Approaches to Reflective Method Invocation

Ian Rogers, Jisheng Zhao, and Ian Watson

The University of Manchester,
Oxford Road, Manchester,
M13 9PL, United Kingdom
{ian.rogers, jisheng.zhao, ian.watson}@manchester.ac.uk

Abstract. Reflective method invocation is a long known performance bottle neck in Java. Different approaches to optimizing reflective method invocation are adopted by JVMs. In this paper we present an overview of the different approaches and a performance analysis using synthetic and the DaCapo benchmarks. We implement all of the approaches in the Jikes RVM.

Keywords: Java Virtual Machine, Reflection, Performance, Method Invocation

1 Introduction

Reflective method calls are a means for programmers to interact with objects and classes the compiler would have been unaware of at static compile time. Reflective method calls are also used internally within the JVM to implement other reflective mechanisms such as creating objects from a class object, as part of the invocation of Java methods from native code and determining the main method to execute given command line arguments. Programmers have used reflective method calls for Java beans and in languages built on top of the JVM like Jython [5]. In Jython parts of the language are modelled with dynamically created Java classes. The Java classes are unknown when Jython starts and accessed in part using reflective object constructors and methods.

The implementation choices for reflection depend upon what use is expected of it. There is a trade-off between fast initial invocation and fast general case performance, as well as choices into what memory needs to be allocated. We present these alternatives in Section 2, we also present some optimizations that hope to achieve middle routes for performance. In Section 3 we analyse the performance of the different approaches on a synthetic benchmark and the DaCapo benchmarks. We consider related work in Section 4. Section 5 concludes the paper.

2 Overview

Figure 1 gives an example of invoking the method *bar* using reflection and printing hello.

```
import java.lang.reflect.Method;
class Example {
  public static void main(String[] args) {
    try {
      Method m = Example.class.getMethod(``bar'');
      m.invoke(null);
    } catch (Exception e) {}
  }
  public static void bar() {
    System.out.println(``hello!'');
  }
}
```

Fig. 1. Example of Reflective Method Invocation

When performing the method invocation Java checks that reflection cannot bypass normal scoping guarantees, it also provides a means for setting a method to be accessible. Public methods are always accessible but otherwise a method's accessibility can vary.

Once past the accessibility checks the arguments to a reflective method call need checking. In some cases the arguments are converted to the type expected by the method call. The arguments are passed in an array.

The final part of the reflective method call is to perform the actual method call. One approach to perform the method call is to have a special helper routine that is responsible for getting the address of the method to invoke (in some situations JIT compiling it), taking the arguments to the call and placing them in the correct position for the calling conventions and finally branching to the method. We call this the *out-of-line machine code* approach.

An alternate approach to performing the method invocation is to dynamically generate Java bytecode that will invoke the method. Generating the bytecode has a run time cost and overhead, not just for the bytecode but also for a wrapper class and object that allow the bytecode of the generated method to be invoked. The bytecode may be created when the call to get the method is made or it may be made on the first invocation. We call these approaches *bytecode generation* and either *eager*, when the bytecode is created when the method is constructed, or *lazy* when the method is first invoked. Being lazy allows memory and time to be saved when method reflection isn't used. For the lazy scheme the generated bytecodes aren't referenced from the method itself but help in a map.

The *out-of-line machine code* takes an argument that is the method to be invoked. If this method is a constant within the JIT compilation then we may simplify the reflective method call to a direct method call. The arguments to the method must be taken out of their wrapper array. We call this approach *out-of-line machine code with simplification*.

An advantage to the *out-of-line machine code with simplification* and *bytecode generation* approaches is that the array used to hold the arguments for the reflective method call may be redundant. This can mean the allocation of this array can be eliminated avoiding run time garbage collection overhead. When this cannot be eliminated we call it a *boxing overhead*.

We summarise the properties of the different approaches in Table 1.

	Out-of-line machine code	Out-of-line machine code with simplification	Bytecode generation eager	Bytecode generation lazy
Creating the reflection wrapper	Fast, method is sought and wrapped up.		Slow bytecode must be generated.	Fast, method is sought and wrapped up.
Initial invocation	Fast as no bytecode or machine code is generated.		Fast as bytecodes allow direct dispatch to method.	Slow bytecode must be generated prior to invocation.
Boxing overhead	Always	Maybe removed with optimizing compilation.		
Extra overheads	None.		Wrapper class and object.	
Second invocation performance	Same speed as initial invocation.		Fast as bytecodes allow direct dispatch to method.	
Frequent invocation performance	Same speed as initial invocation.	Improved if method is a constant.	Fast as bytecodes allow direct dispatch to method.	

Table 1. Summary of Reflective Method Invocation Schemes

The Jikes RVM [1] currently uses an *out-of-line machine code* approach to reflective method invocation, whilst it is well known that Sun perform *bytecode generation*.

3 Performance Analysis

3.1 Synthetic Benchmark

Figure 2 shows our synthetic benchmark, it performs a number of long additions and returns the number of method invocations within 100 milliseconds.

We run the method *doTest* 50 times in each round and average the performance. We give results from the 5th round of execution to avoid measuring compilation overhead. Each test is run 30 times and the mean and 95% confidence intervals calculated for each round. All of these programs are run on a Intel P4 3.0 GHz processor, 1GB memory and OpenSUSE 10.3 operating system. Figure 3 shows the result for the Jikes RVM with each of the four schemes.

The results show that both *eager bytecode generation* and *out-of-line machine code with simplification* can achieve roughly the same peak performance. The extra overhead of the *lazy* invocation is clear, but it still performs better than *out-of-line machine code*. Depending on when the optimizing compilation is performed the *lazy* performance was occasionally as good as the *eager* and *out-of-line machine code with simplification*, but on average for this benchmark it is worse.

3.2 DaCapo Benchmarks

We ran a selection of the DaCapo benchmarks[2] that perform a large amount of reflection method invocation. The results are shown in Figure 4.

```
class test {
  // ...
  static final Method m;
  static {
    try {
      m = test.class.getMethod(''add'', long.class, int.class);
    } catch (Throwable t) {
      throw new Error(t);
    }
  }
  public static long doTest() throws ... {
    long startTime = System.currentTimeMillis();
    long value = 0;
    do {
      value = m.invoke(null, value, 1);
    } while (startTime + 100 > System.currentTimeMillis());
    return value;
  }
  public static long add(long x, int i) {
    if (x+i < x) throw new Error("Overflow!");
    return x + i;
  }
}
```

Fig. 2. Synthetic Method Invocation Benchmark

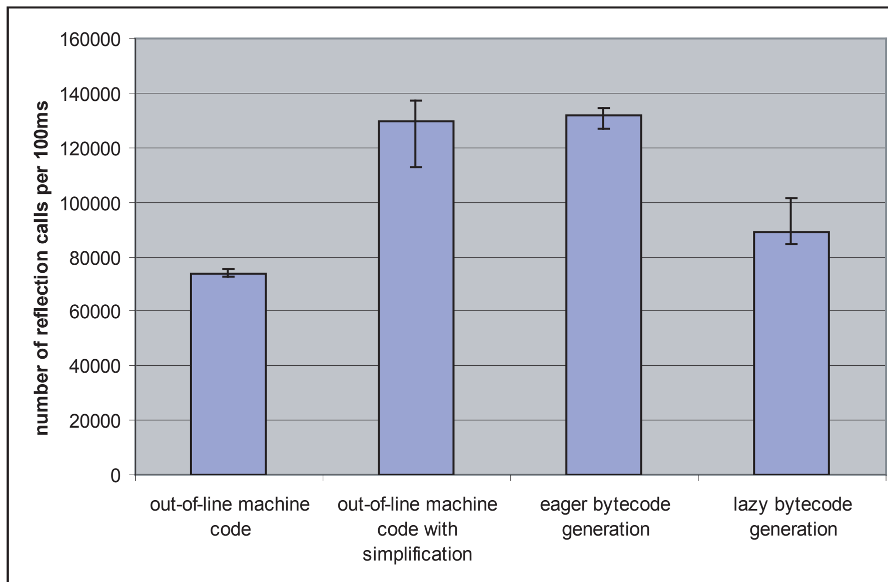

Fig. 3. Performance of Reflective Method Invocation Schemes on a Synthetic Benchmark

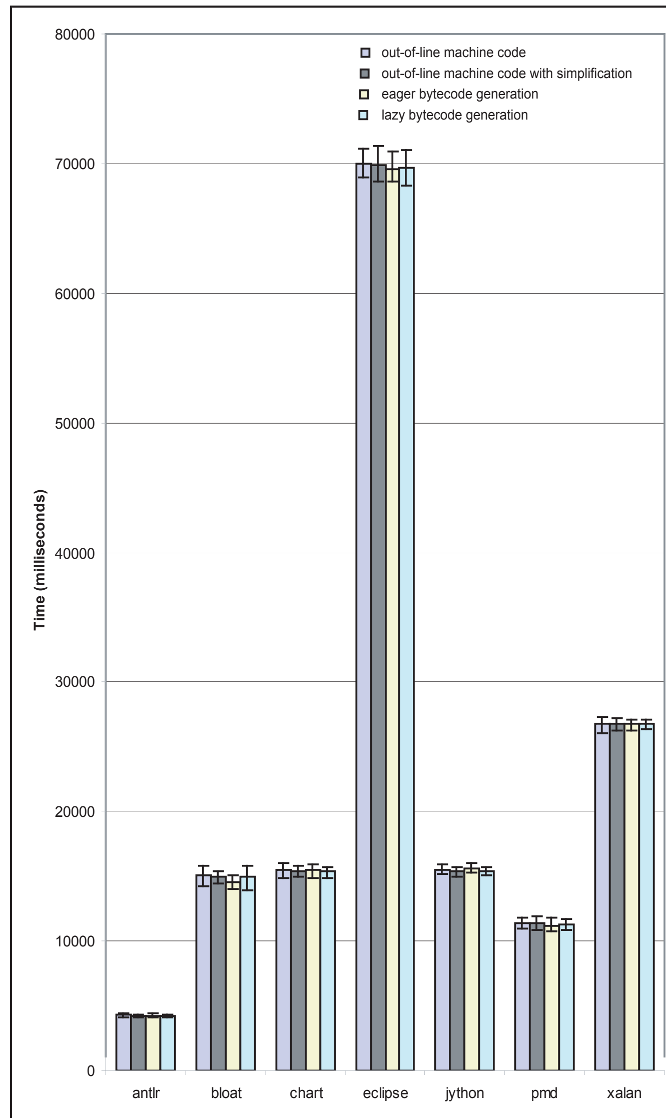

Fig. 4. Performance of Reflective Method Invocation Schemes on DaCapo Benchmarks

The results show that *bytecode generation* has better performance than *out-of-line machine code*. The biggest saving in total execution time comes from *eager bytecode generation* as the execution time for eclipse is long.

Bytecode generation has the advantage of optimizing not just in the optimizing compiler but for interpreted or JIT compiled code. Another problem for the *out-of-line machine code* with simplification approach is it can only simplify when the method being invoked is a constant. Fields that are static final are constant within the optimizing compiler, as in our synthetic benchmark example. We also modify the Jikes RVM class loader so that method lookup operations are pure, this can mean lookups with literal values, such as strings, can be simplified. When the methods are determined using non-literal values or static final fields then simplification cannot ensure dispatch directly to a method, whereas bytecode generation can.

Figure 5 shows the speedup from each method reflection approach on different DaCapo benchmarks. The speedup for each benchmark is independent of its execution time and we compute the average speedup from this. Averaged across all the benchmarks, and compared to *out-of-line machine code*, the addition of *simplification* achieved a further 0.67% speedup, *eager bytecode generation* a 0.88% speedup and *lazy bytecode generation* a 0.89% speedup. It is clear *bytecode generation* achieves the greatest speedup and across a range of the DaCapo benchmarks the *lazy* approach can offer the best average performance. It is interesting to see that much of the average case performance for *eager bytecode generation* comes from DaCapo bloat.

Bytecode generation has an overhead to create the bytecode and in storage. Our results show this is less significant than its associated speedup. The results show that lazy generation, although having a performance cost shown in Section 3.1, for use in the DaCapo benchmarks is the best all-round approach.

4 Related Work

In order to remove the costs associated with reflection our work uses partial evaluation. We simplify or fold expressions to reduce the overall complexity of the program. This is combined with constant and copy propagation. When possible we also avoid walking the run time stack to determine information for reflection, substituting values available to the optimizing compiler. Braux and Noyé [3] consider partial evaluation to eliminate reflective method overheads. Their approach is to use an off-line partial evaluator. We differ in doing analysis online, which is less of a problem for Java thanks to strong typing that reduces the amount of analysis that needs to be performed. This work differs in being focused on reflected method invocation performance.

Livshits, Whaley and Lam [4] provide an analysis of reflection in Java, in particular for calculating precise call-graph and pointer analysis. Their work requires extension for online use where classes within the system can alter dynamically. For example, in their work they don't consider a program like Jython. However, their analysis is more powerful than ours and could potentially yield more op-

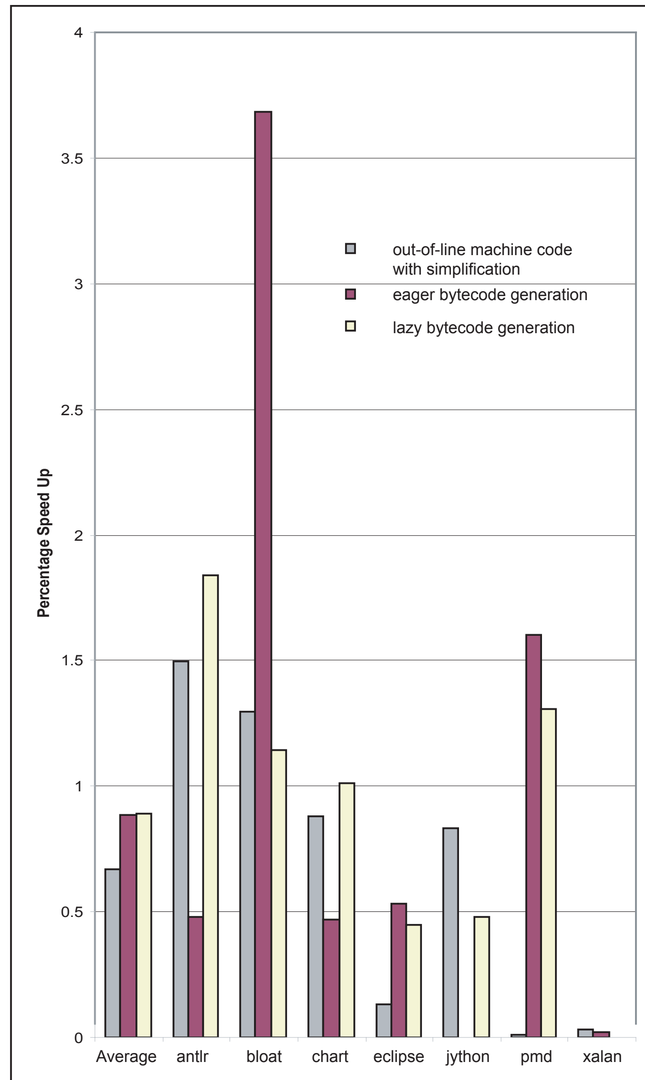

Fig. 5. Speedup from Reflective Method Invocation Schemes Compared to Out-Of-Line Machine Code for DaCapo Benchmarks

timization opportunities. As getting good performance early has proven to be important, we believe such analysis could have only limited run time benefit.

5 Conclusions

We have demonstrated a range of approaches to implement efficient method reflection that demonstrate a performance advantage over *out-of-line machine code*. Of the approaches *bytecode generation* performed best. Across a number of DaCapo benchmarks *eagerly* generating bytecode gives the best overall saving in execution time, however, the average speed up is marginally better if the bytecode is generated in a *lazy* manner. Based on its average case behaviour, *lazy bytecode generation* appears the most beneficial scheme.

References

1. Jikes^{TM}Research Virtual Machine(RVM). http://jikesrvm.org/, 2008.
2. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, October 2006. ACM Press.
3. Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in java. In *PEPM '00: Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 2–11, New York, NY, USA, 2000. ACM.
4. Benjamin Livshits, John Whaley, and Monica Lam. Reflection analysis for Java. Technical report, Stanford University, 2005.
5. S. Pedroni and N. Rappin. *Jython Essentials*. O'Reilly Media, Inc., 2002.