Interpreting Java Program Runtimes

Stuart Hansen
Department of Computer Science
University of Wisconsin - Parkside
Kenosha, WI 53144
(262) 595 - 3395
hansen@cs.uwp.edu

ABSTRACT
Many instructors use program runtimes to illustrate and reinforce algorithm complexity concepts. Hardware, operating system and compilers have historically influenced runtime results, but generally not to the extent of making the data difficult to interpret. The Java virtual machine adds an additional layer of software, making it much harder to obtain easy to interpret results. This paper presents some of the basic issues the author and his students have encountered when analyzing Java program runtimes and briefly discusses strategies to address them.

Categories and Subject Descriptors
C.4 [**Performance of Systems**]: Measurement Techniques, Performance

General Terms
Algorithms, Performance, Experimentation.

Keywords
Java virtual machine, Program runtime, Garbage Collection,

1. INTRODUCTION
The author recently taught Data Structures and Algorithms for the first time since his department's curriculum was converted from C++ to Java. Like many instructors, he wished to demonstrate how the complexity of an algorithm directly impacts its implementation's runtime. The importance of this type of activity is well recognized by computer science educators [2, 7]. During the semester, the author and his students collected runtime data from a variety of programs, but found themselves repeatedly confused by the data because of influences from the virtual machine. Figures 1 through 3 show runtimes for sample programs. The data presented in these figures were obtained using Sun's SDK 1.4.1 running under Debian Linux, Version 3.0 on a 2.4 Gigahertz machine with 512 Megabytes of memory. Times were measured using the method `System.currentTimeMillis()`. Similar results may be obtained using other hardware and software configurations.

Figure 1 shows the runtimes for repeated calls to `Arrays.sort()` with random `int` arrays of size 20,000. `Arrays.sort` uses a version of Quicksort, which is an unstable algorithm. In the long run we expect to see some fluctuation in runtimes. This is not the cause of the problem here, however, as repeated runs of our program with different random arrays produce very similar results. The virtual machine is causing the method to run more slowly on the first call than on subsequent calls.

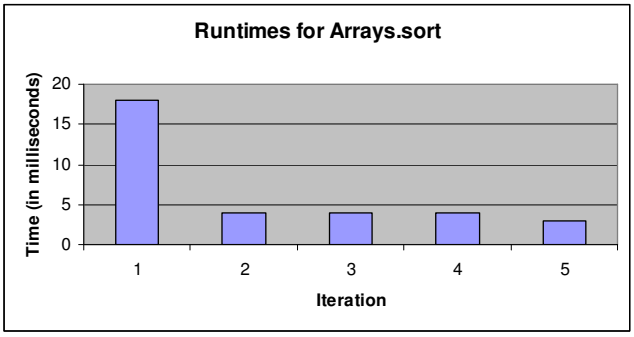

Figure 1. Runtimes of `Arrays.sort` called five times in succession on random `int` arrays of size 20,000. The first run took significantly longer than any of the others.

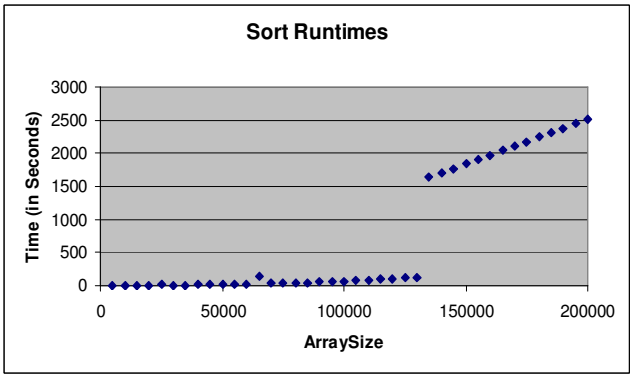

Figure 2. Runtimes obtained for a $O(n^2)$ sort program.

Figure 2 shows the runtimes for a Java program implementing a $O(n^2)$ sort algorithm. The sort is not one of the standard $O(n^2)$ sorts, however. The author had recently introduced his students to the fact that instantiating a Java array is a O(n) operation, since

all cells in the array are initialized. To reinforce this idea, he modified the merge() method from a MergeSort program to use a temporary array the size of the entire array being sorted. This one line change makes the MergeSort program $O(n^2)$. The author was expecting the graph to be roughly parabolic. Instead, it appears to have near linear behavior, with a major discontinuity occurring between 130,000 and 135,000.

Figure 3 shows runtimes for rehashing a Java Hashtable. This experiment was suggested by Michael Clancy during a panel discussion at SIGCSE 2002 [7]. A Hashtable of size 1,000,000 was created and a varying number of Integers was added to it. Since the size of the Hashtable is fixed, rehashing is linear in the number of elements added. The author was expecting the graph in Figure 3 to be roughly a single straight line. Instead, the graph consists of two line segments, with the runtimes for larger data sets being significantly faster than runtimes for smaller data sets. Somehow between 120,000 and 130,000 the program speeds up by over a factor of five.

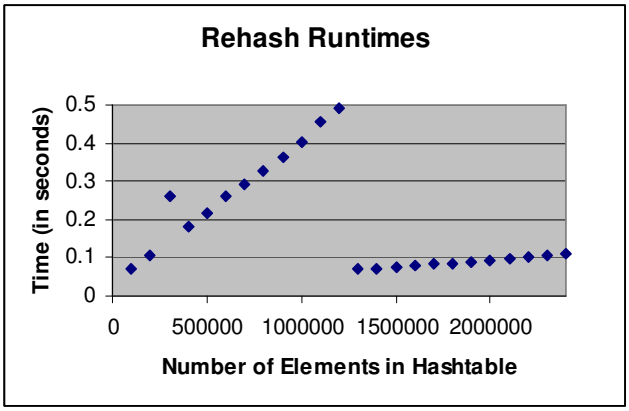

Figure 3. Runtimes obtained when rehashing a hashtable of size 1,000,000.

These examples illustrate that interpreting Java program runtimes is difficult. The goal of this paper is to explain the features of the Java virtual machine that are major influences on program runtimes and show how they may be explained and at least partially controlled to obtain runtimes that reflect the complexity of the underlying algorithm. What we want is to be able to time a $O(n^2)$ algorithm and plot the results, obtaining something akin to a parabola.

This goal is very different than performance tuning. Performance tuning looks at the virtual machine's features with the goal of making a program run faster. There are many texts and papers that address Java performance tuning [1, 6, 8, 9, 10]. Sun's Hotspot virtual machine even tries to tune performance of the program dynamically. This paper does not discuss how to make Java programs run faster. In fact, we will sometimes accept a hit in overall performance if we can thereby obtain runtimes more consistent with the underlying algorithm.

2. JAVA VIRTUAL MACHINES

Most CS instructors are aware of how a computer's hardware and system software influence the runtime of a program. The CPU speed, the amount of main memory, the amount of cache, the operating system and the compiler all play significant roles. We take these into account when designing experiments to illustrate algorithm complexity. For example, if we are working with data structures, we limit their size so that they will fit into physical memory, because we know that as soon as the program starts swapping pages to virtual memory it will slow down significantly. In general, however, it is not difficult to obtain runtimes that are consistent with the underlying algorithm.

Java virtual machines impose another level of software between the program and the operating system. There are many different Java virtual machines available. This paper only discusses Sun's Hotspot virtual machine (JVM). As Java and the JVM have matured, a number of issues have arisen that make designing experiments and interpreting empirical data more difficult. Among the issues are:

1) Starting a Java application incurs a sizable cost. Processes and threads must be created and the JVM must be initialized. Library files must be opened and a fairly large set of classes must be loaded. Only then is programmer code invoked.

2) The JVM automates heap management. The heap is the area of memory where all objects are allocated. Heap management consists of controlling the size of the heap and removing objects from the heap that are no longer in use. All Java instructors know that programmers no longer need to explicitly free memory resources, as the garbage collector takes care of it for us. We also realize that garbage collection takes time. It is important that we understand how garbage collection works and the effect it can have on a program's runtime.

3) The JVM interprets byte code but also compiles byte code into native code for better performance. There are tradeoffs as compiling takes time, but the resulting native code may run significantly faster.

4) The JVM dynamically optimizes running programs. The Hotspot virtual machine takes its name from its ability to analyze a running program, find the bottlenecks, or hotspots, in the program and apply optimizations on the fly.

3. PROGRAM STARTUP

The JVM is resource intensive at program startup. As an example, consider the data presented in Figure 1. It shows the runtimes obtained when invoking `Arrays.sort()` five times in succession on random `int` arrays of size 20,000. The first invocation takes 18 milliseconds, which is more than the next four invocations combined.

Figure 1 only presents part of the picture, however. There are startup costs that are not captured by `System.currentTimeMillis()`. The operating system creates one or more processes for the virtual machine. The JVM is loaded and initialized. Standard libraries must be located and opened. Since these actions occur before the first call to `System.currentTimeMillis()`, they affect the measured elapsed time for running a Java program, but do not influence our recorded runtime and are not discussed further here.

Once the JVM is started, running an application requires loading and initializing a collection of classes. The simplest possible Java application, one with a completely empty main method, loads 280

classes during execution. Loading classes into the JVM is generally a disk-bound activity which is slow compared to CPU activities.

The high cost of the first call to `Arrays.sort()` in Figure 1 is due to loading the `Arrays` class. On the system where the data was collected, the Java libraries are installed on a file server, which slowed the class loader even further.

Frequently, when we are measuring program runtimes, we are interested in taking a series of measurements. It is best to warm up the virtual machine before we start taking collecting data. As a rule of thumb, call each method to be timed at least twice before beginning to record times.

4. HEAP MANAGEMENT

All objects in a Java program live in the JVM's heap. When an object is instantiated, memory is allocated for it in the heap. Sometime after the program finishes using an object, the memory is returned to the free portion of the heap for reallocation. Because Java is object-oriented, many objects are created during the run of even a simple Java program, making heap management important. The two most important issues related to heap management are garbage collection and heap size.

4.1 The JVM's Garbage Collector

When an object is no longer referenced by a program it becomes garbage. Unlike older programming languages, e.g. Pascal, C and C++, it is no longer the programmer's responsibility to return garbage to the heap for reuse. Instead, the JVM's garbage collector takes care of it automatically. The garbage collector finalizes each object and then returns its memory to the heap [3, 4, 5]. There are many different garbage collection algorithms, each with various strengths and weaknesses [5]. The Java Virtual Machine Specification does not indicate which algorithm(s) should be used. It is left to the virtual machine implementer to choose [6]. Sun's JVM can use different algorithms than IBM's Jikes, but both still implement correct Java virtual machines.

4.1.1 Generational Garbage Collection

Sun made its decisions about garbage collecting algorithms based on program observation. Programs allocate many objects that have very short lifetimes. These objects are instantiated, used and discarded in rapid succession. For example, the extent of an object that is local to a method is just the single invocation of that method. If the method is called again, a new object is created for the next invocation. Java programs also have many objects with much longer lifetimes. In fact, if an object survives for more than a short time, it has a high probability of having a very long lifetime. Sun organized the JVM's heap around these different categories of objects, using different garbage collection algorithms for each of them.

The JVM uses a hierarchical garbage collector. It separates the heap into Young and Tenured generations. New objects are allocated in the Young generation. Since there are many objects with very short lifetimes, the Young generation fills up quickly. Many of the objects also become garbage quickly. Minor garbage collections clean up the Young generation using an algorithm that is very fast when most of the space is garbage. If an object survives several minor garbage collections, it is moved to the Tenured generation.

The Tenured generation is garbage collected only when most of its space is used. These full garbage collections use a mark and compact algorithm that runs efficiently when there are still many objects in use. A full garbage collection still takes significantly longer to run than a minor garbage collection. It may be slowed down even further by sometimes doing extra work, like allocating more memory for the heap, if it is still too full after completing the collection.

Garbage collections take place frequently. They are generally transparent to the user, however. If an application is started with the `-verbose:gc` option information about each garbage collection is printed as it occurs. The garbage collector runs in a separate thread from the application program. It does influence the time reported by `System.currentTimeMillis()`, however, because `System.currentTimeMillis()` records elapsed time, not just time dedicated to the application.

4.1.2 Incremental Garbage Collection

The JVM contains an alternative garbage collection algorithm named incremental garbage collection. Incremental garbage collection is specified on the Java command using the `-Xincgc` option. Incremental garbage collection replaces the full garbage collection with a series of smaller steps run more frequently. Since full garbage collection, which can be very time consuming, never takes place, program runtimes are often much more consistent when using incremental garbage collection. On the other hand, incremental garbage collection can cause a performance hit to the overall runtime of the program. Sun included incremental garbage collction primarily for programs with hard deadlines. These programs cannot afford to wait for a full garbage collection to complete, but can wait for the more frequent, but faster incremental garbage collections.

4.2 Heap Size

The amount of memory dedicated to the heap is not fixed. If the heap is still relatively full after a garbage collection, more memory will be allocated to it, up to a specified maximum. The heap's initial size may be set explicitly using the `-Xms` option. The default maximum heap size is usually set so that the entire heap will still reside in main memory. The heap's maximum size may be set using the `-Xmx` option. There are no hard and fast rules for specifying the heap parameters. Specifying a large enough initial heap size can prevent the need for garbage collections in programs with small memory footprints and can delay the need for major garbage collections in others. On the other hand, large heaps take more time for garbage collection when it does occur.

4.3 The Modified MergeSort Explained

The modified MergeSort example shown in Figure 2 illustrates anomalies associated with the heap.

4.3.1 The MergeSort Data

As you recall, Figure 2 charts the runtimes for a modified MergeSort that is a $O(n^2)$ sort algorithm when implemented in Java. There are three anomalies that need explanation:

1) Why isn't the graph parabolic?

2) What is the blip at 65,000?

3) Finally, what is the discontinuity between 130,000 and 135,000?

In fact, the left portion of the data is close to parabolic. It is just that the scale of the y-axis is so perturbed that the parabola is hard to see. Figure 4 re-graphs the data from Figure 2, up to an array of size 130,000. The graph appears roughly parabolic.

There are two outlying blips in Figure 4, one at 25,000 and one at 65,000. These blips are caused by full garbage collections running very frequently. After the heap expands, full garbage collections again run less frequently and the runtimes go back down.

The discontinuity between 130,000 and 135,000 is also a garbage collection problem. Using the `-verbose:gc` option shows that there is a full garbage collection following each minor garbage collection at 135,000. The Tenured generation contains enough objects that full garbage collections take place very frequently and the entire program becomes garbage collection bound. The linux `ps` command shows the system is spending 97% of its time doing garbage collection and very little time actually sorting data.

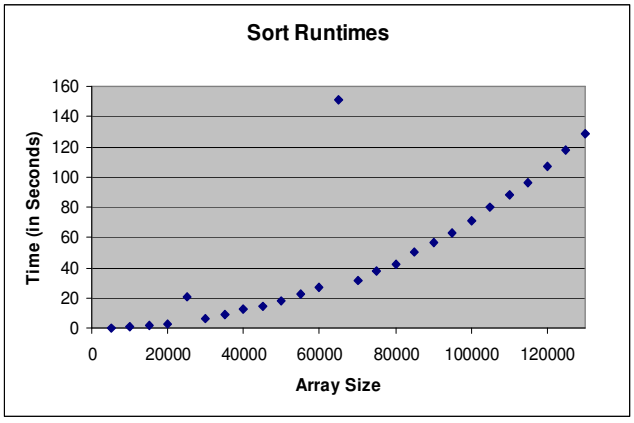

Figure 4. The initial portion of the sort data is close to a parabola.

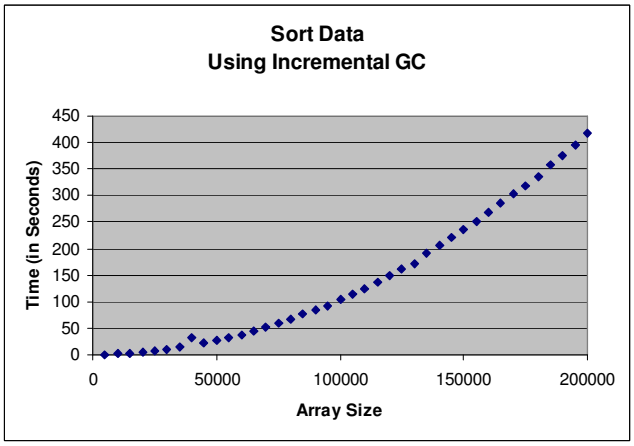

Figure 5. The sort program runtimes when using incremental garbage collection.

Figure 5 shows runtimes from the MergeSort program running with the incremental garbage collector. Recall that the incremental garbage collector is used in place of full garbage collections. Full garbage collections never take place, so the blips disappear and the full garbage collection bottleneck is gone.

5. DYNAMIC OPTIMIZATION

Sun's Hotspot JVM introduced several improvements that can also impact runtime performance.

5.1 Dynamic Compilation

The Hotspot JVM executes both byte code and native code. It takes responsibility for compiling byte code to native code when it feels the performance to be gained is worth the time spent compiling. Most methods are not compiled the first time they are called. This is because many methods are only called once, and compiling such methods would slow down the runtime considerably. While there are several different rules that govern when a method is compiled, the default behavior is to compile a method when it has been called 1500 times.

5.2 Dynamic Optimizations

The Hotspot JVM also applies dynamic optimizations to code that appears to be a bottleneck, or hotspot. Sun claims that this approach shows major advantages over static code optimizers of the type that come with languages like C++. The idea is that the virtual machine will find hotspots in the program and apply optimizations only to those areas. For example, a primary optimization is method inlining, where a method invocation is replaced by the body of the method. Polymorphism makes this very difficult to do at compile time, as any of a number of methods might be called at each invocation. At runtime, however, the JVM has additional information that may let it know which method will be invoked.

Dynamic compilation and dynamic optimizations can play important roles in improving the performance of programs with long runtimes. We have encountered them infrequently in student programs. Most of the time, programmer supplied code is compiled early enough that the runtimes collected reflect executing native code. Occasionally there will be an unexplained blip in runtimes that can be attributed to dynamic optimizations, but the JVM does not give us an easy way to capture when this is occurring.

5.3 Rehashing Explained

The rehashing experiment created and populated a `Hashtable` with `Integer`s and timed how long it took to rehash. The runtimes for various numbers of `Integer`s were shown in Figure 3. There is one major anomaly in the data. When the number of elements in the `Hashtable` grows large enough, the program actually speeds up. Both segments of the graph illustrate close to linear behavior, but the program runs faster and the slope of the line is much less steep for the larger data sets. This is completely due to dynamic optimizations. The JVM inlined method calls and applied other optimizations when approximately 90% of its time was taken up by rehashing. The improvement in performance is dramatic.

Rehashing is a memory intensive process. While the rehashing is taking place there are two hash tables in existence, the old one from which data is being removed and the new one to which data is being inserted. In this example, full garbage collections were frequently taking place when the heap was small, because more space was needed. As the program runs, the heap grows larger, making garbage collection less frequent and increasing the time spent on rehashing. We can obtain improved performance more quickly by starting with a larger heap. Figure 6 shows runtimes for the same program as Figure 3, but with the initial heap size set to 500 Megabytes. Incremental garbage collection was not used. The graph displays the linear behavior consistent with the right hand segment in Figure 3.

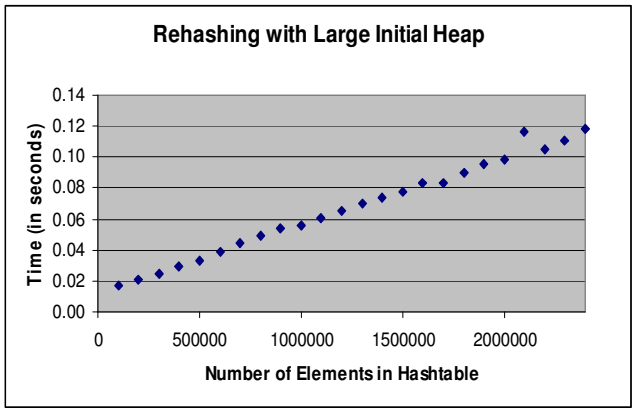

Figure 6. Runtimes obtained by rehashing program when using a large initial heap.

6. CONCLUSIONS

In this paper we have explained three major causes of runtime anomalies in Java programs: program startup consideration, heap management and dynamic optimizations. Our approach is still ad hoc, however. Each time we encounter runtimes in a student program that don't reflect the underlying algorithm, we start from scratch looking for possible causes. Our long term goal is to develop a clear, simple set of instructions for our students so that they can perform runtime experiments on their own and obtain meaningful results.

Source code for all examples presented in this paper is available from the author.

7. REFERENCES

[1] Armstrong, E., *HotSpot: A New Breed of Virtual Machine*, Java World, March 1998. Available on-line at http://www.javaworld.com/javaworld/jw-03-1998/jw-03-hotspot_p.html.

[2] Braught, G., Miller, C., and Reed, D., *Core Empirical Concepts and Skills for Computer Science,* Proceedings of the Thirty-Fifth SIGCSE Technical Symposim on Computer Science Education, Norfolk, VA, March 3-7, 2004.

[3] Goetz, B., *Java Theory and Practice: A Brief History of Garbage Collection*, DeveloperWorks, IBM, October 2003. Available on-line at http://www-106.ibm.com/developerworks/java/library/j-jtp10283.

[4] Goetz, B., *Java Theory and Practice: Garbage Collection in the 1.4.1 JVM*, DeveloperWorks, IBM, November 2003, Available on-line at http://www-106.ibm.com/devloperworks/java/library/j-jtp11253.

[5] Jones, R., Lins, R., *Garbage Collection: Algorithms for Automatic dynamic Memory Management,* John Wiley and Sons, New York, NY, 1996.

[6] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification, Second Edition,* Addison-Wesley, Boston, MA, 2003.

[7] Reed, D., moderator, *Integrating Empirical Methods into Computer Science,* Proceedings of the Thirty-Third SIGCSE Technical Symposium on Computer Science Education, Covington, KY, February 27-March 3, 2002.

[8] Shirazi, J., *Java Performance Tuning,* O'Reilly and Associates, Sebastopol, CA, 2000.

[9] Venners, B., *The Hotspot Virtual Machine: How Hotspot Can Improve Java Program Performance and Designs*, Developer.com, May 1998. Available on-line at http://www.artima.com/desingtechniques/hotspotP.html.

[10] Wilson, S. and Kesselman, J., *Java Platform Performance: Strategies and Tactics,* Addison-Wesley, Boston, MA, 2000.