              databaseName + " in name lookup");
```

Through the course of the program, trace records are used to produce a log of the different paths taken to execute certain operations. Notice the first parameter to the trace call, Trace.Informational, which indicates the level of tracing. The second parameter, the String, isn't output unless tracing is set above this level. Also notice how the second parameter is constructed from a number of substrings. Although this costs a method call on each potential trace, the actual output is not always done, so one might hope the trace statement shouldn't cause any performance concern.

Unfortunately, this is not the case, for two reasons. Let's look at memory allocation before and after this statement.

After the trace statement, even when the trace level is below Informational, a number of Strings have been created. This is pretty obvious when you think about it, but when focusing on other aspects of design (such as the functionality of the Trace system) it's easy to miss something like this. When profiling your program, though, you'll see a lot of Strings created. If you turn off the garbage collector so they stick around, and then examine the allocation backtraces, it becomes obvious what the culprit is.

It's important to note that this situation causes two problems here. One is the allocation of extra Objects, which causes expense in memory space and CPU time. A second problem is that these temporary Objects need to be garbage collected, and the overhead for the Garbage collector to scan and collect all these extra objects will slow down the whole program. Additional Objects will cause the collector to spend more time in the mark phase when those Objects are being referenced, and may slow down the program depending on the type of Garbage Collection algorithm being used. Besides the CPU overhead if the garbage collector has to do more work, other threads will not be able to obtain resources or run if the garbage collector locks parts of the system (some garbage collectors lock access to the memory heap, so other threads will not be able to allocate storage while the garbage collector is running) .

For this particular example of temporary String allocation, there are a number of solutions. One is to explicitly check the tracing level before calling the trace routine, and avoid generating the String if tracing will not occur. Instead of the Trace code checking the trace level, the calling code does, to avoid allocating Strings when they aren't needed. Another solution is to avoid concatenating Strings. In certain cases, the String passed to the Trace routine can be allocated once, and reused.

In a similar example, if a String is built up in multiple statements with the String concatenation (+) operator, the compiler will generate code which produces a lot of temporary Strings and StringBuffers. Using a single StringBuffer and calling the append() method on it until it is completed will reduce the amount of temporary Objects created. For example, when the compiler sees the following code, for each line adding to *myString*, it will generate a new StringBuffer from the variable *myString*, call append on it, and then generate a new String to assign to *myString* from the StringBuffer. This creates an additional temporary String and StringBuffer for each concatenation operation. In this example, five StringBuffers and five Strings are created (note two additional Strings are generated to convert *count* and *resultCount* to Strings).

```
String myString = "You submitted ";
myString = myString + count;
myString = myString + " requests";
// … processing
myString = myString + ", results obtained from ";
myString = myString + resultCount;
myString = myString + " requests";
System.out.println(myString);
```

Replacing this code with a StringBuffer will be more efficient, allocating only one StringBuffer and one String (and the two Strings for *count* and *resultCount*).

```
StringBuffer myStringBuffer = new StringBuffer("You submitted");
myStringBuffer.append(count);
myStringBuffer.append(" requests");
// … processing
myStringBuffer.append(", results obtained from ");
 myStringBuffer.append (resultCount) ;
myStringBuffer.append(" requests");
System.out.println(myString);
```

While the examples here are for tracing, there might be similar cases where Strings are concatenated together in performance sensitive code that is always executed. This might be in an Enterprise Server that constructs a String based request on the fly (such as SQL or HTTP), where both the time to construct the request, and limiting the amount of temporary Objects created are critically important. In these cases, avoiding the allocation of temporary Objects can be accomplished with these techniques, and will allow the program to run more efficiently.

A problem with Graphics

When writing a GUI Application, some simple techniques that appear harmless may have serious performance implications. As an example, a simple paint routine can allocate a lot of Objects, slowing down the program and the garbage collector. The Paint routine for a simple list Object could be implemented like this:

```
public void paint (Graphics g) {
   int firstRow = convertYToRow(g.getClipBounds().y);
   int lastRow = convertYToRow(g.getClipBounds().y + g.getClipBounds().height - 1);
   int i;
   Rectangle r;

   for (i = firstRow; i <= lastRow; i++) {
      r = getBounds();
      r.x = 0;
      // returns a new point containing the row origin
      r.y = getRowOrigin(i).y;
      // returns a new Dimension containing the row dimension
      r.height = getRowSize(i).height;
      paintRow(i, r);
   }
}
```

If this method needs to paint five rows it allocates eighteen Objects: eight Rectangles, five Points, and five Dimensions. If the number of rows increases, the number of allocated Objects increases as well. Implementing a list's Paint method this way will keep the garbage collector extremely busy. If this list is added to a scrollpane, scrolling will pause when the garbage collector needs to run. Temporary Objects can often be eliminated, allowing the routine to run faster and speeding up overall performance by reducing memory management overhead.

A more efficient implementation could be written like this:

```
public void paint (Graphics g) {
   Rectangle clip = g.getClipBounds(); // Cache clip rectangle for next operations
   int firstRow = convertYToRow(clip.y);
   int lastRow = convertYToRow(clip.y + clip.height - 1);
   int i;
   Rectangle r = clip; // Reuse the clip rectangle since we don't need it anymore

   for (i = firstRow; i <= lastRow; i++) {
      r.x = 0;
      r.y = getRowY(i);
      r.width = getWidth();
      r.height = getRowHeight(i);
      paintRow(i, r);
   }
}
```

Note that the rewritten method allocates one temporary Object regardless of the number of painted rows. The rectangle is allocated by the AWT getClipBounds() method. Note this method always returns a new rectangle, although this fact is not obvious from the documentation.

In this example, temporary Objects are reduced by reusing the same objects when possible, and by designing internal methods so they don't have to create new objects, using scalars and input/output

parameters.

While these examples are fairly simple, problems with temporary allocation are fairly common, and the techniques used to diagnose this example will be valuable in other situations. This example also serves to demonstrate how frameworks may allocate a lot of temporary Objects, without documenting where and how they are allocated. Without a profiler, it is almost impossible to diagnose and avoid large amounts of temporary objects created in this way.

Techniques used here:

- *Turn off the garbage collector to see temporary allocations.*
- *Run the garbage collector to mark currently allocated objects, see what is created for specific operations.*
- *Examine allocation backtrace info to see where objects are allocated.*
- *Use a profiler to understand which methods in existing libraries are allocating objects and where these methods are called in your program.*

Using too much CPU time

Memory problems aren't the only performance issues a program may have. There may be places where CPU usage becomes problematic, leading to slow execution. For some parts of the code, such as parsing a command from the user, this might not be critical. For other parts, such as frequently exercised code in a server to look up data, CPU performance might be extremely important. For user programs, or programs started on a server in response to requests, startup time can be an issue as well.

In this example, a large data set of customers is maintained, and queries can be made returning a list of customers. The list can be sorted based on different criteria, such as the customer's last name. The initial algorithm used to do the sort is a simple Sort. Because the actual sort represents a small piece of code contained in a much larger subsystem, the developer's initial focus may not be on the performance of the sort. It is fairly common to see linear lists and simple sorts, particularly when data sets are small, where they work perfectly fine.

Testing customer queries with 100 entries doesn't demonstrate any performance problems, requests are serviced quickly from the user's perspective. Once the system is deployed, however, queries may return in excess of thousands of customers in the list. At this point, queries can start taking an additional few seconds, and for very large data sets, could take up to a minute or more. For a user sitting in front of the screen, this performance is not acceptable. Furthermore, as the number of customer's increases, the time to service the request takes longer and longer.

Looking at the CPU profiling information for the request, we can see that the program spends a lot of time in the String.CompareTo method. This isn't actually the culprit, it is the caller of this method. We can see that this is the sort method. This method does a simple sort of an Array of Strings. For small Arrays, this is a reasonable technique which is perfectly acceptable. For a large data set, however, because this algorithm is order N2, the time to do the sort is prohibitive. If the fetch of the data takes ½ second, spending a few hundred milliseconds doing the sort doesn't add much overhead. Once the sorting time is over a second, the time to do the sort is taking longer than the rest of the fetch request.

Changing the sort routine to a QuickSort reduces the time significantly, bringing it back into an acceptable time. Although this example is somewhat contrived, it is surprising how much code uses linear searching and sorting techniques, so this type of problem can easily crop up years after software is deployed. Software used by a small company in its first year of operation with 100 customers may become unacceptable in it's fifth year when the number of customers is 100,000. Since the increase in the time to service requests is non linear (increasing more or less by the square of the number of customers), users of the system won't notice the problem until the customer base has grown for a few years. At this time, it is unlikely someone might zero in on a linear sort routine in the midst of what could be tens of thousands of lines of Enterprise software. Using a profiler to examine the request will be able to pinpoint the cause right away.

An important point related to sorting is that the state of the data to be sorted will play a role in the performance of the system. Comparing a simple Sort to QuickSort, with large data sets QuickSort will be hundreds of times faster for data sets in the 10,000 item range. For small sets of data, such as 10 items, the simple Sort can actually be faster, due to the overhead associated with the recursive calls in QuickSort, and the complexity of the algorithm compared to a simple Sort. In this case, if a program does thousands of sorts of small sets of data (between 2 and 10 items), Quick Sort can actually slow down the program. As well, if the data to be sorted is already mostly sorted, a simple Sort can be 5 to 10 times as fast. This situation might not occur too often, but it is useful to know.

Techniques used here:

- *Run a CPU Profiler during the course of an operation.*
- *Look at overall CPU time.*
- *Look at hot spot methods*

Conclusion

Java is a powerful language with a lot of industry support and momentum, and thus is a good choice to use for the development of many types of programs. Because of the way Java programs execute, developers will often need to pay special consideration to performance issues in the programs and frameworks they write. Having the right tools to examine Java code to determine what performance problems exist, and what is causing those problems, is indispensable.

While focusing too much on performance early on in development may lead to bad design decisions, ignoring performance issues will also often lead to bad design decisions. It can be pretty hard to rearchitect a large base of code in order to fix performance problems, so if you have some indication along the way of what the program is doing, you can balance clean architectural design with decent performance. Spending the time with a good profiling tool, such as OptimizeIt, will provide huge benefits for Java developers, as well as the users of their code. Maintaining a balanced approach with a clear architecture and pragmatic implementation does not have to be difficult if you keep both of these goals in mind as you develop your code.