The Simplest Heuristics May Be the Best in Java JIT Compilers

Jonathan L. Schilling
Caldera International, Inc.
430 Mountain Avenue
Murray Hill, NJ 07974
jls@caldera.com

Abstract

The simplest strategy in Java just-in-time (JIT) compilers is to compile each Java method the first time it is called. However, better performance can often be obtained by selectively compiling methods based on heuristics of how often they are likely to be called during the rest of the program's execution. Various heuristics are examined when used as part of the Caldera UNIX Java JIT compiler. The simplest heuristics involving the number of times the method has executed so far and the size of the method prove to be the most effective, with more complicated heuristics not providing much or any additional benefit.

Keywords

Java, just-in-time compiler, JIT, selective compilation, heuristics, performance.

1 Introduction

A Java just-in-time (JIT) compiler is an optimization component within Java virtual machines. Instead of Java bytecodes being interpreted each time they are executed, the bytecodes are compiled to the underlying machine code and from that point on the generated machine code is executed.

It follows from this scheme that there is a fundamental tradeoff in JIT compilers. Compiling the bytecodes will cost some appreciable amount of time, while executing generated code instead of interpreting bytecode will presumably save some amount of time each subsequent time the code executes. Depending upon the actual times involved, the compilation of a given piece of bytecode may or may not prove to be a net benefit over the course of any given application's run.

Decisions about whether to compile bytecodes are usually made on a per-method basis. The simplest JIT compilation strategy is to compile every method the first time it is called. More ambitious JIT strategies involve *selective compilation*: some methods get compiled the first time they are called, some methods get compiled later, and some methods never get compiled at all.

Given the dynamic nature of the Java language and its applications, it is usually not feasible to know ahead of time how long methods will take to compile, execute interpretively, or execute as generated code. Thus selective compilation decisions rely on *heuristics* about whether compiling a method will likely result in a time savings for that method.

The goal of the work described here is to see how the addition of selective compilation heuristics can improve the performance of a particular existing virtual machine and JIT compiler.

2 Architecture of the JVM and JIT

This paper describes work done to the Java virtual machine in various versions of the Java Development Kit and Java 2 Platform, Standard Edition for SCO and Caldera UNIX Operating Systems.[1]

[1]Caldera acquired the UNIX business of SCO, Inc. in May 2001. Much of the work this paper describes was done while the author was employed by SCO; for the purposes of this paper the two companies may be considered as the same. On the Caldera OpenLinux® operating system, a different Java implementation is used that is not discussed in this paper.

This Java implementation is based on the original, "classic VM" from Sun Microsystems, Inc., of which SCO and Caldera have been source licensees. Except for the JIT compiler, most of the rest of the Java virtual machine is a straight port of the classic VM, from the Sun Solaris operating system for the Intel IA-32 architecture, to the Open UNIX® 8, UnixWare® 7, and SCO OpenServer™ 5 operating systems for IA-32.

The JIT compiler described here is based upon *sunwjit*, a JIT compiler for Solaris/IA-32 developed by a separate product group within Sun and licensed by SCO and Caldera.[2]

The sunwjit compiler is loaded and invoked per the JIT Interface Specification [19]. When each Java class is first loaded, initialization processing is done to get ready for JIT usage. This includes setting up bits of "shim" code to handle flow of control transitions among interpreted code, compiled code, and machine code. An internal JVM structure known as the "method block" has its "invoker" field [5] set to point to the JIT compiler. When a method in the class is to be compiled, the JIT compiler gets control, translates the method's bytecode down to native IA-32 machine code, and then changes the method block invoker field to point to the generated machine code. The call is then re-invoked, and on this and subsequent invocations the generated machine code executes instead of the bytecode interpreter. In a multithreaded program, compilation may proceed concurrently with execution (interpreted or compiled code) in other parts of the program; only certain operations pertaining to the class of the method being compiled are locked out. This avoids some of the compilation inefficiencies described in [13].

Compiling of methods the first time they are called is done unconditionally in the original Sun version of sunwjit. The only times methods are interpreted are if the JIT is suppressed by a JVM command-line option, or if the JIT has an internal error while compiling, or if the method belongs to one of the primordial classes that execute before the Java `java.lang.Compiler` class is loaded. In the latter case, primordial methods will get compiled the first time they are subsequently executed after the JIT compiler is loaded. Class initializers, which are known to execute at most once, are never compiled. Methods that are never called are never compiled.

[2]The sunwjit JIT compiler also has a code generator for Solaris/SPARC and was used by Sun with the classic VM on that platform as well. Sun does not do further development work on any version of sunwjit, concentrating instead on their successor HotSpot JVM technology.

The code generator in sunwjit has two passes. The first is architecture-independent and scans the bytecode for basic block and stack state information. The second is architecture-specific and generates the actual machine code. There is no global optimization attempted, although "lazy code generation" modeling of the bytecode stack at compile time is performed in the manner of [5], and various other local optimizations are done as well. In general, sunwjit tries to do a decent but not great code generation job, quickly. There is no ability to have different levels of optimization performed.

Caldera has made a number of changes to sunwjit. Some have been due to porting differences between Solaris and the SCO/Caldera UNIX operating systems: signal handling for trapping null reference checks, disassembly logic for tracing through pre-JNI native method stubs, probe techniques for detecting stack overflow, and so on. Others have been due to bug fixes, especially for race conditions in the dynamic code patching logic.

In addition, the infrastructure of the JIT has been modified to permit selective compilation, as described in the remainder of this paper. This includes having the JIT be able to read in and compile all of the "quick"-form bytecodes [9] that are generated after methods have been interpreted for the first time. This also includes having the JIT's exception handling and stack frame walkback logic handle the case where a recursively called method has both interpreted and compiled instances on the JVM's internal stack frame structures at the same time. These and other modifications become necessary once any method may be interpreted an arbitrary number of times before it is compiled.

Finally, the JIT compiler sometimes detects unusual bytecode situations that it cannot handle correctly, such as empty loops and overflows of the IA-32 floating point stack. In these cases, JIT compilation is abandoned, and the JVM will continue to interpret the method each time it is called. This approach avoids adding excessive complexity to the JIT to handle rarely-encountered situations. It is also in accordance with the principle of fail-safe optimization during compilation [15].

There are only two possible execution states within the JVM for any particular execution of a method: it is either fully interpreted or fully compiled (and for debugging purposes either state can be forced on for a particular method by JVM invocation options). There are no cases where compilation is started while the method is being interpreted or where code has been compiled for use but is then backed out and interpretation is resumed. This "one or the other" nature simplifies the JVM and

	no JIT	always JIT	times heuristic
actual javac	7.19	8.38	6.77

Table 1: Early results with JIT

makes for easier reproducibility and debugging of problems; more aggressive JVM compilation strategies often run into difficulties in this area [4, 14].

3 The Simple Heuristics

In early SCO Java Development Kit releases, the JIT was used in the unmodified Sun mode of compiling every method the first time it is called. This resulted in a performance improvement for most applications, but there were exceptions. Among the most notable and visible of these was the `javac` compiler (written in Java and used to compile Java source code to bytecode), which became *slower* when run with the JIT on. A typical result (on a Pentium II 266 MHz machine, 64M RAM, UnixWare 7) in translating a modest source program that computes the palindrome conjecture [22] would be 7.19 seconds without the JIT[3] and 8.38 seconds with the JIT, for a 16% slowdown. (Times are the sum of the UNIX `timex` command user and system time.)

Since `javac` is most developers' first exposure to Java performance concerns, this was not the right direction to be moving in! Also notably slow was the startup time of many graphical applications. Not only does such poor response annoy users, but it can also lead to users changing their behavior regarding using the application [11, 21]. Moreover, non-graphical server application startup time can also be important, especially when user-visible application interfaces are down awaiting a re-start of their server component.

Thus a simple selective compilation heuristic was added in the SCO JDK 1.1.7B release. If the user set a particular environment variable to some positive integer, no method would be compiled until it had executed that many times.

This is based on the idea that you only want to compile methods that are likely to execute a good number of times during the execution of the application (so as to amortize the cost of compilation), and that the methods that are most likely to execute a lot during the whole application are those that have already been executing a lot up until "now" in the application. This is a simplified view of the heuristic described in [1].

As shown in Table 1, by running `javac` with the environment variable set to 40, the same translation ran 6.77 seconds, a speedup of 6% over no JIT and 19% over always running the JIT.

This improvement scheme was limited, however, by not being on by default. It required users to have to know about the feature to make use of it and to have to experiment by trial-and-error to get a somewhat optimal value for the environment variable. So the next step was to make selection compilation on by default and more adaptable to the characteristics of individual methods.

Intuitively, one would think that besides the number of times a method has executed so far in the application, the other simple heuristic should be the size of a method (as measured by the number of bytes of bytecode for the method). On average, larger methods should take longer to compile than smaller methods, and as a rule larger methods should take longer to execute (whether compiled or interpreted) than smaller methods. (We will show later that this is not really the case, but still an important factor.)

As a consequence, for example, you almost certainly *do not* want to compile a small method that is only going to execute once, since the cost of compiling it will well exceed the small amount of execution time gained, whereas you almost certainly *do* want to compile a large method that is going to execute many times, since the cost of compilation will be more than made up for by the larger amount of execution time gained over and over. Whether you want to compile a large method that is only going to execute a few times, or whether you want to compile a small method that is going to execute many times, is less clear.

Looking at the boundaries of this decision, there should be some point where a method's size is big enough that you want to compile it, even if it will only execute once. Similarly, there should be some point where if a method executes that many times you want to compile it, even if it is very small. These boundary points are shown in Figure 1, and are labeled `JIT_MIN_SIZE` and `JIT_MIN_TIMES` respectively. The slope between these points is the "dividing line" of this decision guidance: to the left, the cost of compilation outweighs the benefit and you do not want to compile; to the right, the benefit outweighs the cost, and you do want to compile.

[3]The interpreter used when the JIT is not on is implemented in tightly-written assembly language.

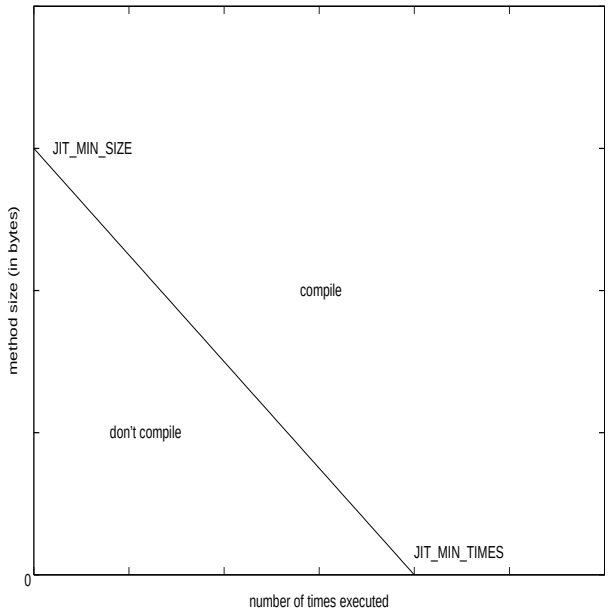

Figure 1: Selective compilation based on both method size and times executed.

This heuristic is computed when the class is first loaded. The JIT gets control to initialize itself for the whole class (but not yet compile any methods). A counter field in the method block is set to the number of times the method should execute before it is compiled. Each time the method executes, the modified JVM interpreter decrements this counter; once it reaches zero, the JIT is invoked to compile the method. This is similar to one of the techniques used by the IBM JIT compiler [17, 18].

Experimentation with a variety of benchmarks and real applications has been done to establish what these boundary points are. The values of `JIT_MIN_TIMES`=40 and `JIT_MIN_SIZE`=150 seem to be best.[4]

4 Results of Simple Heuristics

The performance results in this section show test programs being run five ways: with no JIT at all; with unconditional compilation the first time a method is called; with the default scheme combining the number of times executed and size of a method as described in the previous section and shown by Figure 1 (`JIT_MIN_TIMES`=40 and `JIT_MIN_SIZE`=150); with a heuristic that only uses the number of times executed (`JIT_MIN_TIMES`=40) and ignores the size of a method; and with a heuristic that only uses the size of a method (`JIT_MIN_SIZE`=150) and ignores the number of times executed.

Table 2 shows the results of the SPEC JVM98 benchmarks [16].[5] They are run using Java 2 Standard Edition version 1.3.0 on a dual processor Xeon 400 MHz system, 512 MB RAM, UnixWare 7.1.1. The benchmarks are run with the `-s10` (medium) size, which is sometimes used by researchers to represent short-lived applications or the start-up time for longer-lived ones [3]. Lower numbers are better.

The JVM98 results show that overall, unconditional compilation is slightly better than the default heuristic, which stands to reason for a benchmark that repeatedly executes the same code. ("Repeated execution of code" is not purely a benchmark artifice; it can occur in real, long-running applications that achieve a steady state of execution behavior.) The other three options fare poorly on one or more of the tests; in the case of the heuristic that ignores the size of a method, it is in methods that do *not* repeat a lot that performance is lost.

To contrast with the effect of repeated execution, Table 3 shows a reprise of the times for a stand-alone invocation of the `javac` command (on the same source file as used in Table 1 but on a different machine), this time as part of Java 1.3.0 and with all five ways of running it.[6]

The `javac` results show that it is still the case that unconditional compilation can not only be worse than selective compilation, but can also be worse than not compiling at all.[7] The heuristic that doesn't include size of methods and the default heuristic are close together as the best options.

Finally, Table 4 shows the start-up times of two longer-

[4]These values can be overridden by environment variables of the same name, and sophisticated users can do so if they desire for their specific application. Setting either to zero means compiling methods unconditionally, and setting either to a huge value means only using the other setting in deciding when to compile.

[5]These results are run in batch mode as part of "Research Use". They do not follow the official SPEC run or reporting rules and should not be treated as official SPEC results.

[6]Although javac is one of the benchmarks in SPEC JVM98, this stand-alone use differs because it is a different Sun implementation, running on a different source program, and without repeated execution. Also, javac is the JVM98 benchmark that for some reason shows the most variation in timings from one run to another, while stand-alone `javac` timings are much more stable.

[7]Actually, for *very* short-lived programs (e.g. execution of a `hello` program, or even execution of the palindrome program for a terminating integer such as 187), not compiling at all will produce the best time of all these options. But optimizing programs that take less than a second to execute is not usually a concern on this platform.

	no JIT	always JIT	(default) times & size heuristic	times-only heuristic	size-only heuristic
compress	41.66	5.69	6.08	33.46	6.87
jess	7.55	2.92	2.97	2.85	5.62
db	5.71	3.35	3.40	4.78	4.15
javac	7.34	4.37	5.00	4.56	8.65
mpegaudio	39.54	5.80	5.83	6.42	12.46
mtrt	11.93	5.62	5.47	5.48	11.48
jack	14.51	10.07	10.14	9.84	16.38

Table 2: SPEC JVM98 results.

	no JIT	always JIT	(default) times & size heuristic	times-only heuristic	size-only heuristic
actual javac	3.07	3.47	2.90	2.88	3.14

Table 3: Later javac results.

lived applications. SwingSet2 is a Sun demonstration program that brings up a graphical user interface illustrating a large number of different Java Swing GUI elements. Tarantella ObjectManager is a GUI for the administration of the Tarantella application database.[8] In both cases, the time is measured in wall clock seconds from program start until the GUI is fully presented.

This attention to startup time is important for the psychological reasons referred to earlier. It is also an area ripe for optimization: one investigation found that on average 77% of execution-time Java compilation overhead occurs in the initial 10% of program execution [7].

These start-up time results again show that unconditional compilation can be a losing strategy, and that the default heuristic and the heuristic that only uses times executed are close together as the best choices.

So in sum, what do all of these results show?

- Not using the JIT at all gives bad performance for all repeatedly executed code.
- Unconditional compilation gives inferior performance for some short-lived applications and for the start-up times of longer-lived applications.
- Picking methods to compile based purely on the number of times executed gives bad performance for some short-lived applications.
- Picking methods to compile based purely on size does not work well in any situation.
- The default scheme of combining number of times executed with size of method works the best overall; it does not do badly in any of these different situations.

Selective compiling during application start-up does mean that some methods will get compiled later, during normal application processing. Does this increase response time or overall execution time? Experience with interactive graphical applications has shown that it does not: later compilations are interleaved with user pauses. Running the long-lived SPEC JBB2000 benchmark [16] shows no significant difference between unconditional compilation and the default heuristic (the latter is better by less than 1%; the other two JIT options are worse, and no JIT at all is much worse).

Interestingly, some results of the selective compilation heuristics were better in early-stage implementations that were part of Java 1.2.2. This is probably because there was more execution-intensive Java code in that version of the Java tools and libraries (Sun made many performance improvements in the tools and libraries in Java 1.3.0), which gave more of an opportunity to improve time with generated code where it was warranted and also to save time from not compiling where it was not warranted. Nevertheless, the ability to improve performance of loose Java code is still important for user-written applications, where there will typically be less knowledge of how to write tight Java code.

[8] Tarantella is an Internet infrastructure product that enables web-based access to enterprise applications [20]. Tarantella, Inc. used to be part of SCO, Inc.

	no JIT	always JIT	(default) times & size heuristic	times-only heuristic	size-only heuristic
SwingSet2	23.6	24.2	21.6	21.8	27.2
Tarantella ObjectManager	22.7	20.6	16.7	16.3	24.8

Table 4: Start-up time results.

As a final note, it is not fruitful to compare these results with those from other Java JIT compilers or Java virtual machines, because overall performance differences often derive from completely unrelated characteristics of the underlying virtual machine or operating system (such as threads and synchronization model, memory model, and so forth), and the goal of this work was simply to see the effects of selective compilation.

5 Profiling of JIT Behavior

In order to better understand some of the factors that go into deciding when to compile methods, and how possibly the selective compilation heuristics could be further refined, the SCO/Caldera JVM and JIT were instrumented to report relevant profiling information. The intent was to study and process the collected information off-line, and not to make use of it as part of execution-time decisions about when to compile (which, due to the limitations described in this section, was not deemed feasible). Therefore, the profiling mechanism did not have to run fast itself, although it did have to perturb the application to the least extent possible.

First an issue of timing accuracy had to be resolved. The Sun JVM uses the Solaris `gethrvtime()` call to get accurate timing information on a per-thread basis. There is no equivalent to this call in SCO/Caldera operating systems. Therefore the JVM timing primitives were modified to access the IA-32 (Pentium processors and later) `RDTSC` instruction to get the current clock cycle count; relative times could be constructed from that, once scaled for the machine's processor speed. This provides timings with more than sufficient granularity, but it counts everything that happens from point A to point B, including context switches and time spent in other threads, time spent in the operating system, etc. To help reduce the effect of this, the measurements made in this section were made while running under the classic VM's user-space, single-process, non-preemptive "green threads" threads implementation mode, rather than under the usual "native threads" mode. This prevents any real concurrency from taking place, and allows most method calls to complete without any context switching taking place.

The next step was to record statistics of method calls during the lifetime of a program. This was done both for the case where every method is being interpreted (JIT suppressed) and for the case where every method is being compiled. Recorded for every method call during the lifetime of the program were: the size of the method, the number of times the method is executed, the amount of time it took to interpret the method, the amount of time it took to compile the method (if and when that happened), and the amount of time it took to execute the generated code for the method once compiled. This profiling was done by starting with the Sun HPROF profiler agent, which follows the Java Virtual Machine Profiler Interface (JVMPI). It has an ability to do CPU time profiling by code instrumentation (rather than statistical sampling) [8].[9] HPROF was then modified to implement the `JVMPI_EVENT_COMPILED_METHOD_LOAD` action, to record how long it takes to compile a method, and by modifying the `JVMPI_EVENT_METHOD_ENTRY` and `JVMPI_EVENT_METHOD_EXIT` actions, to record each method's time of execution. (These actions already know how to adjust method timings for the time spent in called methods and the time spent in garbage collection.[10]) Since the JVM interpreter and the JIT honor the JVMPI, this captures method timings whether the JIT is used or not.

At the end of JVM execution, these times are recorded to an external file, which is processed by off-line UNIX commands and scripts to produce, for each method used in the program, the desired summary data.

So the first question to ask is, what is the relationship between the size of a method's bytecodes and the

[9] Statistical sampling is claimed more effective in somewhat different contexts in [3] and [18], but for the purposes of this investigation was too coarse.

[10] In practice, there were rare occasions where for unknown reasons these adjustments produced negative times; these instances were ignored. Also, sometimes the executed code timings produced an obvious outlier value for a particular invocation of a method; these were adjusted to a reasonable time based on other invocations.

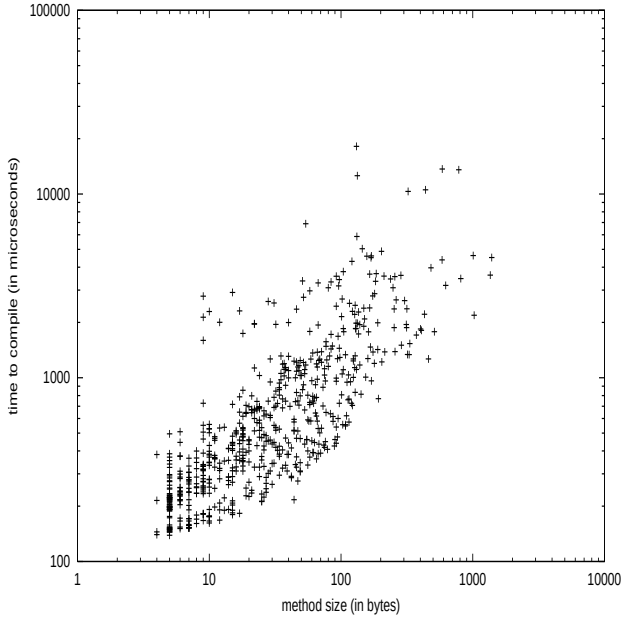

Figure 2: JIT compile time as a function of method size.

amount of time it takes the JIT to compile that method? Figure 2 plots this, for a run of the `javac` application. The two axes are in logarithmic scale, to collapse the wide range of values recorded.

This data shows a reasonably linear relationship between the size of a method and the amount of time it takes to compile it. The same graph for other test programs, such as SPEC JVM98 or some graphical programs, shows a very similar pattern. This is what one would expect for a compiler that does not do global optimization, and verifies an assumption made in [10].[11]

Next, we look at the relationship between the size of a method and the amount of time it takes to execute it. Again using a run of `javac`, Figure 3 plots this for interpreted execution, and Figure 4 plots it for compiled code execution.

In both cases there does not seem to be much of a relationship at all! (As before, a similar pattern shows for other applications.)

Yet the results of the previous section show that overall, a heuristic based on both size and number of times executed produces better results than a heuristic based only on number of times executed. Why would this be?

[11]It does not look to be quadratic, as was found in a study of preprocessed C compilation times that was being used as a proxy for Java just-in-time compilation times in an early paper [13].

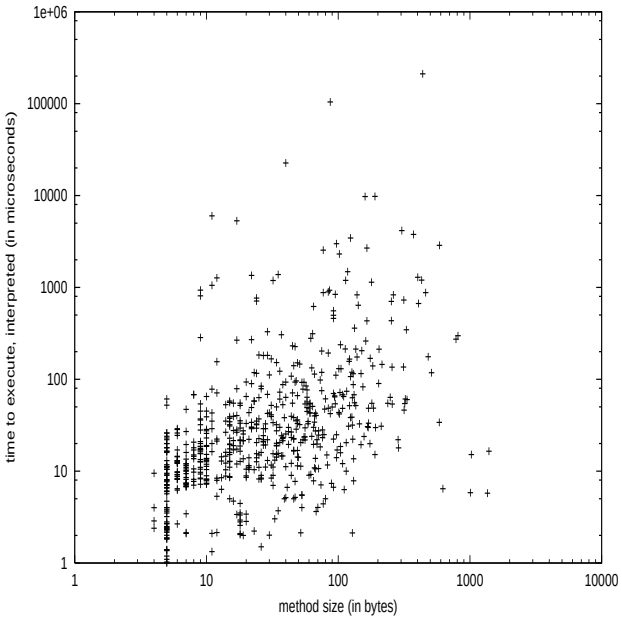

Figure 3: JVM interpreted execution time as a function of method size.

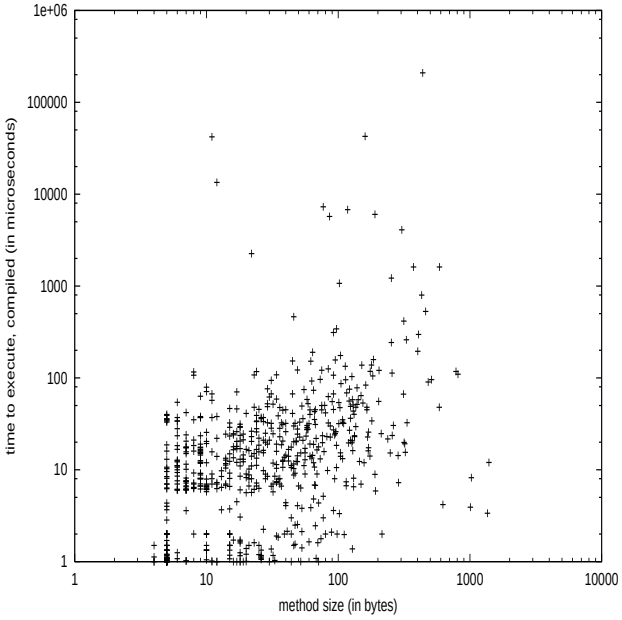

Figure 4: JIT-compiled-code execution time as a function of method size.

The most likely answer is that while large-sized methods do not always take a correspondingly large amount of time to execute, sometimes they do. If you compile a method unnecessarily, the negative benefit is limited to the time it takes to compile, which is a fixed amount. But if you do not compile a large method that does not execute frequently, but does take a long time to execute (perhaps because there is a loop inside it that is executed many times), then the negative benefit becomes huge and is almost unlimited. In other words, the opportunity cost of the compilation decision is bounded in one direction but effectively unbounded in the other direction. Thus a selective compilation heuristic that takes method size into account will be more effective than one that does not.

6 More Complicated Heuristics

It is shown in [3] that predicting which methods to optimize is much harder for short-running applications than for longer-running applications. Accordingly, a variety of more complicated heuristics were tried in an attempt to make better predications.

6.1 The "jit when called by jitted" heuristic

One idea is that once we have decided to compile a method X, to then also immediately compile every method that X calls. The theory is that this will take advantage of locality of reference, in the sense that if method X is getting used a lot, it is likely that the methods Y, Z, etc. that X calls will get used a lot too. Thus we compile Y and Z right away and do not wait for them to reach the point at which they would normally be compiled (thereby saving the excess time that they would be interpreted).

However, measurements of this scheme show that it makes things worse, not better. The Tarantella ObjectManager is 20% slower to start up, the SwingSet2 demo is 21% slower to start up, and javac is 3% slower to run. No tests run faster with it. This suggests that this kind of "locality of reference" does not really exist.

6.2 The "Square decision" heuristic

Another idea is to change the shape of the when-to-compile decision. Instead of interpolating a sloped line between the `JIT_MIN_TIMES` and `JIT_MIN_SIZE` points, as was shown in Figure 1, the decision could be either-or: if a method exceeds the minimum size or (eventually) the minimum number of times to be executed, it is compiled, otherwise it stays interpreted. The shape of the graph would be a right angle, not a sloping line; see Figure 5.

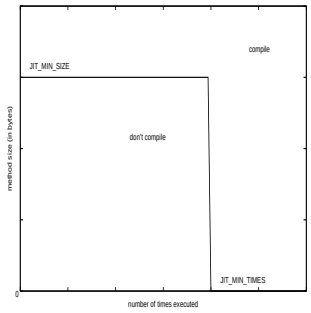

Figure 5: An alternative selective compilation heuristic (not used).

Measurements of this scheme against the default scheme generally showed only slight variations on the SPEC JVM98 benchmarks (± 1% or less, within the margin of error), and it was 3% worse on Tarantella ObjectManager startup. Since intuitively this scheme is less flexible than the default scheme, it was not further explored.

6.3 The "Backward branches" heuristic

Another idea is to try to identify backward branches in methods, assuming that they are representative of loops and likely hot spots in the code. Once identified, the method would be compiled immediately, or at least sooner than it would be otherwise. This heuristic is used in the IBM JIT compiler [17].

However, implementing the heuristic in the SCO/Caldera JIT proved difficult. The bytecode could be scanned during the class loading and verification phase, but given the internal structure of the classic VM there was no good way to pass the result to the JIT. Also, such a scan would potentially flag backward branches in sections of code that rarely or never execute.

So instead, backward branches were looked for during actual interpretation of the method. As it happens, this check is difficult and expensive to do in the optimized, assembly language version of the classic VM interpreter. Therefore, it was first prototyped in the C language version of the interpreter, which is used when the `java_g`

debugging version of the JVM command is run. Once a backward branch is detected in a method, the method is compiled the next time it is called.

Measurements of this scheme against the default scheme for the SPEC JVM98 benchmarks showed that the best times of each were very close. However, there was more variability from run to run in these times, possibly because the larger code of the C interpreter was subject to more cache effects, and the slower default times were worse than the slower backward-branch times. Nevertheless, given that compiled code had more than the usual advantage over interpreted code in this case, the lack of consistently superior results for the backward branch scheme was not encouraging, and the scheme was not further pursued. (The scheme might have proven beneficial with some more effort put into it.)

6.4 The "Core classes known to compile" heuristic

The final idea, and the one of these that was most explored, is to take advantage of the off-line profiling data captured by the procedures described in the previous section. Then, decisions on whether to compile a method immediately can be made at the beginning of application execution, based on whether it is "known" that the method will reach its "crossover point" [13] where compilation is sure to be beneficial. For example, assume that our profiling data indicates that for a given application or application mix, method X will execute on average 100 times, and on average it takes that method 40 microseconds to execute interpretively and 15 microseconds to execute as generated code (thus 2500 microseconds less overall for generated code), and that it takes on average 600 microseconds to compile the method. Compilation is clearly advantageous, and can be done the first time the method is called, without having to wait for it to execute a certain minimum number of times.

This approach could be taken for all methods in an application, but doing so would require a somewhat intrusive, user-visible feedback loop between execution, off-line processing, and subsequent execution. One of the best things about Java optimization is that it usually takes place in a completely transparent way to the user. Thus, this approach was restricted to just profiling Java 2 Standard Edition core library classes that are used heavily across all applications, meaning those in the `java.*`, `javax.*`, `sun.*`, and `com.sun.*` packages. Since Java applications spend a lot of time in the core libraries, this seems a good target for optimization.

Off-line UNIX scripts processed the profiling information for a given application run and selected those methods that were "wins." This information was reformatted into a file that was opened and read at JIT initialization. Method names fully qualified by package and class are often the same for many initial characters, so a length-segmented list [23] was used as the internal representation, allowing quick look-up. Methods in the list were compiled the first time they were executed. Methods not in the list were treated per the normal default heuristic (deciding *never* to compile methods not in the list would be unwise, as they might execute more frequently than expected within any particular application).

Results with this approach were mixed. In the case of the Java 1.3.0 `javac` command, compilation of the usual source was 9% faster using this approach. (With `javac`, all of the application was subject to this treatment, since the compiler implementation is within the `com.sun.*` namespace.)

In the case of the SPEC JVM98 benchmarks, results with this approach were generally slightly better, but by less than 2%, which is within the margin for error for these measurements.

For the SwingSet2 demo and Tarantella ObjectManager start-up times, there was no real measurable improvement.[12] (As an additional experiment, profiling and selecting all application methods, not just core library methods, produced a 3% improvement for the Tarantella ObjectManager. This was at the outside margin of error for the measurement.)

Why was this approach not more successful? One reason is the collected difficulties in producing fine-grained profiling and timing information for the classic VM. While the generalized JIT performance characteristics as shown in Figures 2 through 4 are accurate, for any individual method the data may be a bit unreliable, thus undermining the calculation of methods that are known to be wins to compile. One lesson of this work is that an accurate profiling has to be designed into a Java VM, not tacked on afterwards!

But the major reason may just be that there is only so much improvement that can be gained by selective compilation heuristics in the classic VM model. After all, selective compilation by itself does not improve the

[12] `javac` was run with a "wins" list from its own profiling. The other programs were run using the "wins" list from Tarantella Objectmanager profiling, which was deemed the most typical of these programs. This work did not get to the question of how best to merge these lists to produce one to use for all applications.

quality of the generated code, nor does it affect any of the other performance-sensitive areas of the VM. After the initial gains brought about by the simple heuristic combining method times and size is realized, trying to get additional gains becomes something like trying to draw water from a stone.

7 Related Work

Selective compilation heuristics have been used in a variety of JIT compiler contexts.

The IBM JIT compiler [17, 18] uses a system based on the number of times a method executes, modified by detection of loops. An HP JIT for embedded systems [10] uses an arbitrary (but configurable) "minimum times" of 2 before compiling. Krintz and Calder use the metric, found by experiment, that the 25% most frequently executed methods must be compiled to produce optimum performance, and then use profiling data to identify those 25% with off-line annotations [7]. The LaTTe JIT did not initially do any selective compilation, but in later work added one based on the number of times a method executes [24].

In recent times, JVM optimization has tended towards more ambitious, dynamic, mixed-mode and adaptive schemes, such as Sun's HotSpot [12], IBM's Jalapeño [1], and Intel's Open Runtime Platform [6, 7]. Selective compilation still plays a role in these JVMs [2], but often based on live profiling, and with a wider choice of compilers and optimization levels to be invoked.

8 Conclusions

The work described here shows how adding simple, transparent selective compilation heuristics to an existing classical Java JIT compiler can significantly improve its performance for short- and medium-lived applications and for start-up time in longer-lived applications.

The strategy used based the decision to compile a method on how many times the method has executed so far (a heuristic for how many times the method will execute for the rest of the application), and on the size of the method (a heuristic for how long it will take to both compile and execute the method). Results show this combined decision strategy is superior to using either one individually, to compiling unconditionally, and to not using the JIT at all.

Profiling investigations produced some counter-intuitive results (correlation between method size and execution time is very weak but still necessary to account for). More complicated selection strategies prototyped either failed to produce useful performance improvements or produced only modest and inconsistent gains.

Although more sophisticated performance schemes are now used by leading-edge Java virtual machines, these results still have relevance. There are platforms where adoption of one of the newer virtual machine technologies is impractical, either for business reasons or due to a variety of technical obstacles. For example, the classic VM's "green threads" implementation option can be used on platforms which do not provide any threads support, such as the SCO OpenServer 5 operating system, whereas most newer VM implementations reply upon the underlying platform providing support for a "native threads" implementation.

In such cases, the selective compilation scheme described here has some distinct advantages. While not completely straightforward to implement, it is not large is size (only about 1% of the sunwjit source was modified or added to, and less than 100 source lines were modified in the classic VM proper). It treats the rest of the JIT as a black box, and thus embodies lower risk than, say, trying to modify the JIT's code generator to produce more optimized code. For a system vendor such as SCO or Caldera, this kind of performance enhancement work provides a good return for the amount of engineering time invested.

9 Acknowledgments

Dries Buytaert, Elaine Siegel, and Mike Davidson contributed useful comments on this paper.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 47–65, October 2000.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM: The controller's analytical model. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO '00)*, December 2000.

[3] M. Arnold, M. Hind, and B. G. Ryder. An empirical study of selection optimization. In *13th International Workshop on Languages and Compilers for Parallel Computing*, August 2000.

[4] Y. C. Chung and M. J. Voss. Summary of the Dynamo '00 panel discussion. In *Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo '00)*, pages 79–81, January 2000.

[5] C.-H. A. Hsieh, J. C. Gyllenhaal, and W.-M. W. Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *Proceedings of the 29th Annual Interational Symposium on Microarchitecture*, pages 90–99. IEEE, December 1996.

[6] Intel Corporation. *Open Runtime Platform*. `http://orp.sourceforge.net`.

[7] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, pages 156–167, June 2001.

[8] S. Liang and D. Viswanathan. Comprehensive profiling support in the Java virtual machine. In *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)*, pages 84–89, May 1999.

[9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[10] G. Manjunath and V. Krishnan. A small hybrid JIT for embedded systems. *ACM SIGPLAN Notices*, 35(4):44–50, April 2000.

[11] P. O'Donnell and S. W. Draper. How machine delays change user strategies. *ACM SIGCHI Bulletin*, 28(2), April 1996.

[12] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proc. of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 1–12. USENIX, April 2001.

[13] M. P. Plezbert and R. K. Cytron. Does "Just in Time" = "Better Late than Never"? In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 120–131, January 1997.

[14] K. Russell and L. Bak. The HotSpot Serviceability Agent: An out-of-process high level debugger for a Java virtual machine. In *Proc. of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 117–126. USENIX, April 2001.

[15] J. L. Schilling. Fail-safe programming in compiler optimization. *ACM SIGPLAN Notices*, 28(8):39–42, August 1993.

[16] Standard Performance Evaluation Corporation. *SPEC JVM98 Benchmarks*. `http://www.spec.org/osg/jvm98/`. Also *SPEC JBB2000 Benchmark*. `http://www.spec.org/osg/jbb2000/`.

[17] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[18] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, pages 180–194, October 2001.

[19] Sun Microsystems, Inc. *The JIT Compiler Interface Specification*. `http://java.sun.com/docs/jit_interface.html`.

[20] Tarantella, Inc. *Tarantella Enterprise 3*. `http://www.tarantella.com/products/e3/`.

[21] S. L. Teal and A. I. Rudnicky. A performance model of system delay and user strategy selection. In *Proceedings of the CHI Conference*, pages 295–305. ACM, May 1992.

[22] C. W. Trigg. Palindromes by addition. *Mathematics Magazine*, 40:26–28, 1969.

[23] T. N. Turba. Length-segmented lists. *Communications of the ACM*, 25(8):522–526, August 1982.

[24] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*. IEEE, October 1999. See `http://latte.snu.ac.kr/` for update on using method run counts in later work.