The Costs and Benefits of Java Bytecode Subroutines

Stephen N. Freund*
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
`freunds@cs.stanford.edu`

September 20, 1998

Abstract

Java bytecode subroutines are used to compile the Java source language `try-finally` construct into a succinct combination of special-purpose instructions. However, the space saved by using subroutines, in comparison to simpler compilation strategies, comes at a substantial cost to the complexity of the bytecode verifier and other parts of the Java Virtual Machine. This paper examines the trade-offs between keeping subroutines and eliminating them from the Java bytecode language. We compare the cost of formally specifying the bytecode verifier and implementing the Java Virtual Machine in the presence of subroutines to the space saved by using them when compiling a set of representative Java programs.

1 Introduction

The Java programming language is a statically-typed general-purpose programming language with an implementation architecture that is designed to facilitate transmission of compiled code across a network. In the standard implementation, a Java language program is compiled to Java bytecode and this bytecode is then interpreted by the Java Virtual Machine. We refer to this bytecode language as JVML.

Since bytecode may be written by hand, or corrupted during network transmission, the Java Virtual Machine contains a *bytecode verifier* that performs a number of consistency checks before code is interpreted. As has been demonstrated elsewhere, the correctness of the bytecode verifier is critical to guarantee the security of the Java Virtual Machine [DFW96]. As a step towards obtaining a correct, formal specification for the verifier, we are currently developing a specification of *statically correct bytecode* for a large fragment of JVML in the form of a type system. This system encompasses classes, interfaces, methods, constructors, exceptions, subroutines, and arrays. While still not the complete JVML, the type system for this subset contains all of the difficult static analysis problems faced by the bytecode verifier.

The most difficult and time-consuming part of our work has been handling subroutines effectively. Subroutines are mainly used to allow efficient implementation of the `try-finally` construct from the Java language. Our general approach for modeling and type checking subroutines is based on a type

*Supported in part by NSF grants CCR-9303099 and CCR-9629754, ONR MURI Award N00014-97-1-0505, and a NSF Graduate Research Fellowship.

```
void f() {
  try {
    something();
  } finally {
    done();
  }
}
```

Figure 1: A method using a `try-finally` statement.

system developed by Stata and Abadi [SA98a]. Even with the knowledge and techniques acquired from their earlier work, extending this type system and proofs to include sound type checking for exceptions, object initialization, and other elements of JVML was challenging.

Verifier implementations based on the current Java Virtual Machine Specification [LY96] have fared no better in handling the complexities which seem to be inherent in the analysis of subroutines. For example, several published inconsistencies and bugs, some of which lead to potential security loopholes, may be attributed to earlier versions of the Sun verifier incorrectly checking subroutines and their interactions with other parts of JVML [DFW96, FM98].

Given the important role of the verifier in the Java paradigm and the many difficulties in both specifying and implementing correct verification methods for subroutines, a natural question to ask is whether or not the benefits of subroutines justify the specification and implementation costs. Eliminating subroutines would not affect the semantics of Java. The only difference would be the compilation strategy for methods which use `try-finally` or other statements currently compiled using subroutines. The most straightforward way to translate a `try-finally` block of Java code into JVML without subroutines requires some amount of code duplication, with an exponential blow up in code size in the worst case. Clearly, removing subroutines from JVML would greatly simplify the verifier, as well as possible implementations of other parts of the Java Virtual Machine, such as type-precise garbage collectors [ADM98]. However, we know of no other study to date quantifying the benefits of subroutines and how much code size is actually saved by using subroutines in typical programs.

In this paper, we examine the impact of subroutines on the formal specification and implementation of the bytecode verifier, on the implementation of other parts of the Java Virtual Machine, and on code size for representative programs drawn from a variety of sources. Our analysis shows that the space saved by using subroutines is negligible and that the theoretically possible exponential increase in size does not occur in the programs studied. The added complexity to the verifier and the Java Virtual Machine far outweighs any benefit of subroutines.

Section 2 describes Java bytecode subroutines, how they are used in the compilation of Java programs, and the difficulties in verifying programs which use them. Section 3 presents measurements on the costs and benefits of subroutines, and Section 4 contains a discussion of these results and some concluding remarks.

2 Bytecode Subroutines

This section describes JVML subroutines and the Java language construct which they were designed to implement, the `try-finally` statement. We also discuss how subroutines may be used to compile

```
Method void f()
     // try block
   0 aload_0                                        // load this
   1 invokevirtual #5 <Method void something()>     // call something()
   4 jsr 14                                         // execute finally code
   7 return                                         // exit normally
     // exception handler for try block
   8 astore_1                                       // store exception
   9 jsr 14                                         // execute finally code
  12 aload_1                                        // load exception
  13 athrow                                         // rethrow exception
     // subroutine for finally block
  14 astore_2                                       // store return address
  15 aload_0                                        // load this
  16 invokevirtual #4 <Method void done()>          // call done()
  19 ret 2                                          // return from subroutine

Exception table:
   from   to  target type
     0     4     8   any
```

Figure 2: The translation of the program in Figure 1 into JVML.

`synchronized` statements and conclude this section by describing the major difficulties in verifying bytecode subroutines.

2.1 `try-finally` Statements

Subroutines were designed to allow space efficient compilation of the `finally` clauses of exception handlers in the Java language. The details of Java exception handling facilities appear in [GJS96]. Subroutines share the same activation record as the method which uses them, and they can be called from different locations in the same method, enabling all locations where `finally` code must be executed to jump to a single subroutine containing that code. Without subroutines, the code from the `finally` block of an exception handler must be duplicated at each point where execution may escape from the handler, or some more complicated compilation technique must be used.

Figure 1 contains a sample program using a `try-finally` statement. There are two ways in which execution may exit from the exception handler. Either the end of the `try` block is reached or an exception is thrown. In both cases, the code in the `finally` block must be executed. Figure 2 shows the bytecode translation for this program. At the two points where execution may exit the `try` block, the `jsr` instruction is used to jump to line 14, the beginning of the subroutine containing the translation of the code from the `finally` block. As part of the jump, the return address is pushed onto the operand stack. This return address is stored in a local variable, and, as in line 19, the `ret` instruction causes a jump back to that address.

Without subroutines, the simplest way to compile this `try-finally` statement is to duplicate the body of the `finally` block at line 4 and at line 9. A translation of the program in Figure 1 that does not use subroutines is shown in Figure 3. In most cases, eliminating a subroutine and duplicating the code in this fashion results in a blow up of code size proportional in the number of calls to the subroutine. However, in the case that one subroutine calls another, the situation is

```
Method void f()
    // try block
   0 aload_0                                    // load this
   1 invokevirtual #5 <Method void something()>  // call something()
    // first copy of subroutine
   4 aload_0                                    // load this
   5 invokevirtual #4 <Method void done()>       // call done()
   8 return                                     // exit normally
    // exception handler for try block
   9 astore_1                                   // store exception
    // second copy of subroutine
  10 aload_0                                    // load this
  11 invokevirtual #4 <Method void done()>       // call done()
  14 aload_1                                    // load exception
  15 athrow                                     // rethrow exception

Exception table:
   from   to  target type
     0     4     9   any
```

Figure 3: The translation of the program in Figure 1 into JVML without using subroutines.

much worse, and not using subroutines results in a blow up in code size exponential in the depth of the nesting of calls. Subroutines nested in this way occur when one `try-finally` statement is placed in the `finally` block of another.

There are other implementation strategies which eliminate subroutines but which may fare better in the case when they are nested. We briefly describe one of the these strategies in Appendix A, but the rest of this paper compares subroutines to only the simple code duplication strategy. As we demonstrate below, the most straightforward translation strategy seems to be suitable for all cases that appear in practice, and these more complex techniques are not required.

2.2 `synchronized` Statements

Subroutines have also proved useful in the compilation of `synchronized` statements. An example of a `synchronized` statement is shown in Figure 4. In the body of the `while` loop, the function must first acquire the lock on object `o` before executing the code guarded by the `synchronized` statement. The lock on `o` must then be released at the end of the synchronized block of code and, also, at any other point at which execution escapes from the `synchronized` statement. In Figure 4, this includes releasing the lock at both the `continue` and the `break` statements, as well as in the event that an exception is thrown while executing the body of the `synchronized` statement. A subroutine may be used to avoid duplicating the code to release the lock on `o` at all escape points in much the same way as they are used in the `try-finally` statement.

2.3 Verifying Subroutines

The flexibility of the subroutine mechanism makes bytecode verification of subroutines difficult for two main reasons:

```
void g(Object o) {
  while (true) {
    synchronized(o) {
      if (test()) {
        continue;
      } else {
        break;
      }
    }
  }
}
```

Figure 4: A synchronized statement with multiple escape paths.

- Subroutines are polymorphic over local variables which they do not use. This is a technical property of the bytecode type system that is enforced by the verifier. Polymorphism is needed to allow a subroutine to be called from program points that differ in the types of values stored in some local variables as long as they agree on the types of all local variables used by the subroutine [LY96].
- Subroutines may call other subroutines, as long as a call stack discipline is preserved. In other words, the most recently called subroutine must be the first one to return. A subroutine may, in some cases, return to its caller's caller, or even further up the stack. This situation may arise from an explicit return statement or implicitly as a result of executing a branch instruction or catching an exception.

Since the program itself stores and manipulates return addresses, the verifier can not assume that polymorphic variables and return addresses are used correctly. Therefore, it must check that the above two properties do in fact hold for each method. Otherwise, a program could potentially make arbitrary jumps by returning to addresses which are not valid return addresses for subroutines on the implicit subroutine call stack. As we will demonstrate below, these checks comprise a major part of both the static type system and soundness proof we have developed for JVML, and they have been difficult to implement correctly in real verifiers.

3 Costs and Benefits of Subroutines

This section describes our measurements and analysis of the costs and benefits derived from adding subroutines to JVML. This includes some qualitative aspects, but we have tried to obtain quantitative results where possible. We first discuss some of the difficulties in specifying and implementing checks for subroutines in the verifier. This discussion will be based mainly on our experiences in developing a type system and soundness proof for a large subset of JVML. We will also touch on the impact of subroutines on other parts of the implementation of a Java Virtual Machine. The third part of this section measures the usage of subroutines in a variety of programs by determining the frequency with which they appear and how much space is saved by using them.

3.1 Specification of the Bytecode Verifier

As part of a larger project to clarify the original Java Virtual Machine specification and to study static analysis techniques for intermediate languages, we have been developing a formal specification for a large subset of JVML in the form of a type system. We are in the process of finishing the soundness proof for this system. This section analyzes the cost of adding subroutines to our type system with some rough numeric measurements followed by a more qualitative assessment. Before presenting the statistics, however, we give a very brief overview of our formal system.

Our work is based on that of Stata and Abadi who originally studied a subset of JVML containing only basic operations and subroutines. Having experienced success extending their work to include object initialization [FM98], we are currently studying a much larger fragment of JVML. This fragment includes:

- classes, interfaces, and objects
- constructors and object initialization
- virtual and interface method invocation
- arrays
- exceptions and subroutines
- integer and double primitive types

While this subset contains only about 25 of the 200 instructions in the whole instruction set, it encompasses most of the difficult static analysis problems encountered by the verifier. These problems include the alias analysis used to check the initialization-before-use of objects [FM98], the "polymorphism" and mostly stack-like use of subroutines, and jumps due to exceptions. There are many other issues that arise, including details concerning class declarations, methods, subtyping, etc. However, these problems are more routine and better understood properties of object-oriented type systems, even if they appear in the somewhat unconventional bytecode language.

In order to remove some of the simplifications that Stata and Abadi made to the subroutine mechanism, we have developed an approach which preserves most aspects of their work but relies on computing more detailed static type information about programs before checking them. Our approach is founded more or less on the same ideas as the work of Hagiya and Tozawa [HT98], although our framework is fairly different from theirs. The statistics below are based on our type system and soundness proof for this subset of JVML. While we have not written out all parts of the proof in complete detail, we have completed a large enough fraction to have a good estimate of its size. The technical details of our work will be presented elsewhere.

We first divide our formal system into three parts, the operational semantics, the static semantics, and the proof invariants. The operational semantics are the rules modeling execution steps in our Java Virtual Machine model. The static semantics are the rules which check whether or not a program is well typed. Only well typed programs would be accepted by a verifier based on our rules.

The proof invariants are used in the soundness proof, which shows that well typed programs do not generate type errors when executing on our Java Virtual Machine model. These invariants relate program execution state to the static type information used by the typing rules. An example invariant is that, given the state of a program, the program counter is within the statically computed bounds of the code array associated with the method currently being executed. Other, more complicated invariants are needed to show the correctness of the static analysis techniques for

Part of Formal System	Judgments	
	For Subroutines	Total
operational semantics	2	45
static semantics	30	90
proof invariants	14	26
total	46	161

Table 1: The number of judgments in each part of the formal system, and the number dealing with subroutines in each part.

Soundness Proof	Number	
	For Subroutines	Total
lemmas	50	120
pages	50–70	150

Table 2: The number and length of the proofs used to prove soundness of our system, and the number required only for subroutines.

subroutines and object initialization. See [FM98] or [SA98b] for more detailed discussions of some of these invariants.

Table 1 shows the number of judgments required to describe each of these elements of our work. That table also shows the number of rules in each section that are affected by `jsr` and `ret` in a non-trivial way. The impact on the operational semantics is minimal since the execution rules for these two instructions are very simple. However, one-third of the type system and half of the invariant judgments are either required or significantly affected by subroutines. Table 2 shows the approximate number of lemmas and number of pages required to show that the invariants are preserved by program execution and that the type system is sound. Roughly 40–50% of the proof may be attributed to the presence of subroutines. From these statistics, it is clear that subroutines are perhaps the largest source of complexity in the type system and verifier specification, and we conjecture that other type systems based on the original Sun specification for JVML will reflect this to some extent as well.

Another, less quantitative metric demonstrating how difficult `jsr` and `ret` are to model is to examine interactions between subroutines and other elements of JVML. For example, subroutines significantly complicate the static analysis for object initialization and constructors at the bytecode level, complicate the checking of exception handlers, and necessitate the introduction of some notion of polymorphic local variables into all aspects of the type system. While there will always be some interaction between various features of a language like JVML, the extent and complexity of these interactions for subroutines is very large. This is partially reflected by the number of typing rules and invariants which depend on subroutines. However, the numbers alone do not fully demonstrate the intricacies that subroutines introduce into our system.

3.2 Implementation Issues

We now briefly discuss some of the Java Virtual Machine implementation issues that arise because of subroutines. Obviously, the bytecode verifier needs to check subroutines properly, but subroutines also lead to trade-offs in the design of other areas of the Java Virtual Machine and static analysis tools for JVML.

Given the difficulty in understanding and specifying type checking for bytecode subroutines, it is not surprising that creating an implementation which handles subroutines correctly has also been difficult. One may see this just by looking at the history of bugs and inconsistencies found in the early versions of the Sun verifier. Examples from the literature on Java security and bytecode verification include a bug in the treatment of `try-finally` statements in constructors [DFW96] and an error by which a subroutine could be used to access an uninitialized object [FM98]. Other studies have reported similar errors or inconsistencies [Qia98, SA98a, SMB97].

Bytecode subroutines also impact how other elements of the Java run-time system and analysis tools may be implemented. Any static analysis technique which must compute or use precise information about control flow or the types of local variables will be affected by subroutines because they introduce polymorphic variables and jumps to addresses stored by the program. For example, conservative garbage collection schemes based on computing reachability from a set of root references can only be replaced by exact collection strategies if the garbage collector is able to determine which local variables contain object references at all appropriate collection points in a program. Difficulties arise in the presence of bytecode subroutines because the types of local variables at a program point may vary with execution history. Agesen, Detlefs, and Moss present one solution to allow computation of precise type information for local variables based on rewriting problematic bytecode sequences [ADM98]. As demonstrated by their work, this is not an easy task. Implementation of other elements of the Java Virtual Machine which must compute control flow information, such as the register allocator for a just-in-time compiler, may also be impacted by subroutines.

3.3 Program Statistics

We now look at the benefits of using subroutines by measuring the reduction in code size obtained by using subroutines over duplication of code for `synchronized` and `try-finally` statements. As previously mentioned, code inflation is typically linear in the number of calls to subroutine, but if subroutine calls are nested in a certain way, there could be an exponential blow up in size. To measure the effect in realistic programs, we analyzed several large programs and libraries, as well as many smaller applets and applications. The specific cases used in our measurements are listed below:

JDK 1.1.5 The Java libraries and development tools from Sun Microsystems. This includes all the standard packages, such as `java.lang` and `java.awt`, as well as the applications `javac`, `javadoc`, etc.

CUP A parser generator for Java, written in Java. Available from `http://www.cs.princeton.edu/~appel/modern/java/CUP`.

toba A Java-to-C translator. Available from `http://www.cs.arizona.edu/sumatra/toba`.

cassowary A general constraint solver implemented in Java and several sample programs using it. Available from `http://www.cs.washington.edu/research/constraints/cassowary`.

pizza A compiler for a super set of Java containing parametric polymorphism, closures, and algebraic data types. Available from `http://www.cis.unisa.edu.au/~pizza`.

applets A collection of approximately 150 public domain applets and applications downloaded from the Internet. These were selected at random from two Web sites, `http://eoe.apple.com` and `http://www.jars.com`, and they include email programs, network monitors, an application to write robot drivers, a text editor, a binary search visualizer, and many others.

Program	Lines	Exception Handlers	Synchronized Statements	Subroutines
JDK 1.1.5	450,000	1300	420	229
CUP	10,500	9	0	0
toba	14,200	34	1	0
cassowary	10,700	37	0	0
pizza	31,600	44	0	1
applets	135,500	590	26	6
total	652,500	2014	447	236

Table 3: Size and number of exception handlers, synchronized statements, and subroutines for the test programs.

Jumps	Occurrences
1	107
2	102
3	15
4	8
5	1
6	1
7	0
8	1
9	0
10	1
total	236

Table 4: The number of jumps to each subroutine from the test programs.

Table 3 contains some basic measurements for these programs. These measurements were obtained by examination of both Java source and the compiled bytecode. For those cases where source was not publicly available (most notably, the JDK), we used simple bytecode analysis to obtain approximate measures. Clearly, subroutines did not appear very often, only 236 times in roughly 650,000 lines of Java code. Most of these appeared in the JDK, where they appeared in about 1–2% of the 11,900 methods. For the applets and applications downloaded from the Web, we found roughly one subroutine for every 20,000 lines of code. For the other test programs, subroutines were virtually nonexistent.

Approximately 45–50% of the subroutines were generated for `try-finally` statements, and the remainder were generated for `synchronized` statements. Approximately 7% of `try` statements included a `finally` block, and roughly 20% of `synchronized` statements utilized a subroutine to release the lock on exits from the guarded code. There were *no* occurrences of a `try-finally` statement inside the `finally` block of another, meaning that there was never more than a linear savings in code size for any method containing a subroutine.

To obtain more specific information about the savings in space gained by using subroutines, we also measured the size of and number of calls to each subroutine. The statistics are summarized in Table 4 and Table 5.

Surprisingly, half of the subroutines were called only once. These are mostly subroutines generated for `synchronized` statements where there is only one escaping path from the `synchronized`

Code Size Saved (bytes)	
minimum	0
maximum	412
median	5
mean	10.4
total	2427

Table 5: The number of bytes saved by using subroutines over code duplication.

block that uses a subroutine, usually a `return` statement. At first, this may seem peculiar since throwing an exception would always be a second way to escape the `synchronized` statement. However, it is common practice for compilers, including `javac`, not to use the subroutine called for other escape points for escapes caused by exceptions. This technique appears to reduce the size of methods by a few bytes and may help minimize the number of exception handlers needed for a method using a `synchronized` statement. In this light, our findings do not seem so unreasonable, and using subroutines for `synchronized` statements does not seem to have very much benefit. In cases like this where a subroutine is called only once, the `jsr` and `ret` statements could simply be replaced by `goto` statements with no space penalty. Very few subroutines were called more than twice.

The greatest savings attributed to a subroutine is 412 bytes, which was seen in a method in which a 206 byte subroutine was called three times. This occurred only once, and there were only two subroutines that saved more than 100 bytes. The majority of subroutines were smaller than eight bytes and saved between zero and sixteen bytes. When computing these numbers, we ignored some small factors, such as potentially needing to use longer bytecode instructions to describe jump offsets inside a method made larger by eliminating subroutines. However, these issues do not significantly affect our measurements.

While subroutines may save a small, but noticeable fraction of space in a few methods, they do not save very much space for any reasonable program given the frequency with which they appear. Also, the size of the method is frequently several hundred bytes or more, meaning that very infrequent code inflation of 400 bytes is not a tremendous loss of space.

To put the measured savings into perspective, we computed the number of bytes saved by subroutines in the JDK as about 2,200 bytes. The size of all the class files for the JDK is about 8.7 MB, making the space saved by subroutines approximately 0.02% of the total size. All symbolic names mentioned in the source code are also stored as strings represented as arrays of bytes in the class files. As such, the word `java` appears roughly 30,000 times in the class files for the JDK. If Sun had stuck with Oak, the original name for the Java language project, the shorter name would have saved 13 times as much space as subroutines do in the compilation of the JDK.

4 Discussion

We have attempted to illustrate some of the costs and benefits associated with `jsr` and `ret` in the Java Virtual Machine. We have described some of the difficulties in type checking and verifying them and, also, in proving properties of type systems for simplified bytecode languages which use subroutines. There is a history of both published and anecdotal evidence indicating that existing verifiers may not be as trustworthy as one would like given their role in the Java Virtual Machine environment. Again, subroutines are a major factor for this. Our measurements of various programs indicate that, while fears of code size inflation are reasonable, it is not encountered in any of the programs studied. Only a handful of all the methods in our test cases were significantly smaller due to subroutines.

Exponential code size increase was not seen in any of our tests. One reason that this theoretically possible case does not appear may simply be that nesting `try-finally` statements complicates the flow of control sufficiently to be considered poor programming practice by most. Another reason may be that such an idiom is not useful in light of resource management techniques common in Java programs. Even if such a case occurs extremely rarely, we do not believe that the ramifications would be very severe. The exponential case may increase the size of a single method substantially if subroutines are not used, but in any reasonable program or collection of classes, the overall increase in size would most likely be inconsequential. There are also implementation strategies which do not experience the same blow up which can be explored if this case ever becomes a problem. We summarize one such strategy in Appendix A.

There are, however, certain issues which must still be seriously explored before we may pass final judgment on whether or not the costs to the complexity of JVML outweigh the benefits of subroutines. Most importantly, our choice of programs may not be the best representative collection. It may be likely that our results will vary with different types of programs. For example, programs which use concurrency heavily may possibly have a higher frequency of subroutines generated by synchronization code. Machine generated Java code may be likely to use subroutines heavily and perhaps in ways not common in our test cases. Examining applications for embedded systems and smart cards may also prove useful.

One argument to use subroutines is that the cost of developing and implementing analysis techniques for them is paid only once, but the cost in code size incurred by removing subroutines will be paid by every Java program executed from this point forward. Therefore, if a small number of researchers can create techniques and implementations which handle `jsr` and `ret` properly in any acceptable amount of time (perhaps several man years), the benefits of subroutines are worth it. While this argument may seem reasonable, it ignores one important point.

Specifically, all programs may already pay some cost because of subroutines. It is likely that subroutines increase the time and space needed to verify programs and, perhaps, affect performance of other parts of the Java Virtual Machine as well. For example, garbage collection must be conservative or it must use exact strategies that incur additional time and space overhead from subroutines, as may other tools such as various parts of just-in-time compilers. Eliminating subroutines may also allow better algorithms, which are currently not suitable for JVML, to be used in these pieces. All of these time and space trade-offs may be on the same level as the measurements made in this paper, but they should be investigated since they are a factor in determining the suitability of subroutines. One direction for future work is to investigate some of these online affects of `jsr` and `ret`.

While this paper has not touched on them, there are undoubtedly backwards compatibility and political issues to address when contemplating changes to JVML as well. Load-time bytecode rewriting has proven useful elsewhere [AFM97], and may be an appropriate solution to any compatibility issues. Using this technique, bytecode using subroutines could be translated into bytecode without subroutines with very simple code transformations when the bytecode is loaded by the Java Virtual Machine. Alternatively, such a translation for class files could be done once offline.

We hope that this study will serve both as a first step towards a full examination of the suitability of bytecode subroutines for Java and as an example of how a single, well-intentioned, but perhaps unnecessary, optimization can have significant impact on many areas of a run-time system like Java. In particular, any aspect of the Java Virtual Machine which must perform static analysis on compiled bytecodes may be drastically affected by subroutines. In addition, the motivation for this study arose from the difficulty of developing a formal type system for JVML. As such, we hope that it demonstrates the importance of formally studying implementation and analysis techniques and understanding their ramifications before they are adopted for a new language.

Acknowledgments: Thanks to John Mitchell, Martín Abadi, and Raymie Stata for several useful discussions and their comments on a draft of this paper, and to Ole Agesen for his comments and the suggestion to use `tableswitch` as a replacement for subroutines.

References

[ADM98] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Proc. ACM Conference on Programming Language Design and Implementation*, June 1998.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proc. ACM Conference on Object-Oriented Programming: Languages, Systems, and Applications*, October 1997.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: from HotJava to netscape and beyond. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pages 190–200, 1996.

[FM98] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. In *Proc. ACM Conference on Object-Oriented Programming: Languages, Systems, and Applications*, October 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[HT98] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. Available from http://nicosia.is.s.u-tokyo.ac.jp/members/hagiya.html. A preliminary version appeared in SIG-Notes, PRO-17-3, Information Processing Society of Japan, 1998.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[Qia98] Zhenyu Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer-Verlag, 1998.

[SA98a] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, January 1998.

[SA98b] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. Research Report 158, Digital Equipment Corporation Systems Research Center, June 1998.

[SMB97] Emin Gün Sirer, Sean McDirmid, and Brian Bershad. Kimera: A Java system architecture. Available from http://kimera.cs.washington.edu, November 1997.

A Alternative Compilation Strategy

This appendix gives a brief overview of a different compilation strategy which we are currently examining. This technique eliminates subroutines but is much less likely than code duplication to suffer from significant inflation in code size. The details of this section will appear in an extended version of this paper. Figure 5 shows the compilation of the program from Figure 1 using this new technique.

This compilation method preserves the notion of call and return but eliminates the need to store addresses. At each location where the subroutine is called, a unique number is pushed onto the stack to identify the caller. The standard `goto` instruction is then used to jump to the subroutine. To

```
Method void f()
   0 aconst_null                             // initialize var 1 to eliminate
   1 astore_1                                //  type conflict when verifier
                                             //  merges predecessors of 18
     // try block
   2 aload_0                                 // load this
   3 invokevirtual #5 <Method void something()>  // call something()
   6 iconst_1                                // push 1 to mark 1st jump
   7 goto 18                                 // jump to start of subroutine
  10 return                                  // exit normally
     // exception handler for try block
  11 astore_1                                // store exception
  12 iconst_2                                // push 2 to mark 2nd jump
  13 goto 18                                 // jump to start of subroutine
  16 aload_1                                 // load exception
  17 athrow                                  // rethrow exception
     // subroutine code
  18 astore_2                                // store caller's marker
  19 aload_0                                 // load this
  20 invokevirtual #4 <Method void done()>   // call done()
  23 aload_2                                 // load marker
  24 tableswitch   1 to 2: default = 46      // compute and jump to return
                                             //  address using look-up table
           1: 10                             // address for 1st jump
           2: 16                             // return address for 2nd jump

  46 return                                  // this should never be reached

Exception table:
   from   to  target type
     2     6    11   any
```

Figure 5: The translation of the program in Figure 1 into JVML using `tableswitch`.

return from a subroutine, the `tableswitch` instruction uses these identifying numbers to compute and jump to the proper return address.

Since polymorphic variables are not available, the compiler must introduce additional local variables to eliminate cases which would have used polymorphism if subroutines were available. In some cases, the compiler must also add local variable initialization code in order to produce programs checkable by the standard verifier. These problems are very similar to ones addressed in [ADM98], where they arise in the context of removing polymorphic variables from subroutine calls to improve garbage collection.

In Figure 5, no additional variables are introduced, but local variable 1 is initialized to a value compatible with the type of exceptions at the beginning of the method. If it were not initialized, the verifier would be unable to determine that local variable 1 contains an exception when execution jumps from line 24 back to line 16. The need for the special initialization code results from the existence of a path to the beginning of the subroutine along which local variable 1 would not usually be initialized.

Despite having much more overhead than simple code duplication, this type of implementation strategy may prove very useful if the theoretically possible exponential blow up in size ever causes a significant problem as a result of removing subroutines. With this strategy, one subroutine may call another without creating additional copies of either subroutine. While we are still in the process of fully evaluating the performance of this method, we believe that it is unlikely that many additional local variables would need to be introduced by this method for such cases. It is much more likely that the simple code duplication strategy will be sufficient for virtually every program.