A Technique for Utilizing Optimization Potential during Multicode Identification

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

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

ABSTRACT

Multicodes have been demonstrated to provide performance gains of up to 25 percent. Present multicode identification techniques rely on frequency of occurrence and sequence length alone. This research extends previous work by presenting a multicode identification algorithm based on the frequency of occurrence and the optimization potential of the sequence.

Categories and Subject Descriptors

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

General Terms

Performance, Languages

Keywords

virtual machine, bytecode, optimization, Java, interpreter

1. INTRODUCTION

Multicode analysis and optimization determines frequently occurring sequences of bytecodes executed by a Java application and replaces such sequences with a new bytecode that provides equivalent functionality [2]. This replacement results in performance improvements in Java interpreters for two reasons.

- The total number of bytecodes executed is reduced resulting in fewer costly transfers of control from one bytecode to the next.
- When the instructions used to implement a sequence of bytecodes are concatenated new optimization opportunities become available.

Profiling is used to determine which sequences of bytecodes should be replaced with multicodes. Once the application is profiled, each sequence encountered is assigned a score. Previously, the score of the sequence was computed using only the number of occurrences of the sequence and the length of the sequence. While this technique has shown performance gains of as much as 25 percent, it fails to consider the optimization potential of candidate sequences. The remainder of this paper discusses how optimization potential can be incorporated into the multicode identification process so that further performance gains can be achieved.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

2. OPTIMIZATION POTENTIAL

Figure 1 shows the score of the top 50 sequences for the SPEC JVM 98 `_228_jack` benchmark [1] computed using the frequency and length of the sequence alone. While there is a great deal of difference in the scores of the first handful of sequences the difference in the scores of the sequences diminishes as one progresses to the right. Consequently, when optimization potential is considered in addition to other factors, it is possible that a sequence may move many positions based on its optimization potential. This shift will impact the list of sequences implemented as multicodes in two ways.

- One specific sequence will change position, potentially allowing a sequence that was previously being discarded to be implemented.
- Because steps are taken to ensure that no bytecode is counted as part of two different multicodes, moving one sequence will impact the scores of all sequences considered subsequently that contain any bytecodes from the sequence that changed position.

In order to consider the optimization potential of the sequence, it is both necessary to improve the score computation to consider this information and devise a manner for accurately determining the optimization potential of a sequence.

The full cost of executing a bytecode is the cost of the operations within the body of the bytecode plus the cost of the transfer of control to reach the next bytecode where cost is expressed in any logical unit. Therefore, the cost of executing a sequence of k bytecodes is the sum of the costs of the body for each bytecode plus $k \times T$, where T is the cost of the transfer of control.

The cost of executing the body of the multicode is not simply the sum of the cost of executing the bodies of its

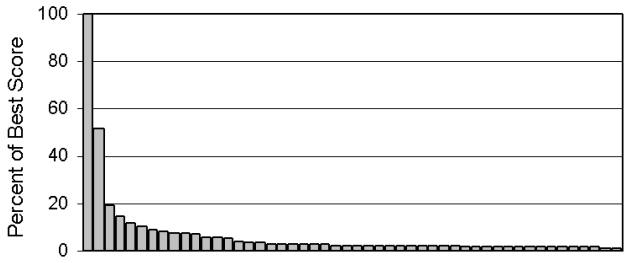

Figure 1: Multicode Scores Expressed as Percent of Best Score

```
public static void do_timing() {
  Initialize local variables
  Begin Timing
  For 1,000,000 Iterations
    Push necessary operands onto stack
    Execute bytecodes or multicode
      begin evaluated
    Pop result from stack
  End For
  End Timing and Display Elapsed Time
}
```

Figure 2: General form of the Main Method used to Determine Optimization Potential

constituent bytecodes because additional optimizations can potentially be performed when the bodies of the bytecodes are adjacent to each other. Equation 1 shows the relationship between the cost of executing a sequence of bytecodes and their corresponding multicode including the necessary transfers of control.

$$Cost(bc_1) + T + ... + Cost(bc_k) + T \geq Cost(mc_1) + T \quad (1)$$

The improvement in performance that can be expected from a single multicode substitution can be expressed as the difference in the two sides of the equation.

$$Gain = (C(bc_1) + ... + C(bc_k) - C(mc_1) + T(k-1)) \quad (2)$$

Having developed a formula that expresses the performance gain that can be expected from a single multicode substitution based on the length of the sequence and its optimization potential, it is necessary to developed techniques to acquire these values. The length of the sequence is immediately available by counting the bytecodes. Unfortunately measuring the cost of the body of a bytecode or multicode or the cost of a transfer of control is not as simple.

In order to overcome this problem, we propose to implement and measure the runtime performance of candidate sequences. Measuring the runtime performance eliminates the need to attempt to assign a value to the cost of a bytecode body or a transfer of control from one bytecode to the next because the total time, including both the body and any transfers of control, can be measured as a total quantity. This reduces our computation of *Gain* to Equation 3, where $\rightarrow$ represents the transfer of control between bytecodes.

$$Gain = Cost(bc_1 \rightarrow ... \rightarrow bc_k \rightarrow) - Cost(mc_1 \rightarrow) \quad (3)$$

Determining the cost of the sequence of bytecodes through timing is accomplished by constructing a new class file at the bytecode level. It contains a static method method that performs the timing. A helper class is also generated that contains additional methods and fields that will be utilized when `invoke*` and field access bytecodes are present in the sequence being evaluated. The general form of the timing method is shown in Figure 2. By timing a sequence of bytecodes and its corresponding multicode 1,000,000 times, an estimate of the expected performance gain for each dynamic occurrence of the multicode can be computed.

Creating the `do_timing` method is not trivial. Operands may need to be present on the stack before a sequence of bytecodes can execute. An analysis is performed to determine the types of such values and bytecodes are generated to put appropriate values into place. In the general case, there may not be sufficient information to fully determine the types of all values. This can arise with a variety of bytecodes such as `invokevirtual` which consumes a variable number of items from the stack and the `getfield` and `getstatic` bytecodes that place a value of arbitrary type onto the stack. We handle such bytecodes by inferring the types of as many values on the stack as possible and then make assumptions to fill in the unknown values. Contradictions can be avoided because the only times that it is not possible to determine what type of value should be placed on the stack is when the value placed on the stack is not used subsequently within the sequence or the subsequent use is by a bytecode that is not concerned with the type of the value consumed from the stack. The `invoke*` bytecodes are handled by invoking a method with sufficient arguments to consume all of the values that are currently on the stack. Popping results from the stack is accomplished using `pop` and `pop2` bytecodes.

The operations associated with pushing of operands and popping of results from the stack do not cause any difficulties with timings. Identical operations are performed for the timing of both the bytecode sequence and the multicode. As a result, when the subtraction is performed while computing the expected performance gain, the cost associated with these setup and cleanup operations is eliminated.

Experimental results have shown that gain values can vary considerably even when the number of transfers removed is the same. For example the commonly occurring sequence `aload_0/getfield` shows an improvement value of 86ms per million executions while `iconst_1/iadd` shows an improvement value of only 40ms per million executions. Even wider variations are observed when sequences of different lengths are compared.

Having described a technique that can determine the optimization potential of a sequence, the score of a sequence can be computed as $N \times Gain$. Using this formula, each multicode substitution will be made so as to minimize application run time rather than minimizing the number of transfers of control. It is expected that determining what multicode substitutions to make using this technique will offer superior results to those achieved previously.

3. CONCLUSION

A framework is presented that utilizes the optimization potential of a bytecode sequence during multicode identification. The technique involves timing candidate sequences to determine the optimization potential so that factors such as pipelining and cache utilization are taken into consideration. This is expected to show performance gains over previous multicode identification techniques which attempted to minimize the number of transfers of control between bytecodes without considering other factors.

4. REFERENCES

[1] Standard Performance Evaluation Corporation. *SPEC JVM98 Benchmarks*, 2004. http://www.spec.org/jvm98.

[2] B. Stephenson and W. Holst. Advancements in multicode optimization. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 186–187, New York, NY, USA, 2004. ACM Press.