# Advancements in Multicode Optimization

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

In previous work, we have shown that multicodes can be used to improve the performance of Java applications. We extend that work by both implementing more multicodes and considering multicodes of greater length. This has resulted in significantly larger performance gains than those that have been achieved previously for some benchmarks.

## Categories and Subject Descriptors

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

## General Terms

Performance, Languages

## Keywords

virtual machine, bytecode, optimization, Java, interpreter

## 1. INTRODUCTION

The Java programming language [1] is being widely used for the development of general purpose applications. Unfortunately, many of the features that make Java popular also hinder its performance. In particular, using a virtual machine to achieve platform independence significantly degrades application performance.

One optimization technique that can be used to help overcome this performance penalty is multicode substitution [4]. This optimization reduces the amount of overhead imposed by the virtual machine by considering some Java bytecodes in groups rather than individually.

Previous work has only considered the implementation of as many as 10 multicodes, each restricted to a maximum length of 5 bytecodes. This work significantly extends the previous study by increasing the number of multicodes implemented to 48 and the maximum length of each multicode to 20 bytecodes. Performance testing has shown that meaningful performance gains can be achieved on Java interpreters for some benchmarks. When the benchmarks _201_compress and _222_mpegaudio are considered from the SPEC JVM98 Benchmark Suite [3], it was found that performance gains exceeding 20 percent could be achieved.

## 2. MULTICODE OVERVIEW

A traditional Java interpreter executes a Java class file using a loop that fetches the next bytecode out of memory, decodes the bytecode using a switch statement and then

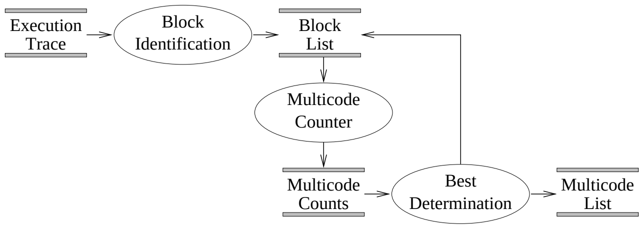

**Figure 1: Multicode Identification Algorithm**

executes a codelet that performs the operations required by the bytecode. While techniques such as threading [2] have been developed to perform decoding more efficiently, these three basic operations are still fundamentally present in the interpreter.

The fetch and decode operations are overhead imposed by the interpreter. As a result, it is beneficial to remove these operations whenever possible. Multicodes accomplish this goal by converting a sequence of Java bytecodes into a single multicode which combines the functionality of the sequence into one new bytecode. Consequently, the number of fetch-decode-execute (FDE) cycles required to accomplish the task is reduced from one FDE cycle per bytecode in the sequence to one FDE cycle per multicode. As our performance results show, this reduction in FDE cycles is a significant saving.

## 3. MULTICODE IDENTIFICATION

The sequences of bytecodes executed by the JVM vary greatly from application to application. As a result, we chose to profile the applications being tested. This was a non-trivial undertaking because it was necessary to keep a record of both the bytecodes that were executed *and the order in which they were executed.* To achieve this goal, it was necessary to save each bytecode as it was executed, which resulted in data files up to 3 gigabytes in size. In addition, we also saved sufficient information to identify the start of each basic block within the list of bytecodes executed.

An iterative algorithm was used to determine which sequences occurred with greatest frequency. While the iterative algorithm has a large computational complexity, it ensures that each bytecode executed only contributes to the count of a single multicode. This constraint is important because each bytecode can only be part of a single multicode. If a bytecode were included in the count of several multicodes, the count for all multicodes except the first would be artificially inflated since only the substitution for the first multicode could actually be performed.

Figure 1 shows the process followed in order to identify the multicodes used during testing. Our algorithm begins by processing the execution trace generated when the application was profiled and separating the bytecodes by basic block. Performing this transformation on the data ensures that we do not identify multicodes that span basic blocks. This constraint is important because it is impossible to branch to the middle of a multicode at the Java bytecode level. In the future, we hope to reduce the strength of this constraint from basic blocks which have only one entrance point and one exit point to multicode blocks which have only one entrance point but may have many exit points.

The multicode counter identifies all sequences of bytecodes of lengths 2 through 20 within each block. Each time a sequence is identified, the count corresponding to the sequence is incremented.

Once all of the sequences have been counted, the best sequence is determined. This is done by assigning a score to each sequence which is the product of its count and its length. We use this score because it represents the total number of bytecodes that will be impacted by the substitution. In the future, the scoring system will be expanded to consider the optimization potential of each sequence, allowing us to make better choices about what sequences to implement as multicodes.

The best sequence is added to the list of multicodes identified. All occurrences of the sequence are removed from the block list. By performing this removal, it is guaranteed that bytecodes used to determine the current multicode will not be counted during the determination of subsequent multicodes. The multicode identification process is repeated until the desired number of multicodes have been determined.

## 4. MULTICODE PERFORMANCE

Previous work has shown that multicodes can be used to achieve performance gains of up to 10 percent. By implementing additional multicodes and extending the identification algorithm to consider multicodes up to 20 bytecodes in length, we have been able to achieve significantly better results for two of the benchmarks in the SPEC JVM 98 Benchmark Suite. In particular, modifying the Kaffe Java interpreter to implement 48 multicodes tailored to each application has shown a performance gain of 21.3 percent for _201_compress and 24.2 percent for _222_mpegaudio. Testing was conducted on a dual processor Pentium III with 768 MB of RAM clocked at 600 MHz running RedHat Linux kernel 2.4.20-20.9smp. Input size one was used during testing.

## 5. FUTURE WORK

A comparison of the lists of multicodes for each of the benchmarks revealed that _201_compress and _222_mpegaudio have higher average multicode lengths than the other benchmarks. Furthermore, _201_compress has more than twice as many multicodes of length 10 than the remaining benchmarks while _222_mpegaudio has almost four times as many. As a result, we believe that it would be beneficial to develop techniques that will allow us to identify longer multicodes without decreasing the number of occurrences of the multicode selected. Once these techniques are implemented, we hope to achieve similar performance gains for the other benchmarks in the SPEC JVM 98 suite.

One technique that will allow us to increase average multicode length is despecializing the Java class file when identifying and substituting multicodes. This will permit bytecodes of the form ⟨Type⟩load/store_⟨n⟩to be replaced with the more general ⟨Type⟩load/store bytecode with the appropriate argument. The transformation will increase the number of times a multicode can be substituted since bytecodes with different implicit arguments will no longer be considered distinct. Previous work has shown that this type of despecialization has minimal impact on the runtime performance of the application [5].

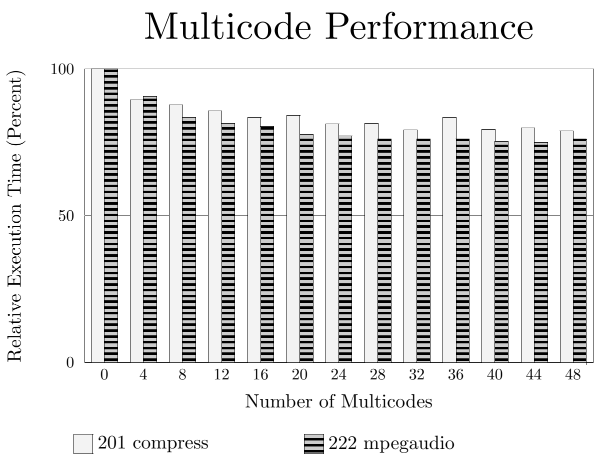

**Figure 2: Relative Performance Change as the Number of Multicodes Increases**

Permitting multicodes to span basic blocks, as described in Section 3 will also increase average multicode length which may further improve the performance of some benchmarks.

## 6. CONCLUSIONS

Increasing the number of multicode substitutions performed as well as increasing the maximum length allowed for a multicode has resulted in a meaningful increase in runtime performance for two benchmarks when compared to the results achieved in previous studies [4]. These gains were achieved for those benchmarks that have the largest average multicode lengths. Techniques such as despecialization and allowing multicode substitutions that cross basic blocks will make it possible to increase the average multicode length for the remaining benchmarks in the hope that it may be possible to improve the performance of these benchmarks as well.

## 7. REFERENCES

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

[2] P. Klint. Interpretation technqiues. *Software–Practice and Experience*, 11:963–973, 1981.

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

[4] 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.

[5] B. Stephenson and W. Holst. A quantitative analysis of the performance impact of specialized bytecodes in java. In *Proceedings of CASCON 2004*, October 2004.