Olympiads in Informatics, 2011, Vol. 5, 103–112

Measuring the Startup Time of a Java Virtual Machine

Bruce MERRY[1], Carl HULTQUIST[2]
[1]*ARM Ltd.*
110 Fulbourn Road, Cambridge, CB1 9NJ, England
[2]*D.E. Shaw & Co.(U.K) Ltd.*
55 Baker Street, London, W1U 8EW, England
e-mail: bmerry@gmail.com, chultquist@gmail.com

Abstract. For many olympiad problems, an execution time limit is used to constrain the classes of algorithms that will be accepted. While Just In Time (JIT) technology allows Java bytecode to be executed at a speed comparable to native code, the Sun Java Virtual Machine has a reputation for a large startup time that can significantly affect measured execution time. We present a technique for measuring the startup time of the virtual machine for each execution run, so that it may be subtracted from the total execution time. We found that the startup time was lower than expected (about 70ms), with little variation between programs or runs.

Key words: Java, startup time, time limit.

1. Introduction

Numerous programming contests allow Java to be used as a programming language (ICPC, 2010; Kolstad, 2009; Merry *et al.*, 2008; Tani and Moriya, 2008; Trefz, 2007). Additionally, it is common practise in programming contests to impose an execution time limit on solutions, both to protect the system against long-running solutions and to enforce some degree of algorithm efficiency on solutions. When multiple programming languages are made available, the question that naturally arises is whether the choice of language affects execution time and if so, whether this is fair to contestants.

Java is different from C-like languages in that it is typically compiled to bytecode and run in an interpreter, rather than compiled directly to machine code. The de-facto standard interpreter is the Sun Java Virtual Machine (JVM). While the execution speed of such interpreted code remains a concern (even with technologies such as Just-in-Time (JIT) compilation), this paper specifically addresses startup overhead. In evaluating Java for the International Olympiad in Informatics, it has previously been noted (Pohl, 2006) that the JVM can take anywhere up to a second to start, and that the startup time is highly variable.

To address this, the South African Computer Olympiad (SACO) grading system measures the startup time each time a Java solution is run, and subtracts this from the total execution time. Section 2 describes how this is achieved.

In order to evaluate the effectiveness of our wrapper, we are interested in the following questions:

- What is the average startup time of a C++ program?
- What is the average startup time of a Java program?
- Does the Java wrapper have any impact on execution time?
- Is the startup time of a Java program greater than that of a C++ program?
- Is the solution time of an empty Java program greater than that of a C++ program?
- Does the choice of program (e.g., a simple versus a complex one) impact the startup time?
- Does the solution time computed using our wrapper have less variation than execution time? In other words, does our wrapper lead to more consistent timing?

Section 3 explains how we tested the effectiveness of our technique, and we report the results of our tests in Section 4. We finish with conclusions in Section 5.

2. Measuring Startup Time

A simple approach to measuring startup time would simply be to run a trivial program multiple times to determine an average. However, the startup time could vary from program to program or even run to run, and so the average is only a crude approximation. The measured average would also only be valid for the combination of hardware and software and so would need to be recalibrated each time a system update was done. It could also vary over time even if the system is not changed, due to factors such as file system fragmentation.

Instead, we measure the startup time of each run as it is happening. To accomplish this, we do not launch the user's Java class directly. Instead, we launch a helper class that does the following:

1. Uses reflection to look up the `main` method in the user's class.
2. Calls a native function we have written that measures the time elapsed so far, and passes it to the process that manages time limit enforcement.
3. Invokes the main method in the user's class.

We discuss these steps in detail in the following subsections.

2.1. *Class Lookup*

Unlike languages such as C, Java provides reflection mechanisms that allow classes and methods to be looked up by name at run time. The name of the user's class is passed to the wrapper on the command-line, and its main method is located with the following code:

```
Class<?> childClass;
Method childMain;
Object childArgs;
```

```
childClass = Class.forName(args[0], false,
    JavaWrapper.class.getClassLoader());
childMain = childClass.getMethod("main", String[].class);
childArgs = Arrays.copyOfRange(args, 1, args.length);
```

It is not strictly necessary to do this class lookup dynamically; one could instead generate a wrapper on the fly for each possible user class name. We have chosen to use reflection largely for convenience.

2.2. *Reporting Startup Time*

Standard Java libraries do not provide a means of determining the CPU time consumed so far. Instead, we use Java Native Interface (JNI), a mechanism that allows Java methods to be implemented in a native language such as C. This C code is compiled into a shared object which the JVM loads at run time.

Since our evaluation system is based on GNU/Linux, our native code calls `getrusage` to determine the CPU time that has been consumed by startup overheads. It then needs to report this to the parent process (which is doing timing and enforcing resource limits), in a secure way. We chose to have the parent process open a pipe between parent and child when the child is created, with a fixed file descriptor number (3). The native code running in the child writes the startup time to this pipe, and then closes it to ensure that the user's code cannot access it.

2.3. *Launching the User's Class*

Having measured all startup overheads to this point, we are ready to launch the user's code. This is done by calling

```
childMain.invoke(null, childArgs).
```

This passes any remaining command-line arguments to the child. Most olympiad problems do not process command-line arguments, but doing so makes the wrapper general-purpose.

2.4. *Exception Handling*

The sample code listed above does not include any exception handling. There is relatively little exception handling required, because an exception will usually indicate a fault in the user's submission (whether because their code threw an exception, or their class didn't have a `main` method with the appropriate signature or permissions, or some other reflection issue), and the submission will score zero with the exact reason being irrelevant. There is one special case: because the parent process is waiting for the child to inform it of the startup time, this must be reported even if the class lookup throws an exception.

Our use of reflection causes exception messages to be different from those that would be seen had the code been run outside the wrapper. This is because exceptions thrown by the invoked method are wrapped in another exception, such as `InvocationTargetException`, to allow exception handlers to distinguish them from exceptions in the invocation process itself. To make our wrapper more transparent, we catch these exceptions and re-throw the underlying exception, causing the error log to match what would have appeared in the absence of the wrapper.

2.5. *Security Considerations*

No matter how well this scheme works, it would be of dubious practical value if contestant code could either perform computation in the nominal "startup" time (thus bypassing the resource limit), or was able to report a false startup time to the parent process. In this section we consider some possible ways that user code might attack the system.

First we consider whether user code can execute during "startup" time. Because we load the user's class before recording the startup time (so that the time taken for this reflection process does not count against the user), we must not allow any user initialisation code, such as a static class initialiser, to be run during this loading. This is done by passing `false` to `Class.forName`, which indicates that the class should not be initialised at this point. The class is instead initialised when the `main` method is invoked.

Since no user code is able to run before the startup time is reported to the parent, it cannot interfere with the reporting process. It cannot even send false data along the same pipe, because the native code that does the reporting closes the pipe before it returns.

One other risk is the presence of the native code: the user's code can cause it to execute again (for example, by calling the `main` method in the wrapper class), and so it must be made robust to repeated execution.

3. Test Setup

In order to test the usage of our wrapper, we used the solutions to several olympiad problems to generate statistics about run time under various scenarios. Specifically, we took the following measurements:

- *Execution time* – the total time taken by the process, including any startup and wrapperoverheads.
- *Startup time* – the startup time measured by our wrapper, prior to the user's code being launched.
- *Solution time* – the time taken by the user's code, computed as the difference between execution time and startup time.

Where our wrapper has not been used, only execution time can be measured directly.

Our test machine has a 2.16G Hz Core 2 Duo T7400, with 1GB RAM, and runs a Linux 2.6.36 64-bit preemptable kernel and 64-bit Gentoo OS. C++ solutions were compiled with GCC 4.4.5 with options `-O2 -s -static` while Java solutions were compiled and executed with Sun Java SE Runtime Environment 1.6.0_24-b07.

The C++ and Java solutions for each problem are functionally identical, so as to minimise the effects of different algorithms impacting our measurements. We also include a special `empty` program which, as its name implies, does no processing and returns immediately. The C++ and Java `empty` programs are as follows:

```
// C++
int main()
{
    return 0;
}
```

```
// Java
public class empty
{
    public static void main(String[] args)
    {
    }
}
```

Since we are aiming to measure startup time rather than solution speed, all our tests were done with very small input files for which processing is expected to take almost zero time.

For statistical significance, each combination of solution, language, and the presence or absence of the wrapper was run 200 times.

One limitation in our approach for collecting test data is the quantisation of the results returned by the `getrusage` system call. The kernel on our test system has a tick frequency of 1000 Hz, which quantises our results to 1 ms buckets. We have not attempted to take these quantisation effects into account when applying statistical tests, and so p values quoted may be inaccurate.

4. Results

The figures below show box plots. Each box represents the inter-quartile range, the bar inside the box represents the median, and the small circles represent outliers (the 1 ms quantisation causes more data points to be flagged as outliers than might otherwise be the case).

We now address the questions listed in Section 1. Firstly we consider C++: every run on all our test programs reported an execution time of 1 ms. It is likely that less than 1 ms was required, but that the 1 ms quantisation caused everything to be rounded up to 1 ms.

We can estimate Java startup time without the use of our wrapper. Since our `empty` program performs no computation, its execution time is essentially the startup time. Figures 1 and 2 show that, on average:

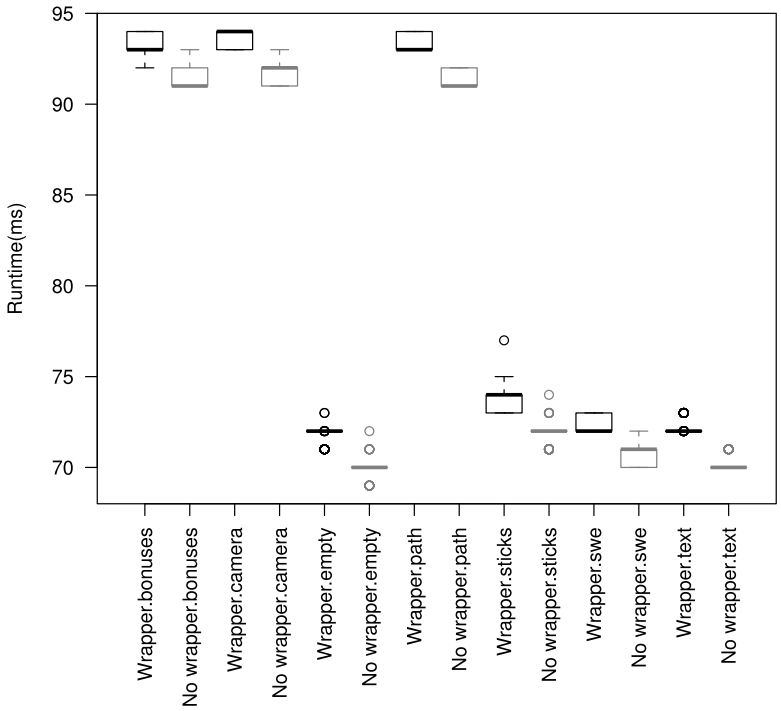

Fig. 1. Execution time for various Java programs, without the use of our wrapper.

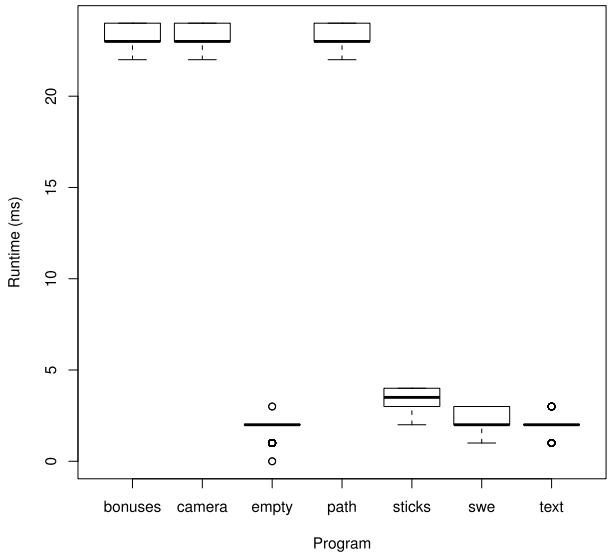

Fig. 2. Solution time for various Java programs with their sample test-case as input, after subtracting the measured startup time of the JVM.

- Java with no wrapper has an execution time of 70.0 ms.
- Java with our wrapper has an execution time of 71.8 ms.

In each of these data-sets, the standard deviation of the measurements is less than 0.5 ms and is thus unreliable in telling us how the individual times vary, due to the quantisation of our measurements to 1 ms buckets. Figure 3 gives an alternative view of the effect of our wrapper, showing the average startup time and solution time for each program. It shows that of the 71.8 ms average time for the empty program, only 1.8 ms is solution time and the remainder is startup time.

These data also address our question of whether the use of our wrapper impacts execution time. The mean execution time for the `empty` Java program is clearly greater than that of the same program without our wrapper, and a t-test confirms that this finding is statistically significant with $p < 0.001$.

Furthermore, it is clear both from the data and conventional wisdom that the startup time of a Java program is greater than that of a C++ program – and our data shows that this is true even when taking the startup time into account. Applying a t-test to our data, we find that the solution time of the wrapped `empty` Java program (1.8 ms) is statistically significantly more than the 1 ms execution time of the `empty` C++ program, with $p < 0.001$. Similarly, the execution time of the unwrapped `empty` Java program (70 ms) is also statistically significantly more than that of C++, again with $p < 0.001$.

The figures also show a clear variation in execution time between the programs; from Fig. 3 and 4 it is clear that this variation is part of the solution time, rather than the startup time. Investigation showed that the three slow programs (`bonuses`, `camera` and `path`) all used the `Scanner` class for parsing input, while the remaining programs did not. This class is more convenient than other methods of parsing input (such as

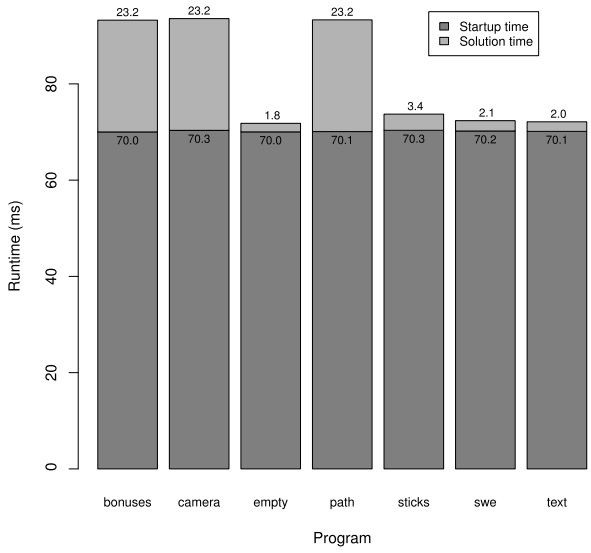

Fig. 3. The average startup time and solution time for various Java programs.

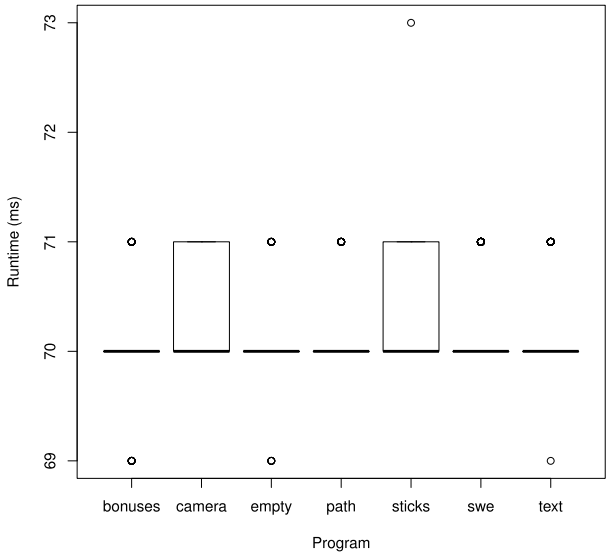

Fig. 4. Startup time measured by our Java wrapper for various Java programs.

`StringTokenizer`), but apparently this convenience comes at a price in initialisation time.

Finally, we address the question of whether subtracting the startup time for a Java program reduces the variation in reported run time. From Figs. 1 and 2, it appears that the standard deviation in run time for each program has not improved after subtracting the startup time. This is further supported by the box plots in Fig. 4, which show that the measured startup time is almost constant at 70 ms. However, given that all the standard deviations in these data sets are less than 1 ms, we cannot draw any definite conclusions due to the 1 ms quantisation of our measurements.

5. Conclusions

Prior experience, both our own and that reported by Pohl (2006), led us to expect that the startup time of Java programs would be significant and highly variable, and that our technique would be valuable for programming contests using Java with time limits of up to a few seconds.

We were thus surprised to find that the startup time we measured was only around 70 ms, and highly consistent across different programs and different runs. Indeed, the variation observed between programs was not due to the startup time of the JVM, but to run time consumed by the `Scanner` class. It is not currently known whether this small and consistent startup time is due to improvements in the Virtual Machine implementation since 2006, or to changes in the underlying environment (faster CPUs and larger caches).

Given the consistency of the measured startup time, it seems that a simple solution of applying a fixed correction to all Java run times is likely to be sufficient. Nevertheless,

our method is particularly convenient since it does not require separate calibration. It also adds only a few milliseconds to the total execution time, and thus will have minimal impact on the rate at which solutions can be judged.

After correcting for the JVM startup time, a trivial Java program runs almost as fast as a trivial C++ program – the difference is less than the quantisation error caused by operating system time-slicing. We thus feel confident that while execution speed may still be a concern (and is beyond the scope of this paper), startup time is no longer an issue when supporting Java as a programming language in a contest.

5.1. *Future Work*

Although we have addressed the issue of startup time in Java programs, there are several additional avenues for research that would assist in supporting the more widespread adoption of Java in programming contests:

- **Class-specific wrapper.** In Section 2.1 we described how we use reflection to look up the user's class, and noted that an alternative approach would be to generate a wrapper on the fly and avoid the reflective lookup. This approach may reduce the additional run time overhead incurred by the wrapper.
- **Shutdown time.** Although our technique focuses on reducing startup time, a similar approach might be used to measure *shutdown time* (that is, the time required for the JVM to terminate after the user's program has completed).
- **Preloading of key classes**. Our analysis identified that the use of certain classes (such as `Scanner`) can have a measurable impact on the startup time of a program. One possible means of ameliorating the effect of such classes would be to preload them in the wrapper, and thus separate their initialisation time from the time attributable to the user's program. Further investigation would be required to determine whether this overhead is for class initialisation or is per-instance.
- **JVM priming.** The JVM is well known for performing just-in-time compilation and adaptive optimisation of executing bytecode, which can result in fully optimised machine code only being executed at some indeterminate point in the program's execution. Java programs may therefore be at a disadvantage to those written in other languages, such as C++ or Pascal, where program code is precompiled straight to optimised machine code. One possible means of reducing this disparity would be to "prime" the JVM by executing the Java program many times within the same JVM instance, thus giving it an opportunity toe fully optimise and compile all the code. Care would, however, need to be taken in such an approach to ensure that the user's program did not store any information for use in successive runs.
- **Other JVM implementations.** This paper has focused on the Sun JVM; it would be interesting to perform similar analysis on other JVM implementations.

References

ICPC. ICPCWiki: World finals rules. (2010).
http://cm.baylor.edu/ICPCWiki/Wiki.jsp?page=World%20Finals%20Rules.
http://cm.baylor.edu/ICPCWiki/Wiki.jsp?page=World%20Finals%20Rules.
Kolstad, R. (2009). Infrastructure for contest task development. *Olympiads in Informatics*, 3, 38–59.
Merry, B., Gallotta, M., Hultquist, C. (2008). Challenges in running a computer olympiad in South Africa. *Olympiads in Informatics*, 2, 105–114.
Pohl, W. (2006). *IOI Newsletter*, 1.
http://ioinformatics.org/newsletters/html/ioinews6.htm.
http://ioinformatics.org/newsletters/html/ioinews6.htm.
Tani, S., Moriya, E. (2008). Japanese olympiad in informatics. *Olympiads in Informatics*, 2, 163–170.
Trefz, N. (2007). The coding window – TopCoder wiki.
http://www.topcoder.com/wiki/display/tc/The+Coding+Window.
http://www.topcoder.com/wiki/display/tc/The+Coding+Window.

B. Merry took part in the IOI from 1996 to 2001, winning two gold medals. Since then he has been involved in numerous programming contests, as well as South Africa's IOI training programme. He obtained his PhD in computer science from the University of Cape Town and is now a software engineer at ARM.

C. Hultquist took part in the IOI from 1999 to 2000, winning a silver medal and a bronze medal. He has since taken part in and organised several other programming contests, including active involvement in South Africa's IOI training programme. Carl obtained his PhD in computer science from the University of Cape Town and is currently working as a software engineer for D.E. Shaw.