Multicodes: Characterization and Optimization of Java Applications

Ben Stephenson and Wade Holst, University of Western Ontario

Western Western

Abstract

Languages that use a virtual machine (VM), like Java, provide executable-level platform-independence. Unfortunately, this comes with a substantial performance penalty compared to languages that do not employ a virtual machine. This poster introduces *multicodes*, which provide a means of increasing the performance of virtual machines.

Multicodes merge multiple instructions, remove expensive transfers between instructions and allow for both general-purpose optimizations and specialized optimizations that have knowledge of the virtual machine architecture. A variety of Java bytecode metrics are presented. Performance improvements showing that multicodes can provide a significant performance gain are also included.

This poster addresses issues such as multicode identification, multicode optimization, multicode instrumentation, interactions between multicodes and JITs, and other issues related to multicodes.

1 Introduction

Since its introduction in 1995, the Java programming language has come into widespread use for the development of general-purpose applications. The language provides many desirable features that allow applications to be developed with greater ease.

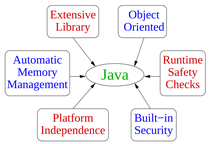

Figure 1: The features provided by the Java programming language.

Unfortunately many of these features also have costs associated with them. In particular, the use of a virtual machine to achieve platform independence adds an additional layer of abstraction and overhead, significantly degrading performance. Extensive efforts have been made to minimize these costs, but substantial performance gains are still possible. A new optimization technique is proposed for Java applications that will help bridge this performance gap without compromising any of the desirable features of the Java language.

1.1 Terminology

Figure 2: The composition of a Java bytecode

Bytecode: A single operation in the machine readable representation of a Java program. A bytecode consists of an *opcode* which uniquely identifies the bytecode and zero or more bytecode specific *operands*.

Instruction: A synonym for bytecode. The terms bytecode and instruction will be used interchangeably.

Opcode: The first byte in an instruction which identifies the operation that will be performed.

Hardware Instruction: A single operation performed by a computer's CPU. One Java bytecode typically maps to several hardware instructions.

Java Micro Instructions (JMI): The code within the JVM necessary to implement the operations performed by a bytecode.

2 The Java Virtual Machine

The Java Virtual Machine (JVM) is a program that reads and executes compiled Java programs. Program execution involves executing a sequence of Java bytecodes. In order to execute one Java bytecode, a sequence of hardware instructions corresponding to that bytecode must be executed on the CPU. Any improvement made in the performance of the JVM results in a corresponding improvement performance of the Java application being executed. Figure 3 outlines the tasks performed by the JVM each time an application is executed. Sequences in red represent overhead incurred by a virtual machine, while those in blue implement user code.

load the Java class files into memory
repeat
fetch next opcode from memory
decode the opcode to determine what JMI sequence should be executed and branch to it
Execute the JMI corresponding to the instruction
until the Java application completes
shutdown the Java Virtual Machine

Figure 3: The algorithm followed by the Java Virtual Machine in order to execute a Java application.

The core data structures used during the execution of a Java method are outlined in Figure 4.

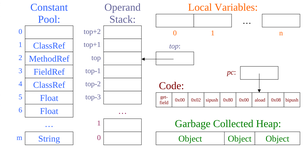

Figure 4: Core JVM Data Structures

Code: The Java bytecodes that define what actions are taken by the method. The *pc* value identifies the bytecode that is currently executing.

Constant Pool: Initialized values that do not change during the execution of the Java application.

Garbage Collected Heap: Memory from which all objects and arrays are allocated.

Local Variables: Temporary locations used for holding values during the execution of a method. Local variables may be accessed in any order.

Operand Stack: Temporary locations used for holding temporary values during the execution of a method. The type of a stack element may change during the execution of a method.

3 Multicodes

To reduce the number of loop iterations that are performed, the Java bytecodes can be considered in groups, rather than individually. The term *multicode* will be used to distinguish these groups of multiple bytecodes from the bytecodes defined by the Java Virtual Machine specification. The use of multicodes will allow the overhead imposed by the Java Virtual Machine to be reduced from one loop iteration per bytecode to one loop iteration per multicode as well as presenting other optimization opportunities.

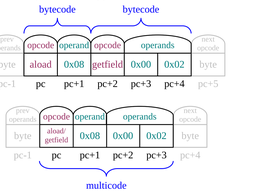

Figure 5: A Comparison of two Bytecodes and a Multicode

4 The Multicode Algorithms

There are five crucial processes in the multicode analysis presented here. Figure 6 provides a dataflow diagram, and the following subsections discuss the processes. A variety of Java applications were used as the data to these algorithms, as summarized in Table 1.

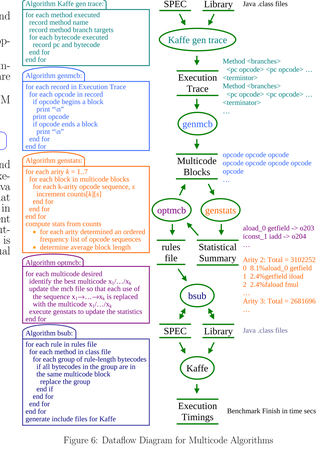

Figure 6: Dataflow Diagram for Multicode Algorithms

4.1 `Kaffe gen trace`: Creating Kaffe Trace Files

- The Kaffe VM was modified to provide:
  - the set of branch targets and handlers within each method
  - a record of each method invoked
  - a record of each bytecode invoked and their order
- Modifications were made to the method invocation routine and each case within the interpreter loop.
- The modified kaffe VM was used to execute each of the applications, producing one .kt file per application. The sizes of these files ranged from 1.1MB for `empty` to 2988MB for `compress`.

Application Name	Description
empty	Program with empty main
hello	A simple "Hello World" program
db	An in-memory database (SPEC JVM)
javac	A java compiler (SPEC JVM)
jess	An expert system shell (SPEC JVM)
jack	A Java parser generator (SPEC JVM)
mpegaudio	An mpeg decoder (SPEC JVM)
compress	A file compressor (SPEC JVM)

Table 1: Java applications used to provide multicode results

4.2 `genmcb`: Creating Multicode blocks

- The .kt files are analyzed to produce multicode blocks, which are sequences of bytecodes where each bytecode in the block can only be reached if the previous bytecode was executed. Some notes:
  - multicode blocks are allowed to have many exit points. (and are thus less strict than the compilation concept of basic blocks).
  - an instruction that is the target of a branch can only occur at the beginning of a multicode.
  - the first instruction of each exception handler starts a multicode block.
  - bytecodes that transfer control out of a method. (i.e. `return`, `ireturn`, etc.) always terminate a multicode block.
  - conditional bytecodes (like `ifcmp`) and transfer bytecodes (like `invokevirtual`) may or may not terminate a multicode.
- Unfortunately it is not necessarily possible to replace every occurrence of a sequence of bytecodes within a Java program with the corresponding multicode.
  - Figure 7 shows an example bytecode stream in which the sequence *istore* $\rightarrow$*iinc* appears twice.
  - The second occurrence of the sequence can be replaced with the corresponding multicode, but the first occurrence cannot, since the second bytecode in that sequence is a branch target.

Figure 7: Bytecode sequence not replaceable by multicode

4.3 `genstats`: Identifying Bytecode Sequences

Multicode blocks can be analyzed to establish numerous interesting statistics:

- The average multicode block length for each application. Figure 8 graphs block length vs. percentage of blocks of that length, and shows the multi-modal nature of multicode block lengths. Multicodes of length 2-3 are common in all applications, as are multicodes of length 9 and lengths 12-15.

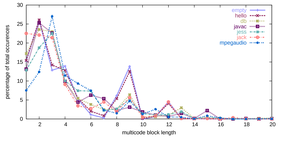

Figure 8: Average multicode block lengths (normalized)

- The number of unique bytecode sequences (of varying arity) actually encountered during dynamic runs. In Figure 9 there are two broad trends, one in which applications peak between arities 3-4, and one in which the peak is beyond the displayed range.

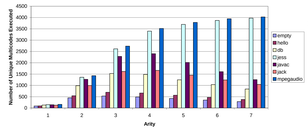

Figure 9: Benchmark Multicode Total and Unique Counts for Various Arities

- The most commonly occurring multicodes of various arities for various applications. Note that $aload_0 \rightarrow getfield$ was the most common 2-arity multicode across all applications considered.

A	app.	multicode	%
2	db	aload_0 getfield	11
	javac	aload_0 getfield	11
	jack	aload_0 getfield	13
	mpeg	aload_0 getfield	6.0
4	db	aload iload_3 iinc aload	3.3
	javac	aload_0 getfield aload_0 getfield	2.7
	jack	aload_0 dup getfield iconst_1	2.9
	mpeg	aload_0 getfield iload aaload	3.0
10	db	aload_0 gfield aload_0 dup gfield dup_x1 iconst_1 iadd pfield iload_1	6.9
	javac	aload_0 gfield aload_0 dup gfield dup_x1 iconst_1 iadd pfield baload	8.0
	jack	aload_0 dup gfield iconst_1 iadd dup_x1 gfield aload_0 gfield gfield	15
	mpeg	faload fmul fadd fstore fload aload_0 getfield iload aaload iload	1.8

Table 2: The most common multicodes for `javac` and `mpegaudio`

- If we substitute the most commonly occurring bytecode sequences with multicodes, how many multicodes are needed to represent P% of the total bytecodes executed? Figure 10 and Figure 11 show results for various applications at arities 2 and 5 respectively. From the figures, the 200 most common multicodes cover 90% or more of the sequences seen for most applications.

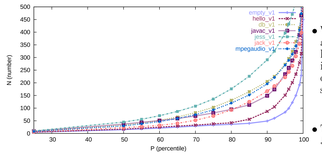

Figure 10: Number of multicodes needed to obtain P% of total bytecodes executed for 2-arity multicodes.

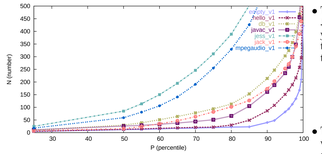

Figure 11: Number of multicodes needed to obtain P% of total bytecodes executed for 5-arity multicodes.

- The level of similarity between two or more applications can determined by establishing a set of applications and determining how many of the multicodes (considering only the N most common multicodes for each application) are common to all applications in the set. Figure 12 shows that `jess` and `mpegaudio` are similar, while Figure 13 shows that `empty`, `hello`, `db`, `javac`, and `jack` are also relatively similar. The graph of all applications shows very little similarity for most arities.

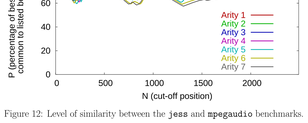

Figure 12: Level of similarity between the `jess` and `mpegaudio` benchmarks.

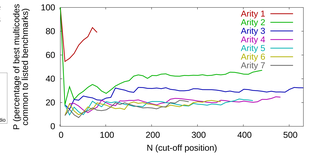

Figure 13: Level of similarity between the `empty`, `hello`, `db`, `javac`, and `jack` applications.

4.4 `optmcb`: Identifying Multicodes

Having identified the set of unique bytecode sequences that appear in various applications, we can now determine which of these bytecode sequences should be converted into multicodes. Some terminology and notation is now introduced that will allow us to do this:

- First, note that the most commonly occurring bytecode sequences do not necessarily make the best multicodes, as will be seen below. The amount of optimization potential, combined with the number of occurrence is a more accurate measure.
- We will denote an arbitrary Java bytecode as x_k.
- We will denote the Java Machine Instructions (JMI) needed to implement an arbitrary bytecode as $<x_k>$.
- Transfers between bytecodes will be denoted using $\rightarrow$.
- The cost to execute a sequence of bytecodes is different than the cost to execute the corresponding multicode because the number transfers has been reduced. This difference in cost will be represented by:

$$\Delta_{x_1/.../x_n} = \mathrm{Cost}(<x_1 \rightarrow ... \rightarrow x_n \rightarrow>) - \mathrm{Cost}([x_1 / .../ x_n \rightarrow]) \quad (1)$$

- When a number of bytecodes are considered as a group, some additional optimizations become possible because there are no longer transfer operations separating the sequences of hardware instructions. These optimizations are discussed in Section 7. The optimized multicode for a sequence of bytecodes will be represented as:

$$\{x_1/x_2/x_3/.../x_n \rightarrow\} \quad (2)$$

- The difference in performance between the original sequence of Java bytecodes and the optimized multicode will be defined as:

$$\Delta_{\{x_1/x_2/...x_n/\}} = \mathrm{Cost}(<x_1 \rightarrow x_2 \rightarrow ... \rightarrow x_n \rightarrow>) - \mathrm{Cost}(\{x_1/x_2/.../x_n \rightarrow\}) \quad (3)$$

- There is no guarantee that converting an arbitrary sequence of Java bytecodes to a multicode will permit any optimizations beyond the reduction in the number of transfers from one bytecode to the next. As a result, the following relationship exists between the costs and Δ values.

$$\mathrm{Cost}(\{x_1/.../x_n \rightarrow\}) \leq \mathrm{Cost}([x_1/.../x_n \rightarrow]) < \mathrm{Cost}(<x_1 \rightarrow ... x_n \rightarrow>) \quad (4)$$

$$\Delta_{\{x_1x_2/...x_n\rightarrow\}} \leq \Delta_{[x_1x_2/...x_n\rightarrow]} \quad (5)$$

- The effectiveness of a multicode cannot be determined from its Δ value alone. The number of times that the multicode will actually execute must also be considered. The following notation will be used represent the number of times that a sequence of bytecodes occurs in a Java program.

$$N_{x_1 \rightarrow x_2 \rightarrow ... \rightarrow x_n \rightarrow} \quad (6)$$

- Equations 7 and 8 show the notation used to represent the number of multicode and the number of optimized multicode substitutions that are possible respectively. These values will always be identical because any sequence of bytecodes that can be replaced with a multicode can also be replaced by an optimized multicode.

$$N_{[x_1/x_2/.../x_n\rightarrow]} \quad (7)$$

$$N_{\{x_1/x_2/.../x_n\rightarrow\}} \quad (8)$$

- Having defined both the improvement achieved for a single substitution and the count of the number of substitutions possible, it is possible to define two ε values as the improvement achieved by performing every possible unoptimized and optimized multicode substitution respectively.

$$\varepsilon_{[x_1/x_2/.../x_n]} = \Delta_{[x_1/x_2/.../x_n]} \times N_{[x_1/x_2/.../x_n]} \quad (9)$$

$$\varepsilon_{\{x_1/x_2/.../x_n\}} = \Delta_{\{x_1/x_2/.../x_n\}} \times N_{\{x_1/x_2/.../x_n\}} \quad (10)$$

- The quantities defined provide sufficient power to accurately compare the effectiveness of different potential multicode sequences based on their ability to improve the performance of a Java application.
- The results presented here use the cost difference between the byte code sequences and the unoptimized JMI for the multicodes. This means that the only speedups come from transfer removal.
  - A constant transfer cost was assigned, whose weight was such t transfers take up 60% of the time for the fastest bytecodes as suggested by previous research on selective inlining.
  - We expect that noticeable improvements will occur when the full optimizations are added. For the results presented here:
- The algorithm for identifying the best multicodes is iterative in nature.
  - The distribution of bytecodes sequences across all arities is computed.
  - The bytecode sequence with the maximum ε value is selected.
  - The multicode block file is modified by replacing all occurrences of the identified bytecode sequence with the multicode byte.
  - Statistics are recomputed for the newly modified file
  - the algorithm recursively calls itself to identify the next best multicode.
- Table 3 shows the top 10 multicodes identified for `mpegaudio` and `javac` using this algorithm. However, due to limitations in the current multicode substitution code, only arities between 2 and 5 were considered during the analysis of best multicodes.

	mpegaudio	javac
1	faload fmul fadd	aload_0 getfield
2	bipush iadd faload aload	aload_0 dup getfield dup_x1 iconst_1
3	aload_0 getfield	iload_2 iload_3 iadd aload_1 arraylength
4	fstore iload iconst_1 iadd bipush	aload iload_3 iinc aload iload_1
5	fstore fload	iconst_0 baload sipush iand ireturn
6	bipush faload	iadd putfield baload bastore iconst_1
7	faload aload iconst_3	iload_3 caload iadd istore_1 iinc
8	faload aload iconst_2	iinc caload castore iinc
9	getstatic iload iinc aaload astore	aload_0 invokevirtual
10	aload_1 iload iinc iload	iadd putfield iload_1 castore aload_0

Table 3: The best 10 multicodes for `mpegaudio` and `javac` (using unoptimized JMI heuristic)

- Figure 14 shows more general results when arities between 2 and 20 are considered. One of the surprises of the research was how often very large arity multicode sequences were deemed the "best" choices.

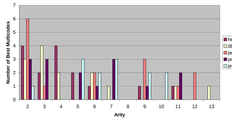

Figure 14: Distribution of Arities for Best Multicodes

4.5 `bsub`: Instrumenting .class files with multicodes

Having identified the best multicodes, there is still a substantial amount of work needed to modify a Java application to benefit from these multicodes, as discussed below.

- Modifications to the Kaffe VM
  - Cases were added to the Kaffe analyzer so that newly created multicodes could be verified.
  - A case was added to the main execution loop for each newly created multicode.
- The `bsub` class-file mutator processes .class files, replacing bytecode sequences with multicodes using the following steps:
  - Read a rules file that describes what multicode substitutions are to be performed.
  - Apply each rule to the class file, replacing every eligible sequence of bytecodes with the multicode.
  - Update all branch offsets and pad bytes used by other bytecodes in the method to take into account the fact that the multicode is smaller than the sequence it replaced.
- Performance results were determined using the following technique
  - One or more multicodes were selected for implementation and placed in a rule file.
  - The rule file was applied every class-file in the Java library and every class-file in the SPEC JVM98 benchmark suite.
  - Updates were performed to allow Kaffe to execute the multicodes
  - The benchmark being tested was executed 12 times. The highest and lowest execution times were discarded and the average of the remaining 10 values was reported.
  - Timing results were performed on a dual-processor 650MHz Dell Pentium III with 512MB of RAM running RedHat Linux 7.3.
- Initial timing results were determined using only 2-arity bytecodes.
  - The bytecode sequence $aload_0 \rightarrow getfield$ occurred most frequently when all of the benchmarks were considered and is shown as *1 Multicode* in Figure 15.
  - The set labeled *5 Multicodes* show the performance of the multicodes $aload_0 \rightarrow getfield$, $iconst_1 \rightarrow iadd$, $aload_0 \rightarrow dup$, $dup \rightarrow getfield$ and $getfield \rightarrow arraylength$.

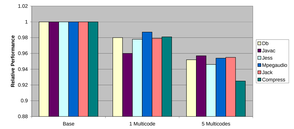

Figure 15: Performance results for various benchmarks

- Timing results were also performed for the top multicodes determined for a benchmark rather than the top multicodes across all benchmarks.
  - Figure 16 shows the performance of `mpegaudio` as each multicode in Table 3 is added.

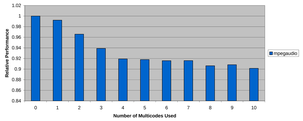

Figure 16: The performance of `mpegaudio` as each new multicode is added

5 Further Optimizations

When a sequence of bytecodes is executed, a transfer of control is performed between each bytecode which prevents many optimizations that span multiple bytecodes from being performed. Because these transfers are removed within an multicode, many additional optimization opportunities become available.

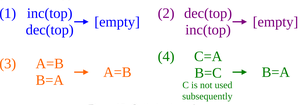

Figure 17: Optimization rules

Consider the bytecode sequence, multicode and optimized multicode for `fstore/fload`$\rightarrow$. This is the eighth most commonly occurring multicode for `jess`.

Bytecode JMI		Purpose
fstore	vars[code[pc+1]] = stack[top]; decrement(top);	Store a value on the stack into a local variable
fload	increment(top); stack[top] = vars[code[pc+1]];	Load a value from the local variables to the stack

Multicode JMI		
fstore/ fload	vars[code[pc+1]] = stack[top]; decrement(top); increment(top); stack[top] = vars[code[pc+2]];	Note that the last offset is pc+2 because pc is incremented between bytecodes.

Optimized JMI Multicode		
fstore/ fload	vars[code[pc+1]] = stack[top]; stack[top] = vars[code[pc+2]];	Rule (2) has been applied to remove unnecessary decrement and increment operations.

Figure 18: The optimization process

If additional data is collected, it may be determined that `code[pc+1]` and `code[pc+2]` both refer to the same variable. When this is the case, one byte can be removed from the multicode's operands, allowing optimization (3) to be applied.

Optimized JMI Multicode		
fstore/ fload	vars[code[pc+1]] = stack[top]; stack[top] = vars[code[pc+1]];	When fstore and fload refer to the same variable, only one operand byte is necessary.

Optimized JMI Multicode		
fstore/ fload	vars[code[pc+1]] = stack[top];	Applying optimization rule (3) reduces the JMI to a single line.

Figure 19: Additional optimizations

- Further research will result in additional optimization rules
  - The sequence $aload \rightarrow dup \rightarrow getfield \rightarrow iconst_1 \rightarrow iadd \rightarrow putfield \rightarrow$ is generated when a field is incremented.
  - The optimization rules outlined in Figure 16 are not sufficiently powerful to reach optimal JMI for this sequence.

6 Future Work

The preliminary performance gains presented on this poster have shown that some performance gains can be achieved through the use of a limited number of multicodes. Additional work must be performed so that more multicodes can be employed.

Using Wide: The wide bytecode is only permitted to precede 12 bytecodes in the current JVM specification. Up to 244 multicodes could be implemented using the the other possible byte values that can follow wide.

Replace Specialized Bytecodes: This technique allows approximately 50 additional bytecodes to be implemented with relative ease. Some work must be removed and replaced with implementation and verification code for the newly created multicodes.

Two-byte instructions: By restructuring the Java class files and Virtual Machine, the size of an opcode can be increased from 1 byte to 2 bytes. This will allow over 65000 multicodes to be implemented but will require extensive modifications to the class files and virtual machine.

Future work with multicodes will also explore their use in conjunction with a JIT compiler. While some JIT compilers already make use of bytecode idioms to perform optimization, we believe that performance gains can be achieved using multicodes because much of the work can be performed ahead of time. This will remove the need to determine when an idiom can be used at runtime and could allow larger numbers of idioms to be employed.

7 Related Techniques

Our work with multicodes builds on previous techniques that have been developed to improve the performance of Java interpreters.

Direct Threading with Selective Inlining: Direct threading converts the bytecode stream to a stream of memory addresses where each address points to the JMI for a bytecode. Selective inlining creates new JMI blocks at runtime that correspond to several bytecodes. While substantial performance gains can be achieved with this technique, it increases the code size considerably. The potential for performing additional optimizations on the newly generated JMI blocks is also limited because this technique is performed at runtime.

Superoperators: This technique was developed to improve the performance of virtual machines executing C programs. New operators are determined from a tree-based representation of the program. These operators are used to build a new virtual machine that is capable of executing the bytecode representation of the program with the help of native assembly language function prologs.

Bytecode Idioms: IBM's Java Just-in-Time Compiler uses "a table of frequently appearing bytecode sequences as idioms to mitigate the inefficiency in code generation caused by stack semantics." Matching is performed at runtime during the compilation process. As a result, the number and length of idioms that can be used is limited by the time available for JIT compilation.

8 Conclusions

Multicodes provide a sound analytical tool for identifying the best possible bytecode sequences to replace with multicodes. While the concept of combining bytecode sequences to remove transfer costs is not new (and in fact has roots as far back as the 1960's) our work provides:

- Consideration of high arity multicodes
- The ability to compare expected performance benefit across arities
- An automated process for determining the best candidate bytecode sequences
- An ahead of time analysis allowing full optimization
- Improved performance for a variety of Java benchmarks

Five algorithms have been developed as part of this research to analyze the bytecode sequences, identify optimal multicodes, update the library and application .class files to make use of those multicodes and determine performance gains. Even using only 5 two-arity bytecodes, speedups were observed in every benchmark under consideration. When 10 multicodes tailored to a specific application were implemented, a performance gain of nearly 10 percent was realized. When the optimizing JMI algorithms are finished, we expect even more dramatic improvements, especially in used in conjunction with despecialization and/or instruction widening.

Acknowledgments

This research would not have been possible without the support of the Natural Sciences and Engineering Research Council of Canada.

Related Readings

Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno JVM. In *Conference on Object Oriented*, pages 47–65, 2000.

M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings ACM 1999 Java Grande Conference*, page 129–141, San Francisco, CA, United States, June 1999. ACM.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Boston, Massachusetts, second edition, 2000.

David Gregg, James Power, and et al. Platform independent dynamic java virtual machine analysis: the java grande forum benchmark suite.

I. Kazuaki. Design, implementation and evaluation of optimizations in a just-in-time compiler.

Paul Klint. Interpretation techniques. *Software- Practice and Experience*, 11:963–973, 1981.

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*, 2nd Edition. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

Ian Piumarta and Fabio Riccardi. Optimizing direct-threaded code by selective inlining. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332, San Francisco, California, 1995.

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

Ramesh Radhakrishnan, Narayanan Vijaykrishnan, Lizy Kurian John, Anand Sivasubramaniam, Juan Rubio, and Jyotna Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–146, 2001.

Standard Performance Evaluation Corporation. Spec java virtual machine benchmark suite. http://www.spec.org/jvm98.

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.