Multicodes: Optimizing Virtual Machines using Bytecode Sequences

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

A virtual machine optimization technique that makes use of bytecode sequences is introduced. The process of determining candidate sequences is discussed and performance gains achieved when applied to a Java interpreter are presented. The suitability of this optimization for JVMs that perform just-in-time compilation is also discussed.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Optimization

General Terms

Performance, Languages

Keywords

virtual machine, bytecode, optimization, Java, JVM, interpreter

1. INTRODUCTION

Languages like Java [1] that use a virtual machine provide a valuable advantage over languages that compile down to native machine code; a virtual machine adds a level of abstraction that allows the same "object" file to be used across multiple platforms. However, like all abstractions, the introduction of a virtual machine incurs efficiency penalties.

This paper introduces a new technique to help lessen the inefficiencies incurred by a virtual machine which we refer to as multicode substitution. It is related to *superoperators* [5], *bytecode idioms* [7], and *selective inlining* [4] but has distinct qualities of its own.

2. TERMINOLOGY

We will use the terms *bytecode* and *instruction* interchangeably in this paper. These terms refer to one of the atomic Java Virtual Machine instructions defined by the JVM Specification [3]. Each bytecode consists of an *opcode*, which uniquely identifies the instruction, and zero or more instruction-specific *operands*. Each bytecode has a collection of statements associated with it that implement the bytecode in question. These will be referred to as *Java Micro Instructions* and will be abbreviated *JMI*.

Between each instruction, the virtual machine must perform a *transfer* to the next instruction. Previous research has shown that for simple bytecodes, almost all of the execution time is spent performing transfers as opposed to executing the bytecode itself [4].

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

`iload_1`	`istore_1`	`dup`
++top; st[top] = vars[1];	vars[1] = st[top]; –top;	++top; st[top] = st[top-1];

Table 1: Example JMI for simple bytecodes

`istore_1/iload_1`	`dup/istore_1`

vars[1] = st[top];
–top;
++top;
st[top] = vars[1];
	++top;
	st[top] = st[top-1];
	vars[1] = st[top];
	–top;

Optimized JMI	*Optimized JMI*
vars[1] = st[top];	vars[1] = st[top];

Table 2: Example unoptimized (top) and optimized (bottom) JMI for multicodes

3. MULTICODES

A *multicode* is a sequence of bytecodes that are treated as an atomic unit. No transfer operations are performed within a multicode. However, transfers of control still exist between multicodes. The *arity* of a multicode is defined to be the number of bytecodes in the sequence.

An *unoptimized multicode* is one whose JMI is simply the concatenation of the JMI for each participating bytecodes. Note that the use of an unoptimized multicode still results in a performance gain because the number of transfers executed is reduced.

When several bytecode's JMI are concatenated some operations may become redundant. For example, if one bytecode ends with an operation such as incrementing the stack pointer and the next bytecode immediately undoes this operation both operations can be removed. An *optimized multicode* is one whose JMI has been optimized to remove these kinds of unnecessary operations. Table 1 shows the JMI for three commonly occurring bytecodes. Table 2 shows the optimized and unoptimized JMI for 2 multicodes constructed from the bytecodes shown in Table 1.

3.1 Multicode Identification

The number of possible k-arity multicodes quickly becomes excessive as k increases. For example, a rough approximation of the number of 2-arity multicodes is the number of bytecodes squared, or $201^2 = 40401$. However, this is an upper bound, because many bytecodes cannot legally follow each other because the types of the stack values provided

and required are incompatible. For example, less than 28000 2-arity sequences are type correct (approximately 70 percent) and less than half of the 8.1 million 3-arity sequences are type correct. Furthermore, testing has also shown that the number of distinct multicodes that actually occur in any application is relatively small. For example, no application executed more than 1430 distinct 2-arity multicodes or 2736 3-arity multicodes.

The existing Java Virtual Machine specification leaves 52 bytecodes undefined. However, specific implementations of the virtual machine may make use of these bytecodes suggesting that fewer than 52 bytecodes are still available. Several other techniques exist for increasing the number of available bytecodes including increasing the size of each opcode from 1 byte to 2 bytes (either at class load time or by storing 2-byte opcodes in a class file), despecializing the class file to remove uses of specialized bytecodes such as `iload_0` and allowing the `wide` bytecode followed by a bytecode not normally permitted to follow `wide` to represent a multicode.

Regardless of the approach taken to increase the number of bytecodes available for use by multicodes, there will still generally be more multicodes used than can be implemented. This suggests that it is necessary to determine which multicodes are best to implement. As an initial approximation, one can simply count the number of times each multicode executes across a variety of (hopefully representative) applications and conclude that the multicodes that occur with greatest frequency should be implemented. A more careful analysis must consider the amount of optimization that can be performed on each candidate sequence. An algorithm is presented which allows the performance benefits of candidate sequences to be compared based on both their frequency of occurrence and optimization potential.

3.2 Multicode Optimization

As shown in Table 2, some bytecode sequences can provide a significant amount of room for optimization when transfers are removed. While few commonly occurring bytecode sequences offer such substantial optimization potential, most sequences offer some optimization potential when the bytecodes in the sequence are related to each other in some way (for example, the first bytecode places a value on the stack which is used by the second). Several optimization rules are presented which can be used to optimize multicodes. Furthermore, these rules take the semantics of the operand stack into consideration. This allows optimizations to be performed which could not be performed by a general purpose optimizer that is not aware of the stack semantics being applied to the array used to represent the operand stack.

3.3 JIT Compilers and Multicodes

Most performance oriented JVMs rely on JIT technology, which replaces the software fetch-decode-execute loop used to execute the bytecodes with chip-specific, optimized, low-level code for the most computational intensive areas of the program. All optimizations that are identified for use in multicodes are also available for use in JIT optimizers. Since much of the multicode analysis can take place off-line, JIT optimizers will benefit from the identification of sequences of bytecodes that can be efficiently optimized. Since JITing occurs at run-time, their optimizations are often less complete than those in a more traditional optimizing compiler. By identifying and optimizing sequences off-line, JIT optimizers can spend more time on other forms of optimization.

Some JIT compilers, such as that included with IBM's Jikes Research Virtual Machine, make use of bytecode idioms during their compilation process. By performing multicode substitution ahead of time the need to perform matching of the idioms to the code stream at runtime is removed, improving performance.

3.4 Multicode Results

In order to prove the utility of multicodes, we have obtained two kinds of results. First, we establish numerous multicode metrics, including an identification of the most common k-arity multicodes, and, more generally, a complete ordering of multicodes based on frequency of occurrence for arities from 1 to 7. Additional metrics include the percentage of multicodes common to all benchmarks, an identification of multicodes dominant across all benchmarks, and statistics on average and maximum multicode block size. This work expands on previous studies which have computed bytecode metrics [6, 2], without considering the order in which the bytecodes were executed.

Second, we present timing results demonstrating the relative improvement of a JVM with multicode support compared with one without multicode support. It was found that adding only five commonly occurring multicodes (`aload_0/getfield`, `iconst_1/iadd`, `aload_0/dup`, `dup/getfield` and `getfield/arraylength`) to the Kaffe virtual machine resulted in speedups for Java applications ranging between 4.3% (*javac*) and 7.5% (*compress*) depending on the application in question. Further results, as well as the full text of the poster, can be found on the web at http://www.csd.uwo.ca/~wade/Meta/Multicodes.

4. ACKNOWLEDGMENTS

The authors would like to thank NSERC for their generous support of this research.

5. REFERENCES

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

[2] D. Gregg, J. Power, et al. Platform independent dynamic java virtual machine analysis: the java grande forum benchmark suite.

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

[4] I. Piumarta and F. Riccardi. Optimizing direct-threaded code by selective inlining. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[5] T. 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.

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

[7] T. Suganuma, T. Ogasawara, et al. Overview of the IBM java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.