Pure Method Analysis within Jikes RVM

Jisheng Zhao, Ian Rogers, Chris Kirkham, and Ian Watson

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

Abstract. Not all compiler analysis can determine a method call will have no side effects, for example if the method called is performing memoization of common results. Having a pure attribute allows such methods to be flagged to the compiler and their values evaluated at compile time. This allows a greater amount of partial evaluation with little compile time overhead. The lack of overhead to the compiler motivates us to look for other instances of pure methods, where rather than spend time trying to inline and reduce code, we merely evaluate a call at run time. This paper presents a study of different methods of inferring pure methods in the dynamic compilation environment of a Java Virtual Machine (JVM). We look at programmer specified annotations, determining pure method information from naive bytecode analysis and more sophisticated analysis in an optimizing compiler. The optimizing compiler analysis is able to speed up the run time environment by an average of 1.29% on a range of DaCapo and SpecJVM benchmarks.

Keywords: Java Virtual Machine, Dynamic Compilation, Program Analysis

1 Introduction

Method inlining is a widely used approach for eliminating run time overhead of method calls. JIT compilers employ method inlining to improve the compiled program's performance but inlining has a compilation cost when simplifying the code is not trivial.

Pure annotations and compiler pragmas exist for two reasons. A pure method may not have side effects, and a compiler may verify this [4]. A pure method may have optimizations performed upon it, such as compile time evaluation [2]. In functional languages all functions are implicitly pure. In our system pure is a guarantee that a method cannot have: memory load/store operations, synchronization operations in more than one thread, throw exceptions, allocate memory, perform method calls to non-pure methods, and perform run time services that have side effects.

By looking for pure methods the optimizing compiler can reduce method calls and perform other optimizations that wouldn't be safe in the presence of a

```
x = y + 1;
foo(...);
z = y + 1;
   (a)

x = bar(const_0, const_1);

         (c)
```

```
x = y + 1;
foo(...);
z = x;
      (b)

x = val; // val is calculated by
// compiler evaluation of
// bar(const_1, const_2)
      (d)
```

Fig. 1. Example of Pure Method Simplification

method call. For example, Figure 1 (a) lists a small segment of code: *foo* is a pure method, so *foo* it will not affect the values of the fields x and y, thus the compiler can perform common sub-expression elimination on the third statement making it assign x to z directly (see Figure 1). Figure 1 (c) gives another simple example, *bar* is a pure method and has two constant input parameters, so the compiler can evaluate method *bar(const_0, const_1)* at compile time, get the return value *val* and assign it to x directly.

A pure method can also be utilised to guarantee that arguments won't escape a thread. If an object doesn't escape a thread then synchronization operations on it may be removed. Note that such optimizations are only possible when the pure method is only precisely reachable, that is if the method requires dynamic dispatch we don't consider it for optimization. Dynamic dispatch may be eliminated by guarded method inlining.

Although the program languages or compilers (e.g. Java, gcc) provide pure method annotations, the application programmers may not annotate all of the pure methods in the application code. There are also potentially pure methods in libraries (e.g. Java's classpath) that were not annotated by the developers. The compile time simplicity of evaluating a pure method and the extra optimization opportunities it brings, make it attractive to automatically enable the compiler to determine pure methods. This analysis can be viewed as a form of inter-procedural analysis, but without any dependency tracking. In this paper, we investigate how and what are the advantages of a dynamic compiler performing automatic pure method analysis and annotation. The annotation provides more opportunities for partial evaluation whilst reducing compile time costs.

2 Automatic Pure Method Analysis and Annotation

Automatic analysis means that the compiler needs to recoup the analysis workload. This is a common trade-off for compiler optimizations in a run time compiler. The automatic pure method analysis and annotation needs to balance the optimization benefit and compilation overhead. Fortunately, the constraints for a pure method are simple (as mentioned in Section 1). In this work we consider two run time approaches to performing pure method analysis:

– *Simple Pure Method Analysis*: the compiler performs a simple constant time analysis to check if a method is pure method. Using a summary of bytecodes, held for each method within the run time, a simple mask can determine that a bytecode containing method can be considered pure (see Figure 2 (a)). When the class is loaded the loader scans the bytecodes and records information on what operations the method can perform. Combining this with a check that the method obeys the pure method constraints (described in Section 1), a simple pure method analysis is possible.
– *Optimizing Compiler Analysis*: this analysis is performed after the optimizing compiler has performed a number of optimizations. In this study, we employ constant propagation, copy propagation, type propagation, and local sub-expression elimination prior to the analysis (see Figure 2 (b)). By creating a pure method the analysis will enable further optimizations when the method is considered during the compilation of other methods.

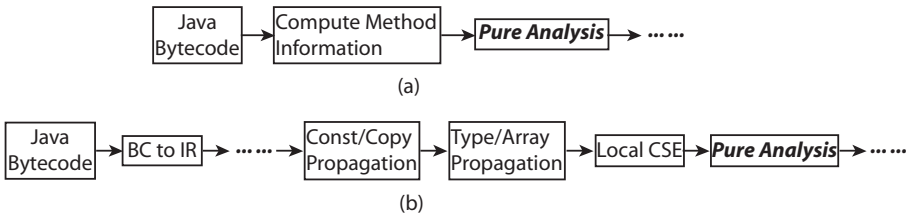

Fig. 2. The Location of Purity Analysis in Optimizing Compiler.

In our analysis the following compilers phases use pure method information: simplification (evaluating a method call with all constant arguments), common sub-expression elimination, escape analysis and dead code elimination. We consider two further configurations when evaluating pure methods. We look at just the pure annotations provided by the programmer, and whether pure annotations are optimized by the compiler or not.

3 Evaluation and Discussion

We use the Jikes Research Virtual Machine (RVM) [1], an open source Java Virtual Machine (JVM) written in Java. Jikes RVM performs a quick compilation on a method upon its first execution; hot methods are recompiled by an optimizing compiler which has three optimization levels: 0, 1 and 2. To evaluate the methodologies proposed in the previous section, we chose DaCapo benchmarks [5] and SpecJVM98 1.04 [8]. All of these programs are run on a Intel P4 3.0 GHz processor, 1GB memory and OpenSUSE 10.3 operating system. Each benchmark was run 60 times and the mean and 95% confidence intervals calculated.

Table 1 lists the analysis result gained from the two approaches (simple/complex pure method analysis) by applying dynamic compilation. As the analysis is only utilised in the JikesRVM's optimizing compiler it will only be applied to frequently executed methods. This limits the number of annotated pure methods that can be determined at run time. In all of these benchmark programs, there are no pure methods annotated by the application programmer.

Analysis Approach	antlr	bloat	pmd	fop	hsqldb	jython	xalan	mpegaudio	javac	compress	raytrace
Simple Pure Method Analysis	1	0	2	0	0	5	0	2	0	0	0
Complex Pure Method Analysis	1 or 2	1	2 or 3	0	1	6	0	2 or 3	10	0 or 1	1

Table 1. Number of Pure Methods Determined Dynamically.

For some benchmark programs, the number of pure methods determined at run time is not a static value (e.g. antlr, pmd) when they are evaluated in multiple times. Because JikesRVM's adaptive recompilation system is not precise for identifying all of the hot methods in each run (i.e. it may lost some hot method in some run).

On top of the methods determined dynamically in the different benchmarks the Jikes RVM boot image is compiled and analysed before run time. Originally, there are 282 pure methods in JikesRVM boot image. Using simple analysis 1,469 additional methods were found to be pure in the boot image.

In Figure 3 we have four configurations:

- *No Pure*: there are no pure method related optimizations in compiler.
- *Programmer Provided*: the pure methods are annotated by the application's programmer or have been added to the class library during the Jikes RVM build using bytecode engineering. There is no compiler analysis for pure annotations. 236 methods are annotated through bytecode engineering of the class library and 155 methods are directly annotated within Jikes RVM and the class library support code.
- *Simple Pure Method Analysis*: use simple flags to determine pure methods as introduced in Section 2. This analysis may be used in building the boot image.
- *Optimizing Compiler Analysis*: use the optimizing compiler to determine pure methods as introduced in Section 2. This technique may also be applied to the boot image of the Jikes RVM, but isn't in this work.

We use the *No Pure* scheme to compare the performance with other three schemes. Figure 4 shows the speedup of the schemes: *Programmer Provided*, *Simple Pure Method Analysis*, and *Optimizing Compiler Analysis*.

On average with optimizing compiler analysis a speed up of 1.29% was achieved over not using pure annotations. The simple analysis for hsqldb gave a small -0.13% slow down, but this was the only noticeable slow down due to the

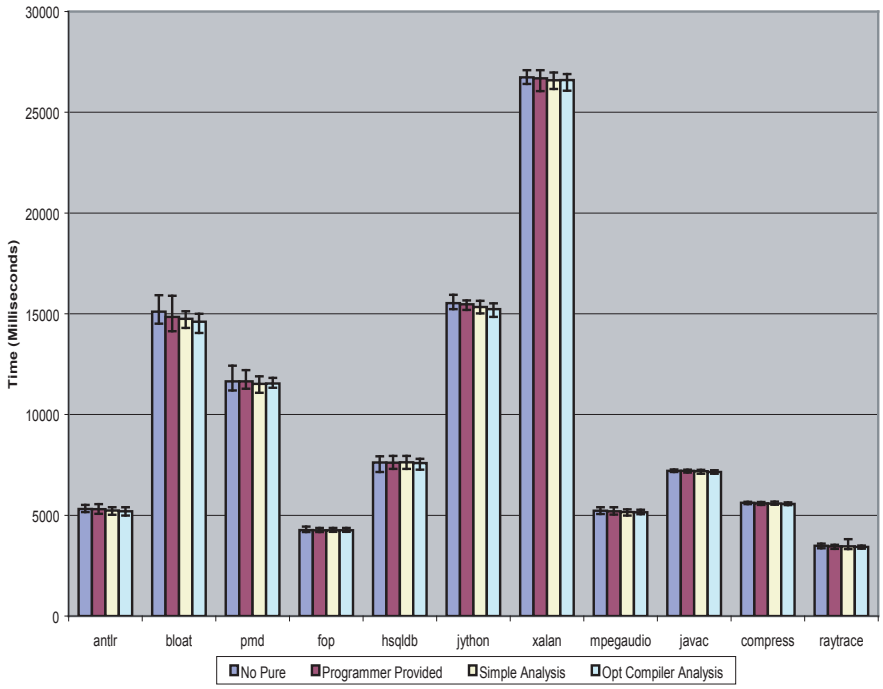

Fig. 3. Mean and 95% Confidence Interval for Overall Times

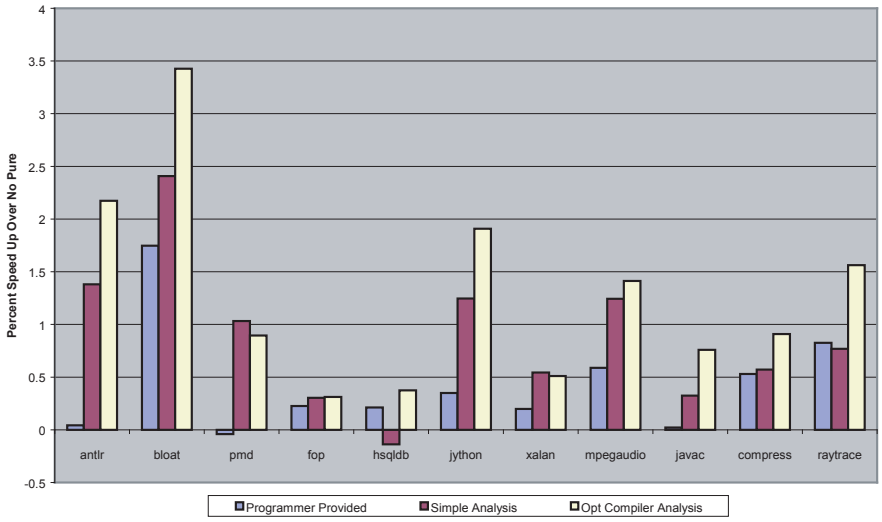

Fig. 4. Speedup Over No Pure Annotations.

extra analysis and optimization. The optimizing compiler analysis more than recouped this slowdown.

For *Programmer Provided* configuration, the speedup is mainly gained from the programmer providing pure methods that benefit code within the boot image. The difference of speedup between *Simple Pure Method Analysis* and *Optimizing Compiler Analysis* is caused by the dynamically determined pure methods in the benchmark programs.

4 Related Work

Xu et al. [10] explored both static and dynamic purity analysis for Java bytecode. The static analysis is implemented in the Soot compiler [9], which analyses and then generates Java class files with additional purity information. The dynamic analysis is implemented in an extended component of SableVM [6]: a Java bytecode interpreter. The purity analysis is evaluated by different levels of constraints (strong purity, moderate purity, weak purity and once-impure purity). They measure the performance improvement of a memoization optimization, in SableVM, and show sizeable run time overhead to their system.

Salicanu and Rinard [7] introduced a purity analysis approach in a static Java bytecode compiler: Flex [3]. Their approach is built on top of a combined pointer and escape analysis for Java programs and is capable of determining that methods are pure even when the methods do heap mutation, provided that the mutation affects only objects created after the beginning of the method.

In contrast with the above work, the method presented in this paper implements purity analysis in a JIT optimizing compiler that needs to achieve a good balance between analysis overhead and benefit. Our results have shown that employing light weight purity analysis in JIT compilation has benefitted the run time performance.

5 Conclusion and Future Work

We have shown that determining pure methods at run time is advantageous to the dynamic compilation environment, it can lower the overhead of compilation and lead to an overall speedup. On average we achieved a 1.29% speed up on a range DaCapo and SpecJVM benchmarks. We hope to further utilise approaches such as annotations to perform efficient optimizations. We are particularly interested in the area of value specialisation. There are other compiler optimizations that could use the pure method information, for example memoization could be added onto methods to cache common results.

References

1. Jikes^{TM}Research Virtual Machine(RVM). http://jikesrvm.sourceforge.net, 2005.
2. Gcc online documents. http://gcc.gnu.org/onlinedocs/, 2007.
3. C. Scott Ananian. Mit flex compiler infrastructure for Java. http://www.flex-compiler.lcs.mit.edu, 1998-2004.
4. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS'04: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer, January 2005.
5. 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.
6. Etienne Gagnon. *A portable research framework for the execution of Java bytecode*. PhD thesis, Montreal, Que., Canada, Canada, 2003. Adviser-Laurie J. Hendren.
7. A. Salcianu and M. Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAILTR-949, MIT, May 2004.
8. SPEC JVM98 benchmarks. http://www.spec.org/osg/jvm98/, 1998.
9. R. Vall, e Phong, C. Etienne, G. Laurie, H. Patrick, and L. Vijay. Soot - a java bytecode optimization framework, 1999.
10. Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for java programs. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 75–82, New York, NY, USA, 2007. ACM.