Advancements in Multicode Optimization

Ben Stephenson and Wade Holst
University of Western Ontario
ben@csd.uwo.ca wade@csd.uwo.ca

Abstract

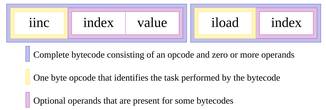
Previous work has shown that multicodes can be used to improve the performance of Java applications [2]. We extend that work by both implementing more multicodes and considering multicodes of greater length. This has resulted in greater performance gains. In particular, it has been possible to achieve gains in excess of 20 percent for the compress and mpegaudio benchmarks while other benchmarks have shown performance gains of approximately 10 percent. It has been observed that those benchmarks that showed the greatest performance gains also have the longest average multicode length. Areas of future work that are expected to increase multicode length are also presented.

1 Organization

Section 2 defines terms that are used throughout the remainder of the poster. An overview of the general concepts related to multicodes are presented in section 3. This is followed by a discussion of the algorithm employed to identify which sequences should be implemented as multicodes in Section 4. Section 5 outlines how multicode substitution is performed while Sections 6 and 7 discuss the testing environment used and the performance results that have been achieved. Areas of future work are identified in Section 8. Finally, Section 9 summarizes our work and presents conclusions.

2 Terminology

Java bytecodes are used to describe the functionality of a Java application to the JVM. A bytecode consists of an opcode which identifies its function and zero or more operands as shown below.



Each Java bytecode is implemented by a codelet within the virtual machine that modifies the appropriate data structures. We will refer to this codelet as the Java Micro Instructions (JMI) for a particular bytecode.

```
switch (code[pc])
{
  case iload:
     top++;
     stack[top] = locals[code[pc+1]];
     break;

  case iadd:
     stack[top] = stack[top] + stack[top−1];
     top−−;
     break;

     ...
}
```

Java Micro Instructions for iload
Java Micro Instructions for iadd
Operations performed by the virtual machine

3 Multicodes

A Java interpreter executes each bytecode in the Java class file as a distinct unit. A loop is used to repeatedly fetch bytecodes, determine what JMI to execute and execute the JMI. This loop imposes a large amount of overhead. As the diagram shows, many operations must be performed to reach the JMI for a bytecode that often consists of only a few lines of code.

```
for (;;)
{
   pc = npc;
   npc = pc + length[code[pc]];

   switch (code[pc])
   {
      case iload_2:
         top++;
         stack[top] = local[2];
         break;

      case iload:
         top++;
         stack[top] = locals[code[pc+1]];
         break;

      case iadd:
         stack[top] = stack[top] + stack[top−1];
         top−−;
         break;

      case istore_2:
         locals[2] = stack[top];
         top−−;
         break;

         ...
   }
}
```

Overhead imposed by the virtual machine
Codelets that implement the functionality of each Java bytecode

Techniques do exist that can reduce the amount of overhead imposed by this loop. However, regardless of the technique employed, it is still necessary for a Java interpreter to repeatedly fetch the next bytecode, determine what JMI to execute and then execute the JMI corresponding to the bytecode.

Multicodes help reduce this overhead by merging several bytecodes into one multicode. When the multicode is executed, the amount of overhead required to execute the entire multicode is the same as the amount of overhead to execute one bytecode. As a result, the total amount of overhead is reduced.

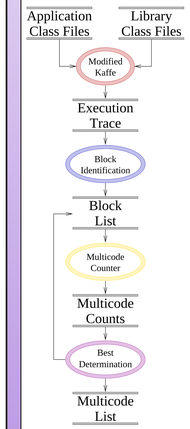
The JMI for the multicode is formed by merging the JMI for each of the constituent bytecodes. The JMI for the multicode iload_2/iload/iadd/istore_2 is shown below.

```
case iload_2_iload_iadd_istore_2:
   top++;
   stack[top] = local[2];
   top++;
   stack[top] = locals[code[pc+1]];
   stack[top] = stack[top] + stack[top−1];
   top−−;
   locals[2] = stack[top];
   top−−;
   break;
```

JMI for the entire multicode iload_2/iload/iadd/istore_2
JMI contributed by iload_2
JMI contributed by iload
JMI contributed by iadd
JMI contributed by istore_2

4 Multicode Identification

Profiling was used to determine what sequences of bytecodes should be replaced with multicodes.



The Kaffe virtual machine was modified to collect information about the application as it runs.
- The name of each method invoked
- Each bytecode executed
- The program counter value for each bytecode

The execution trace is processed to store the information necessary to identify the sequences of interest more concisely. The output format simple lists each bytecode executed, using newlines to indicate the seperation between basic blocks.

```
for i = 2 to 20
   for each line in the file
      for each opcode sequence of length i
         let 's' be the sequence name
         increment counts[s]
      end for
   end for
end for
```

Determine the score for each sequence as its count times its length. Do not consider sequences that contain a sequence marker or a branch bytecode. Denote the sequence with maximum score as 'best' and add it to the multicode list.

```
for each line in the block list
   for each occurrence of 'best'
      replace the occurence with a marker
         denoting the sequence
   end for
end for
```

Using this algorithm identifies sequences of many different lengths for each benchmark. The top 18 sequences for _201_compress are shown below. These sequences were used to achieve the performance gains described in subsequent sections.

```
 1 aload_0 getfield
 2 dup getfield dup_x1 iconst_1 iadd putfield
 3 astore aload getfield iload_2 iaload istore iload iload_1
 4 astore aload dup getfield iconst_1 isub putfield aload getfield
   aload getfield baload invokevirtual
 5 ishl iload iadd istore_1 iload_3 iload ishl iload ixor istore_2
 6 aload_0 dup getfield iconst_1
 7 iload_1 saload bipush ishl bipush iushr
 8 iload_2 iload isub dup istore_2
 9 astore aload getfield iload_2 baload istore aload getfield aload
10 bipush iload_2 isub baload iand sipush iand istore_1 iload_3
   bipush iload_2 isub isub istore_3 bipush iload_2 isub istore_2
   iload_3 bipush
11 iload baload getstatic iload_2 baload iand iload_1 iload_2 ishl
   getstatic iload_2 baload iand ior i2b bastore iinc iload_3 bi-
   push iload_2
12 iload iload_2 iconst_3 ishr iadd istore iload_2 bipush iand is-
   tore_2
13 iload_2 iload iadd istore_2
14 astore aload getfield
15 astore iload_1 i2b istore aload getfield iload_2 iload bastore
   aload_0 iload_2 iconst_1 iadd putfield
16 iadd putfield iload_3
17 baload sipush iand
18 isub isub istore_3 iload_1 bipush iload_2 isub ishr istore_1
   iload_3 bipush
```

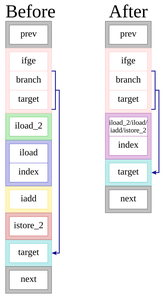
5 Multicode Substitution

An application was developed that parses binary Java class files. It is capable of identify occurrences of the sequences determined using the multicode identification algorithm and replacing those sequences with bytecodes that are unused in the Java Virtual Machine Specification [1]. For example, consider what happens when the sequence show below is replaced.

Once the substitution has been performed, the total number of bytes required to represent the operations within the Java class file is reduced as shown below.

The substitution process is complicated by the fact that the multicode is smaller than the sequence of bytecodes due to the reduced number of opcodes involved.

- Branch displacements must be adjusted so that they continue to target the desired bytecode.
  - Before the multicode substitution is performed, the branch must span 4 opcodes and 1 operand for a total of 5 bytes
  - After the substitution, the branch displacement must be reduced because 3 opcodes have been removed



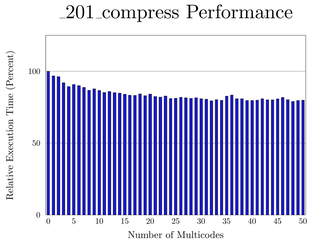
- Exception handler ranges must be updated so that they continue to encompass the same bytecodes.
- The number of pad bytes included within lookupswitch and tableswitch must be modified so that the arguments continue to begin at memory addresses that are word aligned.

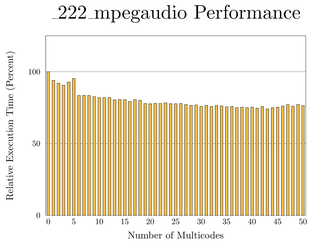
6 Testing Environment

The benchmarks in the SPEC JVM 98 Benchmark Suite were profiled and tested at input size 1. A modified version of the Kaffe VM 1.0.7 was executed on a Pentium III clocked at 650MHz running RedHat Linux 2.4.20-20.9smp.

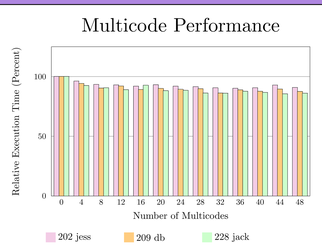
7 Multicode Performance

Testing revealed significant performance gains for some benchmarks. In particular, _201_compress and _222_mpegaudio showed gains of more than 20 percent when 50 multicode substitutions were performed as illustrated in the graphs below.





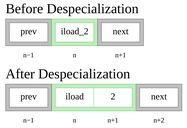
Performance gains of approximately 10 percent were achieved for _202_jess, _209_db and _228_jack.



8 Future Work

The Java Virtual Machine Specification defines the functionality of 201 distinct bytecodes.

- Some bytecodes provide unique functionality
- Some bytecodes are specialized forms of more general bytecodes
  - Specialized bytecodes may automatically use values that must be passed as arguments to the more general form
  - The functionality of some bytecodes is easily duplicated with a short sequence of bytecodes
  - Previous work has shown that performing as many as 67 despecializations has little impact on application performance [3]



The number of bytecodes impacted by a multicode substitution will increase if despecializations are performed before multicode identification and substitution since the number of distinct sequences will be reduced. For example, after despecialization, the following sequences are all equivalent:

aload_0/getfield
aload_1/getfield
aload_2/getfield
aload_3/getfield
aload/getfield

The size of the set of equivalent sequences increases exponentially with the number of specialized bytecodes in the sequence. For example, a sequence that contains 10 specialized load and/or store bytecodes has a set of equivalent sequences with 4^{10} or over 1,000,000 unique members.

It was found that both of the benchmarks that showed large performance gains shared common characteristics:

- These benchmarks had at least twice as many multicodes with length 10 or more than the other benchmarks
- The average multicode length for these benchmarks was longer than the average multicode length for the remaining benchmarks

This indicates that it is desirable to select multicodes that are as long as possible while still occurring frequently.

The current analysis being performed does not permit multicodes to span basic blocks. However, this constraint can be loosened without causing any problems with multicode substitution. A multicode block will be defined to be a sequence of bytecodes with exactly one entry point. Unlike the definition of a basic block, a multicode block is permitted to have many exit points. As a result, bytecodes that perform branching may reside inside a multicode. It is expected that revising the analysis so that multicodes can span basic blocks but not multicode blocks will increase both the average multicode length and the total number of bytecodes impacted by multicode substitution, thus further improving the performance benefit achieved with multicode substitution.

9 Conclusion

The performance gains achieved by multicodes have been improved to more than 20 percent for some benchmarks by

- Increasing the number of multicode substitutions performed
- Increasing the length of each multicode used

It was found that the largest performance gains were achieved were achieved for those benchmarks with the largest average multicode size. As a result, future work will use techniques that help increase multicode size

- Despecialization can be used to increase the number of bytecodes impacted by each substitution
- Permitting multicodes to span basic blocks in some situations will also increase the average multicode length

Acknowledgments

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

References

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

[2] 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.

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

Western