A Quantitative Analysis of the Performance Impact of Specialized Bytecodes in Java

Ben Stephenson
University of Western Ontario
London, Ontario, Canada
ben@csd.uwo.ca

Wade Holst
University of Western Ontario
London, Ontario, Canada
wade@csd.uwo.ca

Abstract

Java is implemented by 201 bytecodes that serve the same purpose as assembler instructions while providing object-file platform independence. A collection of *core* bytecodes provide critical and independent functionality while a collection of *specialized* bytecodes is meant to improve on the performance of some of the core bytecodes. This study identifies 67 specialized bytecodes and shows the impact of their removal by despecializing them into semantically equivalent core bytecodes.

A detailed analysis of the effects of despecialization on execution efficiency and class-file size was performed. The effects on the SPEC JVM98 Benchmark Suite were analyzed for various subsets of the despecialized bytecodes using three distinct Java virtual machines. When all 67 bytecodes were despecialized, the average slow down across all benchmarks and virtual machines was 2.1 percent, while the single largest performance loss for any one benchmark was 12.7 percent. In some cases, a speedup was observed. An analysis of the impact of despecialization on class file size was also conducted. It was found that the average class file size increased by approximately 6 percent when 67 specialized bytecodes were removed.

This study shows that many of the specialized bytecodes currently in use offer little benefit to either execution efficiency or class file size. Thus, they can be considered as candidates for removal to make room for new bytecodes that will allow for the efficient implementation of new language features, offer significant performance gains or meaningfully reduce class file sizes. Furthermore, these results should be considered by designers who are developing new instruction sets for both virtual machines and hardware processors.

1 Introduction

Java bytecodes are an intermediate representation of an application written in the Java programming language [9]. Sequences of Java bytecodes are generated when an application is compiled and stored into a Java class file. These class files are subsequently executed by a Java Virtual Machine (JVM). The Java Virtual Machine Specification [11] describes 201 bytecodes that can legally reside within a Java class file. While many of these bytecodes provide unique functionality that cannot be mimicked by other bytecodes, some specialized bytecodes exist that can be replaced with one or a small number of the other more general bytecodes. Such specialized bytecodes often make use of implicit information that must be specified as an argument to the more general form.

A variety of studies have been performed to classify various aspects of Java applications at the bytecode level. Some have considered the static nature of the applications [1] while others have considered their dynamic nature [7, 14]. Such studies have also been extended to consider the sequences of bytecodes executed by the application [12]. However, all similar studies that we are aware of have considered the complete Java bytecode language as written, without considering subsets of the instruction set.

This study quantitatively analyses the impact these specialized bytecodes have on the performance of Java applications. Three different virtual machine configurations are employed. Testing is performed on the benchmarks in the SPEC JVM98 suite under nine different despecialization conditions. The impact of removing these specialized bytecodes on class file size is also considered.

1.1 Motivation

While the current Java Virtual Machine (JVM) specification leaves approximately 50 bytecodes unallocated, some virtual machines make use of these bytecodes internally for optimizations. For example, Sun's JVM implementation [10] makes use of 25 of these 'unused' bytecodes to perform *quickening*, a bytecode rewriting optimization that removes unnecessary runtime checks. Furthermore, some hardware implementations of Java have used these bytecodes to implement low level operations such as accessing arbitrary memory locations and accessing status registers within the processor [20]. As a result, these unassigned bytecodes are not always available, despite the fact that their functionality is not defined by the JVM Specification [11]. Consequently, it will be necessary to free up bytecodes in some implementations before any new bytecodes can be introduced. Consider the following reasons why we may want to add support for new bytecodes to the Java Virtual Machine.

1. **Support for new language features:** As new features are added to the Java language it may be desirable to introduce new bytecodes to the Virtual machine so that these features can be implemented efficiently. For example, an efficient implementation of parameterized types in Java that makes use of 2 new bytecodes, `invokewhere` and `invokestaticwhere`, is described in [4].

   It has also been suggested that Java should provide support for multimethods. Like most object-oriented languages, Java uses the dynamic type of only the receiver, in conjunction with the method name, to identify the method to invoke. Languages like Dylan [2] and Cecil [6] extend this concept so that the dynamic types of all arguments are used to determine the method to invoke. Previous research has demonstrated that Java can be extended to support multimethods [8] using existing bytecodes. However, introducing at least one (and up to three) additional bytecodes would provide a significantly more efficient multimethod implementation.

2. **Support for new native types:** The performance problems associated with the use of complex numbers in Java have been well documented. Because complex numbers are often implemented as objects in Java they can be two orders of magnitude slower than equivalent Fortran code [21]. It has been suggested that one technique for narrowing this performance difference would be the introduction of complex numbers as a natively supported Java virtual machine type [5]. Such an implementation would require the introduction of new bytecodes to support the additional primitive type.

3. **Replacement of Common Bytecode Sequences:** Previous research has shown that some sequences of bytecodes occur with great frequency, such as the sequence of 6 bytecodes used to update the value of a field [17]. Techniques such as bytecode idioms [18] and multicodes [16] have been demonstrated to improve runtime performance by replacing a sequence with a new bytecode that provides the functionality of the original sequence in a single bytecode.

Performing despecialization without introducing any new bytecodes also has benefits.

1. **Improved instruction cache performance for interpreters:** When Java bytecodes are being interpreted, it is desirable to have as much of the virtual machine code that performs the operations for the bytecodes in the instruction cache as possible. A smaller set of bytecodes reduces the total number of machine instructions used to implement the Java bytecodes, reducing the possibility of instruction cache thrashing.

2. **Simplification of Java Hardware:** Microprocessors that directly execute Java bytecodes have been developed. If the total number of bytecodes in the Java virtual machine specification were decreased, the size and complexity of such processors would also decrease.

1.2 Organization

The remainder of this paper is structured as follows. Section 2 discusses the various categories of specialized bytecodes that are considered by this study and how each category can be despecialized. Performance results, including both execution time and class file size are discussed in Section 3. Section 4 examines problems that these despecializations cause with respect to the validity of the modified class files. Section 5 identifies areas where this study will be extended in the future. Finally, conclusions are presented in Section 6.

2 Categories

The specialized bytecodes defined by the JVM specification can be separated into several logical categories. Each of these categories is considered in the subsections below.

In the subsections, `<Type>` will represent the data type accessed by the bytecode. In Java, bytecodes often encode their type as a single letter at the start of their name: `a` for object, `f` for float, `i` for integer and `l` for long. In addition, specialized bytecodes often implicitly dictate the value of an argument that would be passed to a corresponding core bytecode. For example, `iload_1` is a specialized version of `iload`. The latter takes a single operand specifying which local variable to load, while the former is hard-coded to refer to local variable 1. Such specialized bytecodes will use the notation `<n>` to refer to the hard-coded operand. Thus, the notation `<Type>load_<n>` concisely identifies an entire collection of bytecodes.

2.1 Load Despecialization

The Java Virtual Machine instruction set includes 25 different load bytecodes that copy the value of a local variable onto the operand stack. Of these 25 bytecodes, the 20 `<Type>load<n>` bytecodes are specialized forms of the 5 general `<Type>load` bytecodes. In each case, the specialized form is replaced with its more general form.

2.2 Store Despecialization

Previous studies have shown that some applications perform more load operations than store operations [7]. As a result we elected to determine the impact of despecializing the load and store instructions separately. Instructions of the form `<Type>store_<n>` where `<n>` is between 0 and 3 are replaced with the more generic `<Type>store` instructions that explicitly pass `<n>` as the operand.

2.3 Integer Constants

Seven bytecodes are dedicated to placing very small integer constants onto the operand stack: `iconst_<n>`, for `<n>` matching 0 through 5 and `iconst_m1` for -1. In each case, the bytecode in question is replaced with the `bipush` bytecode with the appropriate argument byte.

2.4 Non-Integer Constants

There are seven bytecodes that place a non-integer numeric constant value on the stack. These include `dconst_0`, `dconst_1`, `fconst_0`, `fconst_1`, `fconst_2`, `lconst_0` and `lconst_1`. All of these can be replaced by a load from the constant pool if the appropriate value is inserted into the constant pool.

The load from the constant pool may be implemented with an `ldc` bytecode followed by a single argument byte if the value being loaded is a float and it is located at constant pool position 255 or less. Otherwise the `ldc_w` bytecode must be used to load a float constant. The `ldc2_w` bytecode is used to load constants of type long and double.

2.5 Branch Despecialization

The Java Bytecode language provides two separate series of integer comparison and branch bytecodes. The `if_icmp<Condition>` series compares two values on the stack while the `if<Condition>` series compares a single value on the stack against zero. The six `if<Condition>` bytecodes can easily be replaced by the corresponding `if_icmp<Condition>` bytecodes by inserting an additional `iconst_0` bytecode ahead of the comparison so that zero is the top value on the stack.

Further branch despecialization can be performed by the judicious insertion of existing bytecodes. For example, the `if_icmplt` bytecode determines if one integer is less than another. It can be replaced by `if_icmpgt` when a `swap` instruction is inserted before the comparison. Similarly, `if_icmple` can be replaced with `if_icmpge` when a `swap` bytecode is added ahead of the comparison.

2.6 Widening Despecialization

In addition to the despecializations outlined above, it is also possible to despecialize `bipush` to `sipush`. We classify this replacement as a widening despecialization because `sipush` is identical to `bipush` except that the argument to `sipush` is *wider* because it is 2 bytes in size rather than 1. The `sipush` bytecode can be further despecialized to `ldc` or `ldc_w` by adding the integer to the constant pool. While this is not a widening despecialization in the code attribute (because `sipush` requires 2 argument bytes while `ldc` and `ldc_w` require 1 and 2 argument bytes respectively), the number of bytes used to represent the constant value is widened because the 2 byte constant previously stored within the code attribute now occupies 4 bytes in the constant pool.

It is also possible to despecialize all uses of `ldc` by widening to `ldc_w`. Such an operation simply increases the number of argument bytes from one to two, which results in the use of a two byte index into the constant pool even when elements before position 256 are accessed. Note that this despecialization does not increase the number of elements in the constant pool because the values being reference are already present.

Finally, it is also possible to eliminate the `goto` bytecode by using the `goto_w` bytecode. This widening operation increases the number of argument bytes from two bytes to four. Similarly, `jsr` can be despecialized to `jsr_w` by increasing the number of argument bytes by two.

2.7 Additional Despecializations

When all of the despecializations described in the previous sections are performed, at total of 67 bytecodes can be eliminated. A summary of all of the despecialization opportunities discussed to this point can be found in Table 1.

Specialized Bytecode	**Despecialized Bytecode**
aload_<n>	aload
dload_<n>	dload
fload_<n>	fload
iload_<n>	iload
lload_<n>	lload
astore_<n>	astore
dstore_<n>	dstore
fstore_<n>	fstore
istore_<n>	istore
lstore_<n>	lstore
dconst<n>	ldc2_w
fconst<n>	ldc/ldc_w
iconst<n>	bipush
lconst<n>	ldc2_w
if<Cond>	iconst_0 if_icmp<Cond>
if_icmplt	swap if_icmpgt
if_icmple	swap if_icmpge
bipush	sipush
sipush	ldc or ldc_w
ldc	ldc_w
goto	goto_w
jsr	jsr_w

Table 1: Each Specialized Bytecode Tested and its Corresponding Despecialized Form

Additional bytecodes can be eliminated. However, we have not yet implemented or tested the despecializations summarized in Table 2.

3 Performance Testing

A variety of virtual machines were tested on several benchmarks. The virtual machines employed are described in Section 3.1. The benchmarks and despecialization conditions are presented in Sections 3.2 and 3.3 respectively. Sec-

if_icmpeq	if_icmpne goto
if_acmpeq	if_acmpne goto
ifnull	ifnonnull goto
i2c	ldc_w iand
i2b	ishl ishr
i2s	ishl ishr
iinc	iload ldc_w iadd istore
tableswitch	lookupswitch

Table 2: Additional Despecialization Opportunities

	Version / Config	Abbrev.
Sun	1.4.2_02-b03 / mixed mode	Sun
IBM	Jikes RVM 2.3.1 / production	RVM
Kaffe	1.0.7 Interpreter	Kaffe Intrp

Table 3: Virtual Machine Configurations

tions 3.4 through 3.6 describe the performance impact of the despecialization conditions for the benchmarks when executed on different virtual machines. Finally, Section 3.7 describes the impact despecialization has on class file size.

3.1 VM Configurations

Testing was performed on three different virtual machines under a variety of configurations. The virtual machines include Sun's Java Development Kit j2sdk1.4.2_02, Jikes RVM 2.3.1 with Classpath 0.07 and Kaffe 1.0.7. The configurations used for each virtual machine are summarized in Table 3.

This particular set of virtual machines was selected because it contains a wide spectrum of implementation strategies for virtual machines used on desktop computers. The Sun configuration makes use of both an interpreter and an optimizing just-in-time compiler. A Java application is initially executed and profiled with the interpreter. Once it is decided that a method in the application is frequently executed, the JIT compiler is invoked for that method [19].

IBM has taken a *compile everything* approach to the implementation of their research virtual machine (RVM). Every method is initially compiled with the baseline compiler which performs little optimization. Frequently executed methods are identified and recompiled using the optimizing compiler [3].

Kaffe [13] is a comparatively low performance virtual machine. The Kaffe Intrp configuration is a simple bytecode interpreter that performs almost no optimizations.

3.2 Benchmarks

Benchmarks in the SPEC JVM98 Benchmark Suite [15] were tested under the conditions outlined in Section 3.3. These benchmarks are briefly described in Table 4. This benchmark suite was selected because its performance is commonly reported when new optimization techniques are presented. The Standard Performance Evaluation Corporation considers 5 of the 7 applications in the suite to be representative of real applications.

In every case, the benchmarks were executed on a dual processor Dell Pentium III with 768MB of RAM at a clock rate of 600 MHz running RedHat Linux kernel 2.4.20-20.9smp. The benchmarks were executed using the command line interface. Each execution of the benchmark was performed 12 independent times. The values reported were determined by discarding the fastest and slowest run times and computing the average of the remaining values.

For the Kaffe Intrp configuration, benchmarks Comp, Mpeg, MTRT and Jack were executed for input size 1 while benchmarks DB, Jess and Javac were executed for input size 10. The other virtual machine configurations executed all of the benchmarks at input size 100. Smaller sizes were used for the Kaffe Intrp configuration to prevent the execution times from being prohibitively large.

3.3 Despecialization Conditions

Ten different testing conditions were considered. The unmodified libraries provided with each virtual machine were used in conjunction with the unmodified SPEC JVM98 class files to establish baseline execution times. The despecialization conditions are further described in Table 5.

A Java classfile-to-classfile mutator was developed that is capable of performing the despecializations described here. In addition to performing the required substitutions to remove the specialized bytecodes, it also ensures that none of the constraints imposed by the JVM specification such as the size of the code attribute or constant pool are violated. This tool is also responsible for ensuring that the original exception semantics are retained in the modified class file as described at the end of Section 4.

3.4 Kaffe Intrp Results

We begin by considering the performance of the Kaffe Intrp virtual machine. Because this virtual machine performs few optimizations, it was our expectation that this VM would suffer

Benchmark	Abbr.	Description
_201_compress	Comp	Performs the modified Lempel-Ziv compression method which involves finding common substrings and replacing them with codes of variable size
_202_jess	Jess	An expert system that applies a set of rules to a data set in order to solve a series of puzzles
_209_db	DB	Performs a variety of database operations on a memory resident database
_213_javac	Javac	The Java source code to bytecode compiler from Sun's JDK 1.0.2
_222_mpegaudio	Mpeg	An MPEG Layer-3 audio file decompresser. Classes in this application have been obfuscated to remove information that would aid in decompilation
_227_mtrt	MTRT	A multithreaded application that renders an image of a dinosaur
_228_jack	Jack	A parser generator that processes a high-level grammar specification and creates a Java application capable of parsing the grammar

Table 4: Benchmark Descriptions

Condition	Description
Baseline	The original SPEC JVM98 and virtual machine runtime libraries are used.
Load	All load bytecodes are despecialized as described in Section 2.1.
Store	All store bytecodes are despecialized as described in Section 2.2.
Load Store	All load and store bytecodes are despecialized, reducing the set of bytecodes by 40.
Branch	All of the instructions identified in Section 2.5 are despecialized
Constant	All <`Type`>`const_`<`n`> bytecodes are despecialized as described in Sections 2.3 and 2.4.
Constant Widening	All constants are despecialized so that they reside in the constant pool and are accessed using `ldc_w` or `ldc2_w` as appropriate. The widening is also applied to `goto` and `jsr`.
Load Store Constant (LSC)	All loads and stores are despecialized. Constants of non-integer type are moved from the code stream to the constant pool. Integer constants are created using `bipush`.
Load Store Constant Branching (LSCB)	All loads and stores are despecialized. Branch bytecodes are despecialized before constants are despecialized so that any `iconst_0` bytecodes introduced are subsequently despecialized when the rest of the constants are moved from the code attribute to the constant pool.
Complete	All of the despecializations described in Sections 2.1 through 2.6 are performed. The despecializations are performed in such an order so as to ensure that later despecializations do not introduce bytecodes removed by earlier despecializations.

Table 5: Testing Conditions

the greatest performance penalty when despecialization was performed. In particular, we expected to see longer run times when bytecodes of the form <`Type`>`const_`<`n`> were replaced with the appropriate `ldc`, `ldc_w` or `ldc2_w` bytecode. This expectation was based on the fact that the Kaffe interpreter does not perform quickening on loads from the constant pool [10]. It was also our expectation that any despecializations that required the specialized bytecode to be replaced with two or more general bytecodes would suffer a performance penalty because it is necessary to perform an extra iteration of the interpreter loop for each additional bytecode every time the method is executed.

We expected to see smaller performance penalties when Load and Store despecializations were performed. These despecializations require a small amount of additional work because the offset to load from or store to must be loaded out of the code attribute rather than being immediately available.

Figure 1 (located in the appendix) shows the execution time for each of the testing conditions described in Table 5. The bar on the left side of each graph shows the execution time of the unmodified SPEC application and Java library classes while the remaining nine bars each show the execution time of each of the despecialization conditions outlined previously.

It is interesting to note that every benchmark showed a performance improvement for at least 2 of the conditions tested. The largest performance gain achieved was almost 1.5 percent when constant and widening despecializations were performed on Jess. As noted previously, we did not expect to see any performance improvements so additional tests were performed to verify the accuracy of the results. The performance results were continually reproducible. While the cause of this speed improvement is not known, one possibility is that it may result from improved cache or branch prediction performance since the number of unique bytecodes actually executed in the interpreter loop is decreased.

The largest performance penalty was also observed while executing the Jess benchmark. In particular, a performance penalty of just less than 4 percent was observed when complete despecialization was performed. While the fact that there was a performance penalty for complete despecialization came as no surprise, we were surprised to find that the greatest penalty across all applications was a mere 4 percent when 67 bytecodes were despecialized.

Test Condition	Total Performance Change (s)	Weighted Average (Percent)
Base	0.00	0.00
Load	0.75	0.24
Store	0.51	0.17
Load Store	-0.43	-0.14
Branch	-3.86	-1.26
Constant	-0.13	-0.04
Constant Widening	-1.04	-0.34
LSC	-0.70	-0.23
LSCB	-4.46	-1.45
Complete	-7.27	-2.37

Table 6: A Summary of the Performance Statistics for Kaffe Intrp

Test Condition	Total Performance Change (s)	Weighted Average (Percent)
Base	0.00	0.00
Load	-0.26	-0.15
Store	1.36	0.78
Load Store	0.74	0.42
Branch	-2.38	-1.36
Constant	0.43	0.25
Constant Widening	-0.98	-0.56
LSC	0.17	0.10
LSCB	-2.04	-1.17
Complete	-3.89	-2.23

Table 7: A Summary of the Performance Statistics for RVM

Summary results for each testing condition can be found in Table 6. This table shows the total performance gain (with a performance loss indicated by a negative value) in seconds and the total weighted performance change as a percentage of the total execution time. The baseline total execution time for all 7 benchmarks was 304.4 seconds.

Test Condition	Total Performance Change (s)	Weighted Average (Percent)
Base	0.00	0.00
Load	0.63	0.46
Store	-0.43	-0.31
Load Store	-0.02	-0.01
Branch	-1.14	-0.83
Constant	-1.11	-0.80
Constant Widening	-0.83	-0.60
LSC	-0.44	-0.32
LSCB	-2.02	-1.46
Complete	-1.99	-1.44

Table 8: A Summary of the Performance Statistics for Sun

3.5 RVM Results

IBM's RVM showed the largest percentage performance penalties of all configurations considered. While the overall impact of complete despecialization was a performance drop of approximately 2.4 percent, MTRT's performance under complete despecialization was almost 12.7 percent slower than its performance when no despecializations were performed. An examination of Figure 2 (located in the appendix) shows that this performance penalty is a direct result of despecializing branch instructions.

The benchmark which achieved the best relative performance in the RVM configuration was Jess when load despecialization was performed. It showed a speed increase of approximately 2.3 percent, giving an overall performance range of 15 percent. It is our expectation that an optimizer targeted at applications that have been despecialized would be able to reduce the impact of branch despecialization, narrowing the performance range considerably. The performance penalty associated with branch despecialization may also be the result of poor handling of the despecialized code by the RVM optimizer as MTRT does not show such pronounced performance losses when branch despecialization is performed on the other virtual machines.

3.6 Sun Results

When complete despecialization is performed, an overall performance loss of about 1.4 percent is experienced with the Sun configuration as shown in Table 8. Figure 3 (located in the appendix) shows that the worst performing benchmark and despecialization condition was Jack when constant despecialization was performed. This condition showed a performance loss of about 6.3 percent. All of the other conditions except load despecialization showed a similar performance penalty for Jack.

The benchmark that showed the largest performance increase for the Sun configuration was Javac in the LSC despecialization condition. It showed a performance gain of more than 1.7 percent. Similar gains were also observed for this benchmark for the load and load store despecialization conditions.

A total performance range of 8.0 percent was observed across all benchmarks for the Sun virtual machine. While a narrower performance range than that observed for RVM configuration, this range still exceeds the range determined for each of the Kaffe Intrp configuration. This increase in performance range with increased optimization reinforces our belief that despecialization may be hindering optimization of some benchmarks. We believe that an optimizer targeted at class files that do not contain specialized bytecodes should provide better results and decrease the performance spread across benchmarks observed in the faster virtual machines.

3.7 Impact on Class File Size

Despecializing Java class files increases their size. This occurs because some despecialization operations replace bytecodes that contained implicit information with bytecodes that specify this information with additional arguments. Furthermore, other despecializations require that the specialized bytecode be replaced with two or more general bytecodes.

File size comparison was performed in four categories. These categories consist of the files that make up the SPEC application classes, the files that make up the Kaffe runtime library, the files that make up the RVM runtime library and the files that make up the Sun runtime library. In each category, the average file size in bytes is

Category	Original Average	Despecialized Average	Percent Increase	Original Maximum	Despecialized Maximum	Percent Increase
SPEC Classes	2582	2745	6.3	35697	38381	7.5
Kaffe Library	1825	1929	5.7	26172	27743	6.0
RVM Library	2644	2785	5.3	283144	311148	9.9
Sun Library	2700	2870	6.3	122426	124213	1.5

Table 9: A Summary of Despecialization's Impact on File Size

reported for the original files and the files after complete despecialization has been performed. The size of the largest file is also reported before and after despecialization. These results are shown in Table 9.

It is interesting to note that before despecialization is performed, the largest file in the SPEC category is `SPEC/spec/io/TableOfExistingFiles.class`. However, after complete despecialization is performed the largest file is `SPEC/spec/benchmarks/_222_mpegaudio/p.class`. This indicates that the second file made use of specialized bytecodes with larger despecialized forms. The largest file remained the same for the other three categories.

It is also important to note that the file `install.class` was excluded from the values reported for the SPEC category. This choice was made because its large size is primarily due to a specialized attribute that contains compressed data. Inclusion of this file would skew the results for SPEC, resulting in much smaller increases that do not accurately portray the classes that make up the SPEC benchmarks.

4 Generality

The despecialization rules outlined above do not impact the computational power of the Java virtual machine. However, some rules may be difficult to apply because they increase the size of the code for a method or the constant pool. In the worst case this may require methods or classes to be split because the size of the code attribute exceeds 65535 bytes or the number of number of elements in the constant pool exceeds 65534[1]. In practice this is not an issue. The longest method encountered in any of the SPEC application classes or any of the Java virtual machine runtime libraries was 21104 bytes, which expanded to 29697 bytes after performing complete despecialization. Similarly, the largest constant pool encountered before despecialization contained 4302 entries. This increased to 4546 entries after complete despecialization was performed.

[1]The constant pool is restricted to a maximum of 65534 entries rather than 65535 because constant pool index 0 is not valid [11]

The number of argument bytes to conditional branches also poses a potential problem. Because two bytes are used to encode a displacement to the target bytecode, the maximum distance that can be spanned is 32767 bytes forward or 32768 bytes back. As a result, any despecialization which increases the size of the code attribute may result in a situation where it is necessary for a conditional branch to jump too far. This problem does not exist for unconditional branches because a wide form of `goto` exists that is capable of branching to any target location from any point in the method. This problem is never encountered in practice because a code attribute larger than 32767 bytes is never encountered in practice. Before despecialization, the longest branch spanned 9896 bytes which grew to 12940 bytes after all of the despecializations described in Sections 2.1 through 2.6 are performed.

Some despecialization operations also impact the maximum number of elements that need to reside on the stack at one time during the execution of a method. For example, the branch despecializations discussed in Section 2.5 for bytecodes of the form `if<Condition>` potentially increase the maximum number of items on the stack by one. This increase occurs because the zero contained implicitly in the `if<Condition>` bytecode must be placed on the stack before the corresponding `if_icmp<Condition>` bytecode is executed. After complete despecialization, the largest value for max stack is a mere 43 words while the maximum value supported is 65535 words.

The despecialization operations described previously have no impact on the semantics of synchronization or threading. All of the despecialization operations implemented perform their transformation for a bytecode at the point in the method where that bytecode resides. As a result, the new sequence of bytecodes will reside within a monitor if and only if the original sequence of bytecodes reside within a monitor. Since none of the transformations introduced modify any values outside of the operand stack, other than those values modified by the original specialized bytecode, no race conditions are introduced between threads. The fact that an additional value may be added to the operand stack does not present a problem because each thread of execution has its own operand stack that cannot be viewed or modified by other threads of execution.

A small amount of work must be performed to update the exception handlers when a method is despecialized. Since despecialization potentially modifies the position of bytecodes within the method it is necessary to update the `start_pc`, `end_pc`, and `handler_pc` for each exception. These updates ensure that if a bytecode within an exception range is despecialized, all of the bytecodes that replace it also reside within the same exception range, guaranteeing that the semantics of the exceptions are maintained.

The despecializations and modifications described do not reduce the verifiability of the Java class files. Any class file that passed verification before it was modified will still pass verification after modification because the modifications outlined previously expressly preserve the original semantics of the class. This result has been confirmed experimentally. All of the modified class files loaded during the execution of the SPEC benchmarks on Sun's virtual machine passed verification.

5 Future Work

While the results presented here are valuable, they may not accurately predict the impact of the despecialization across all environments where Java is employed. As a result, additional studies will be performed that consider other desktop computing platforms such Windows, Solaris and MacOS running on their corresponding hardware. A future study will also evaluate the impact of despecialization on resource constrained environments such as handheld devices and high performance computing environments such as large Internet servers. Furthermore, the number of benchmarks tested will be extended so that the impacts of despecialization can be observed on a larger set of applications.

Section 2.7 identifies a small number of additional opportunities to despecialize bytecodes. Performance and class file size results will be gathered for these additional cases in order to determine if they are also candidates for removal from the Java bytecode language.

The performance testing presented in this study only considers the specialized bytecodes in broad categories. It will be interesting to profile the impact of each specialized bytecode individually in order to determine if despecializing a small number of byte codes is responsible for most of the performance differences observed. We predict that this will lead to the identification of a limited number of specialized bytecodes that are of significant value and should be retained.

6 Conclusions

The Java Virtual Machine Specification currently defines the functionality of 201 bytecodes. While this leaves approximately 50 bytecodes undefined and available for future expansions, many Java implementations already make use of these bytecodes for optimizations and implementation specific extensions. As a result, using these bytecodes to extend Java's capabilities will cause serious problems for some implementations.

This study has identified 67 bytecodes that that are specialized forms of the remaining core bytecodes. The specialized bytecodes were replaced with their more general counterparts and performance results were computed. Three different virtual machines were tested while running the benchmarks in the SPEC JVM98 Benchmark Suite. It was found that the largest performance penalty occurred when MTRT was executed on IBM's Research Virtual Machine. A performance loss of approximately 12.7 percent was observed for this case. However, this performance loss was atypically large when compared with the results from the other benchmarks. When all 7 SPEC benchmarks were executed on IBM's RVM, the weighted average performance penalty was only 2.4 percent when complete despecialization was performed. All of the other virtual machines tested showed even smaller performance penalties across all benchmarks.

The impact of despecialization on class file size was also measured. It was found that despecializing 67 bytecodes resulted in an average class file size increase of approximately 6 percent over the size of the unmodified files. Any increase in class file size can potentially cause constraints imposed by the Java Virtual Machine Specification such as the maximum method size to be violated. It was found that no classes in the SPEC Benchmark Suite or any of the virtual machine runtime libraries violated any of these constraints after complete despecialization was performed.

These results lead to the conclusion that the specialized bytecodes identified by this study contribute very little to the performance of the Java virtual machine while accounting for one third of the bytecodes defined by the Java virtual machine specification. Previous studies have shown that introducing only 5 multicodes offers larger performance gains than the performance benefits offered by the 67 specialized bytecodes identified in this study [16].

Alternate bytecodes should be identified to replace these specialized bytecodes. Such bytecodes should be used to efficiently implement new language features such as parameterized types and multimethods as well as providing native support for new data types such as complex numbers. Commonly occurring sequences of bytecodes should also be considered for replacement with a single bytecode in order to improve runtime performance.

7 Acknowledgments

The authors would like to thank the Natural Sciences and Engineering Research Council of Canada for their generous support of this research.

8 About the Authors

Ben Stephenson is a Ph.D. candidate studying at the University of Western Ontario. He is currently researching new optimization techniques for improving the performance of virtual machines. At the present time, this research is primarily focused on the Java virtual machine. Ben plans to pursue an academic career.

Wade Holst received his Ph.D. from the University of Alberta in 1999 and is currently an assistant professor at the University of Western Ontario. His areas of research include programming language design and implementation, with a focus on the tension between utility and efficiency in language features like aspects, components, and multimethods. Other research areas include virtual machine implementations and optimizations, language interoperability and meta programming.

References

[1] D. N. Antonioli and M. Pilz. Analysis of the java class file format. Technical Report ifi-98.04, Department of Computer Science, University of Zurich, April 1998.

[2] Apple Computer, Inc. *Dylan Interim Reference Manual*, 1994.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm. *SIGPLAN Not.*, 35(10):47–65, 2000.

[4] J. L. Bank and A. C. B. Myers. Parameterized types and java. Technical Report MIT/LCS/TM-553, 1996.

[5] B. Blount and S. Chatterjee. An evaluation of java for numerical computing. In *ISCOPE*, pages 35–46, 1998.

[6] C. Chambers. Object-oriented multimethods in cecil. In O. L. Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[7] C. Daly, J. Horgan, J. Power, and J. Waldron. Platform independent dynamic java virtual machine analysis: the java grande forum benchmark suite. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 106–115. ACM Press, 2001.

[8] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, and W. Holst. Multi-Dispatch in the Java Virtual Machine: Design and Implementation. In *Proceedings of 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, pages 77–92, 2001.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison–Wesley, Boston, Massachusetts, second edition, 2000.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, 2nd Edition*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

[12] D. O'Donoghue, A. Leddy, J. Power, and J. Waldron. Bigram analysis of java bytecode sequences. In *Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, pages 187–192. National University of Ireland, 2002.

[13] J. Pick. Kaffe.org, 2004. http://www.-kaffe.org.

[14] R. Radhakrishnan, J. Rubio, and L. John. Characterization of java applications at bytecode and ultra-SPARC machine code levels. pages 281–284.

[15] Standard Performance Evaluation Corporation. *SPEC JVM98 Benchmarks*, 2003. http://www.spec.org/jvm98.

[16] B. Stephenson and W. Holst. Multicodes: optimizing virtual machines using bytecode sequences. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 328–329. ACM Press, 2003.

[17] B. Stephenson and W. Holst. A quantitative analysis of java bytecode sequences. In *Proceedings of the 3rd International Conference on the Principles and Practice of Programming in Java*, June 2004.

[18] 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.

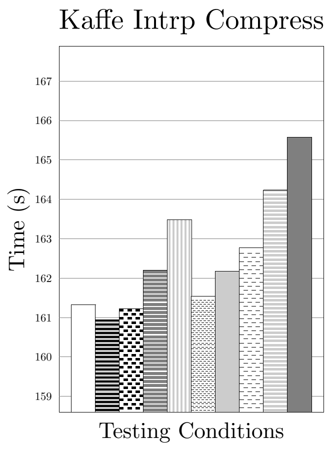
[19] Sun Microsystems. *The Java HotSpot Virtual Machine, v1.4.1, d2: A Technical White Paper*, September 2002.

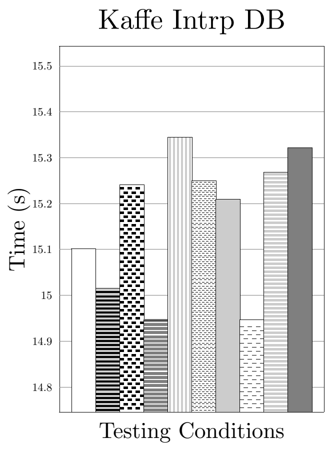
[20] Sun Microsystems, Inc., Palo Alta, California. *picoJava-II Programmer's Reference Manual*, March 1999.

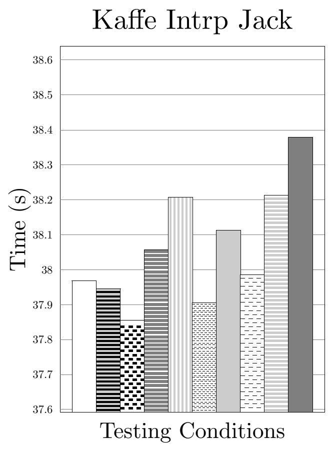
[21] P. Wu, S. Midkiff, J. Moreira, and M. Gupta. Efficient support for complex numbers in java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 109–118. ACM Press, 1999.

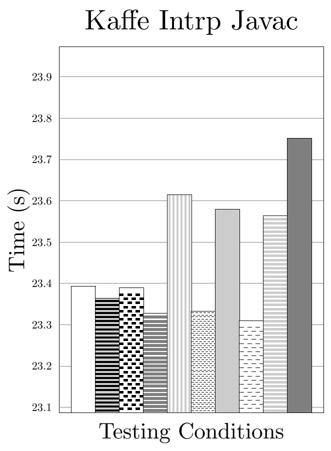
A Performance Figures

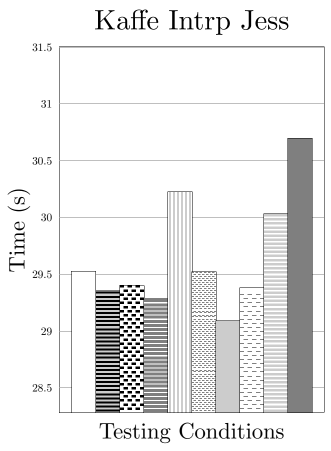
The figures in this appendix show detailed performance results for every combination of virtual machine configuration, benchmark and despecialization condition. Note that each graph has been scaled to maximize the amount of detail presented. As a result, the time scale used on the vertical axis varies from graph to graph.

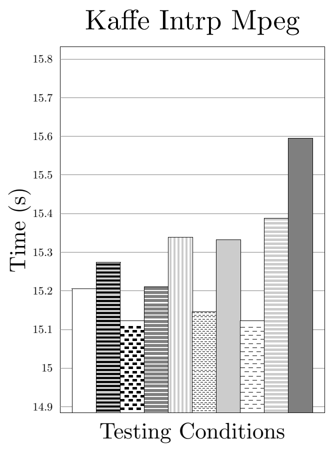


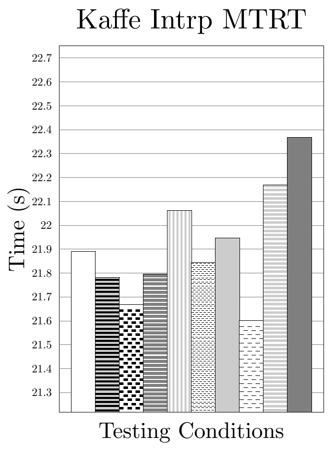












Legend

- □ Base
- ▤ Load
- ▩ Store
- ▤ Load Store
- ▥ Branch
- ▒ Constant
- ▨ Constant Widening
- ▦ LSC
- ▤ LSCB
- ■ Complete

Figure 1: Performance Results for the Kaffe Intrp Configuration

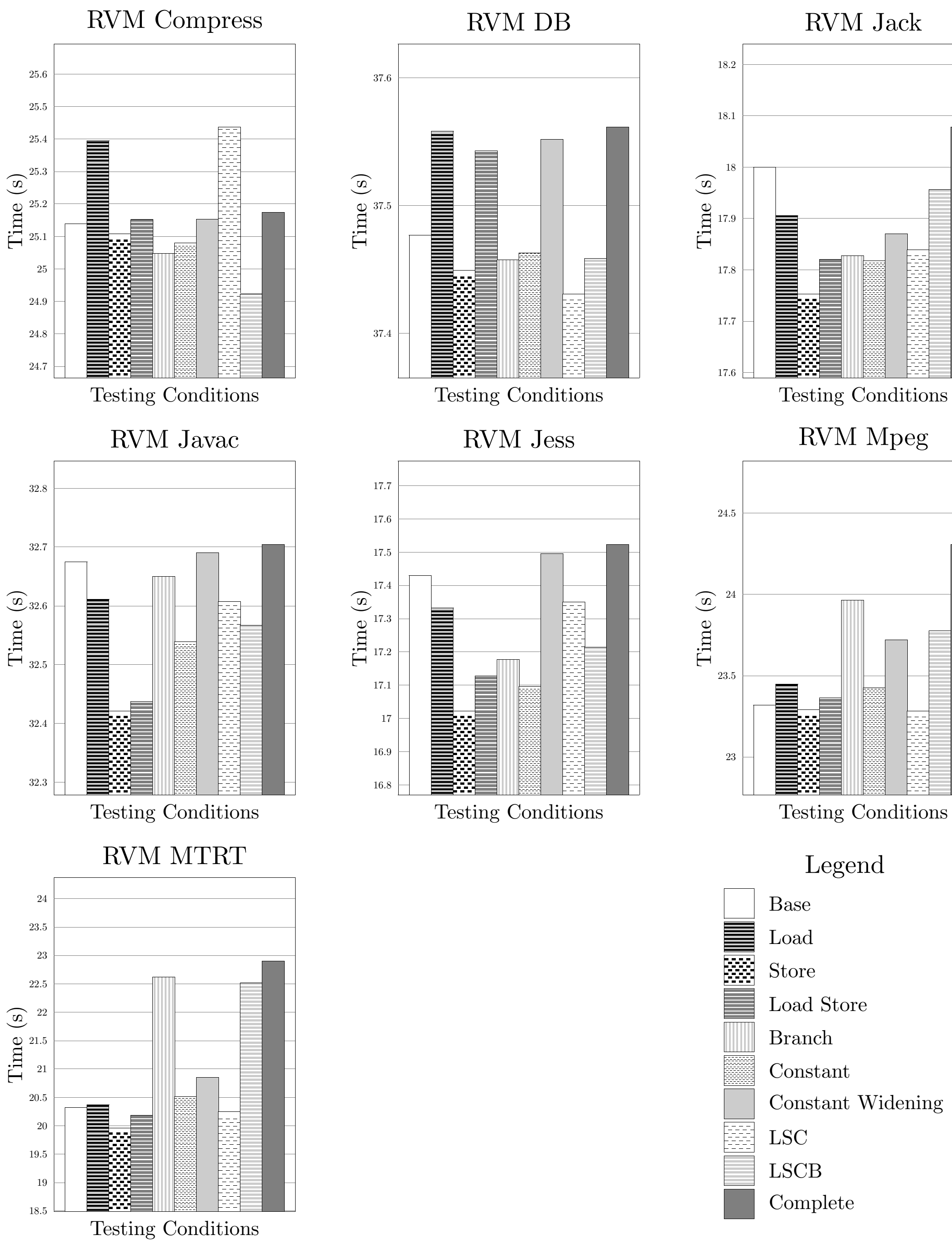


Figure 2: Performance Results for the RVM Configuration

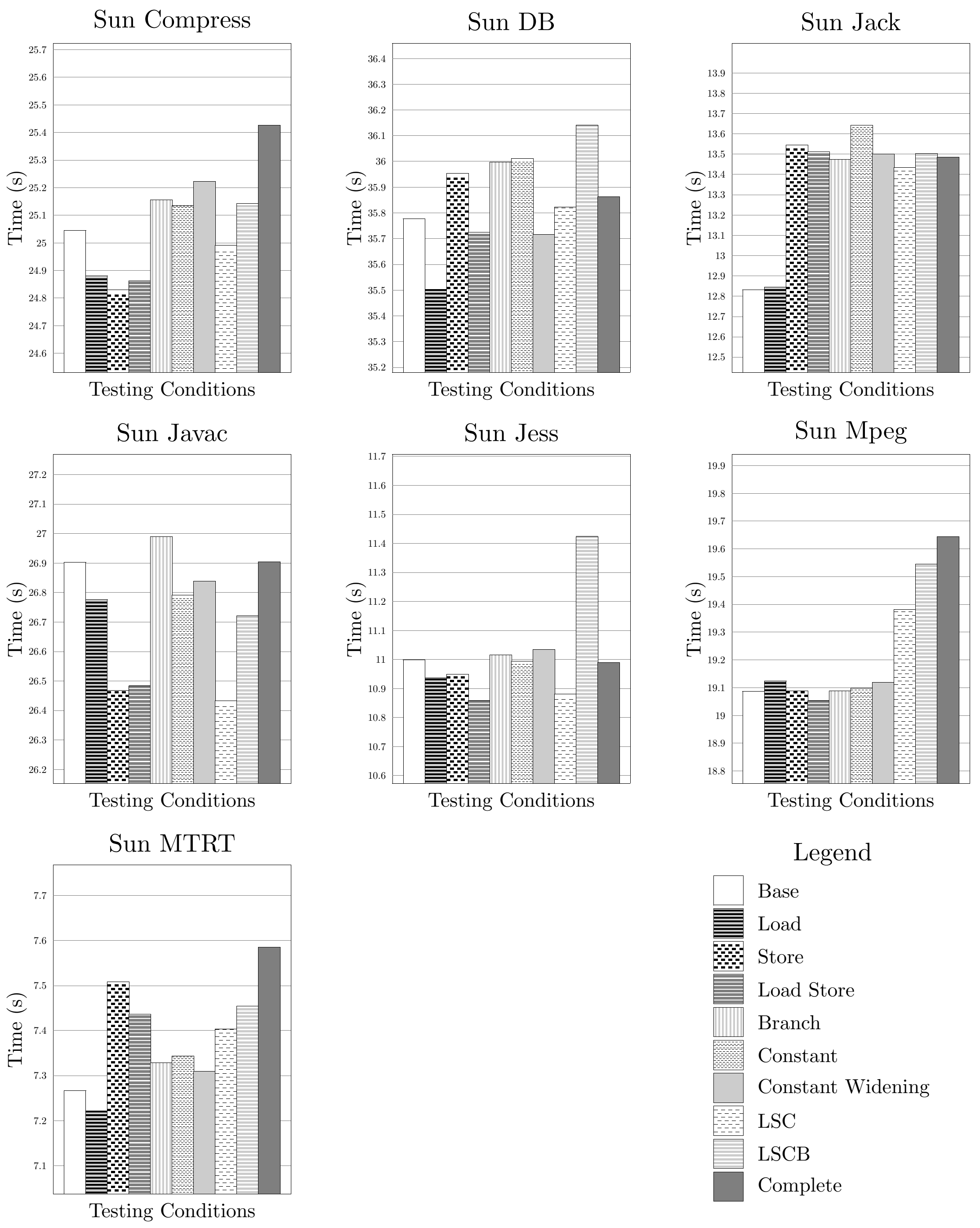


Figure 3: Performance Results for the Sun Configuration