Optimizing Java Bytecode using the Soot Framework: Is it Feasible?

Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren,
Patrick Lam, Patrice Pominville and Vijay Sundaresan
{rvalleerai,gagnon,hendren,plam,patrice}@sable.mcgill.ca
vijaysun@ca.ibm.com

Sable Research Group, School of Computer Science, McGill University

Abstract. This paper presents Soot, a framework for optimizing Java™ bytecode. The framework is implemented in Java and supports three intermediate representations for representing Java bytecode: Baf, a streamlined representation of Java's stack-based bytecode; Jimple, a typed three-address intermediate representation suitable for optimization; and Grimp, an aggregated version of Jimple.
Our approach to class file optimization is to first convert the stack-based bytecode into Jimple, a three-address form more amenable to traditional program optimization, and then convert the optimized Jimple back to bytecode.
In order to demonstrate that our approach is feasible, we present experimental results showing the effects of processing class files through our framework. In particular, we study the techniques necessary to effectively translate Jimple back to bytecode, without losing performance.
Finally, we demonstrate that class file optimization can be quite effective by showing the results of some basic optimizations using our framework. Our experiments were done on ten benchmarks, including seven SPECjvm98 benchmarks, and were executed on five different Java virtual machine implementations.

1 Introduction

Java provides many attractive features such as platform independence, execution safety, garbage collection and object orientation. These features facilitate application development but are expensive to support; applications written in Java are often much slower than their counterparts written in C or C++. To use these features without having to pay a great performance penalty, sophisticated optimizations and runtime systems are required. For example, Just-In-Time compilers[1], adaptive compilers such as Hotspot™ and Way-Ahead-Of-Time Java compilers[18, 17] are three approaches used to improve performance.

Our approach is to statically optimize Java bytecode. There are several reasons for optimizing at the bytecode level. Firstly, the optimized bytecode can then be executed using any standard Java Virtual Machine (JVM) implementation (interpreter, JIT, adaptive), or it could be used as the input to a bytecode→C or bytecode→native-code compiler. Thus, the overall performance improvement is due to both our static bytecode optimization, and the optimizations and sophisticated runtime systems used in the virtual machines executing the optimized bytecode. Secondly, many different compilers for a variety of languages (Ada, Scheme, Fortran, Eiffel, etc.) now produce Java bytecode as their target code. Thus, our optimization techniques can be applied as a backend to all of these compilers.

The goal of our work is to develop tools that simplify the task of optimizing Java bytecode, and to demonstrate that significant optimization can be achieved using these tools. Thus, we have developed the Soot[20] framework which provides a set of intermediate representations and a set of Java APIs for optimizing Java bytecode directly. Since our framework is written in Java, and provides a set of clean APIs, it should be portable and easy to build upon.

Early in our work we found that optimizing stack-based bytecode directly was, in general, too difficult. Thus, our framework consists of three intermediate representations, two of which are stackless representations. *Baf* is a streamlined representation of the stack-based bytecode, whereas *Jimple* and *Grimp* are more standard, stackless intermediate representations. Jimple is a three-address representation, where each instruction is simple, and each variable has a type. It is ideal for implementing standard compiler analyses and transformations. Grimp is similar to Jimple, but has aggregated expressions. Grimp is useful for decompilation, and as a means to generate efficient bytecode from Jimple.

In order to optimize bytecode we first convert bytecode to Jimple (the three-address representation), analyze and optimize the Jimple code, and then convert the optimized Jimple back to bytecode. In this paper we focus on the techniques used to translate from Jimple to efficient bytecode, and we give two alternative approaches to this translation.

Our framework is designed so that many different analyses could be implemented, above and beyond those carried out by `javac` or a JIT. A typical transformation might be removing redundant field accesses, or inlining. For these sorts of optimizations, improvements at the Jimple level correspond directly to improvements in final bytecode produced.

We have performed substantial experimental studies to validate our approach. Our first results show that we can indeed go through the cycle, bytecode to Jimple and back to bytecode, without losing any performance. This means that any optimizations made in Jimple will likely also result in optimizations in the final bytecode. Our second set of results show the effect of optimizing class files using method inlining and a set of simple intraprocedural optimizations. These results show that we can achieve performance gains over a variety of Java Virtual Machines.

In summary, our contributions in this paper are: (1) three intermediate representations which provide a general-purpose framework for bytecode optimizations, and (2) a comprehensive set of results obtained by applying our framework to a set of real Java applications.

The rest of the paper is organized as follows. Section 2 gives an overview of the framework and the intermediate representations. Section 3 describes two alternative approaches to translating Jimple back to bytecode. Section 4 presents and discusses our experimental results. Section 5 discusses related work and Section 6 covers the conclusions and future work.

2 Framework Overview

The Soot framework has been designed to simplify the process of developing new optimizations for Java bytecode. Figure 1 shows the overall structure of the framework. As indicated at the top of the figure, many different compilers can be used to generate the class files. The large box labeled SOOT demonstrates that the framework takes the original class files as input, and produces optimized class files as output. Finally, as shown at the bottom of the figure, these optimized class files can then be used as input to Java interpreters, Just-In-Time (JIT) compilers, adaptive execution engines like Hotspot^{TM}, and Ahead-of-Time compilers such as the High Performance Compiler for Java (HPCJ)^{TM} or TowerJ^{TM}.

The internal structure of Soot is indicated inside the large box in Figure 1. Shaded components correspond to modules that we have designed/implemented and each component is discussed in the following subsections.

2.1 Jimplify

The first phase is to *jimplify* the input class files, that is, to convert class files to the Jimple three-address representation. We convert to a three-address representation because optimizing stack code directly is awkward for several reasons. First, the stack implicitly participates in every computation; there are effectively two types of variables, the implicit stack variables and explicit local variables. Second, the expressions are not explicit, and must be located on the stack[25]. For example, a simple instruction such as `add` can have its operands separated by an arbitrary number of stack instructions, and even by basic block boundaries. Another difficulty is the untyped nature of the stack and of the local variables in the bytecode, as this does not allow analyses to make use of the declared type of variables. A fourth problem is the `jsr` bytecode. The `jsr` bytecode is difficult to handle because it is essentially a interprocedural feature which is inserted into a traditionally intraprocedural context.

We produce Jimple from bytecode by first processing the bytecode from the input class files and representing it in an internal format.[1] During this translation `jsr` instructions are eliminated by inline expansion of code. After producing the internal form of bytecode, the translation to Jimple code proceeds as follows:

1. *Produce naive 3-address code*: Map every stack variable to a local variable, by determining the stack height at every instruction. Then map each instruction which acts implicitly on the stack variables to a 3-address code statement

[1] We currently use a tool called Coffi to achieve this translation.

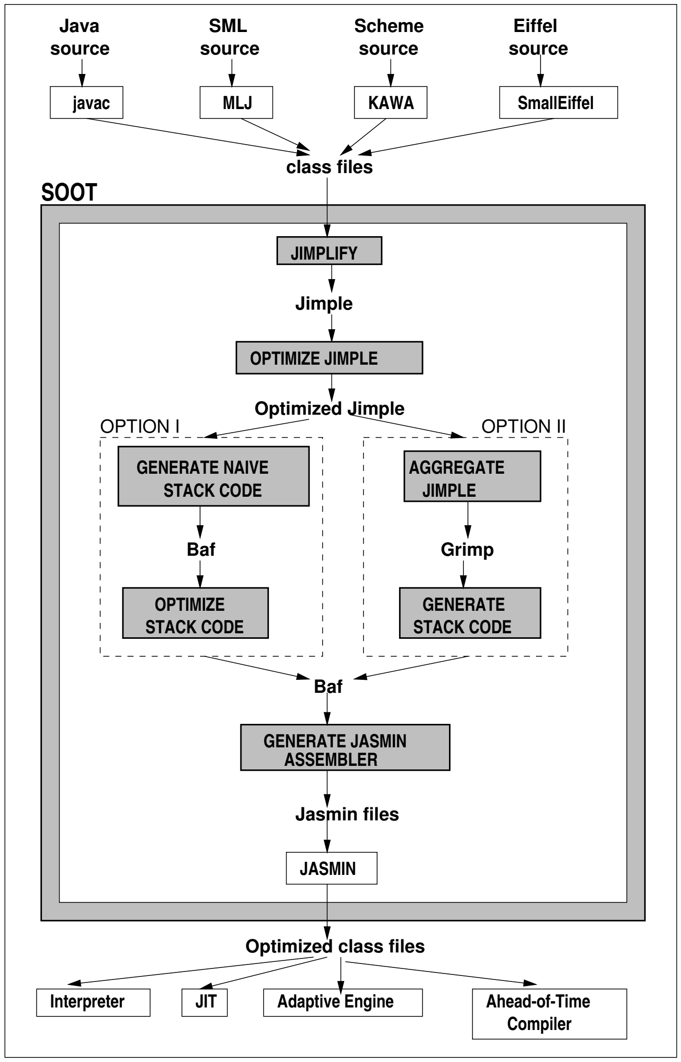

Fig. 1. Soot Framework Structure

which refers to the local variables explicitly. For example, if the current stack height is 3, then the instruction `iadd` would be translated to the Jimple statement `$i2 = $i2 + $i3`. This is a standard technique and is also used in other systems [18, 17].

2. *Type the local variables*: The resulting Jimple code may be untypable because a local variable may be used with two different types in different contexts. Thus, we split the uses and definitions of local variables according to webs[16]. This produces, in almost all cases, Jimple code whose local vari-

ables can be given a primitive, class, or interface type.[2] To do this, we invoke an algorithm described in [9]. The complete solution to this typing problem is NP-complete, but in practice simple polynomial algorithms suffice. Although splitting the local variables in this step produces many local variables, the resulting Jimple code tends to be easier to analyze because it inherits some of the disambiguation benefits of SSA form[5].

3. *Clean up the code*: Jimple code must now be compacted because step 1 produced extremely verbose code[18, 17]. We have found that simple aggregation (collapsing single def/use pairs) followed by copy propagation/elimination of stack variables to be sufficient to eliminate almost all redundant stack variables.

Figure 2(a) shows an input Java program and Figure 2(b) shows the bytecode generated by `javac` as we would represent it in our Baf representation. Figure 2(c) shows the Jimple code that would result from jimplifying the bytecode. Note that all local variables have been given types. Variables beginning with `$` correspond to variables that were inserted to stand for stack locations, whereas variables that do not begin with `$` correspond to local variables from the bytecode.

2.2 Optimize Jimple

Most of the analyses and transformations developed by our group, as well as other research groups, are implemented using the Jimple representation. We have currently implemented many intraprocedural and interprocedural optimizations. In this paper we focus on studying the effect of several simple intraprocedural techniques in conjunction with method inlining.

2.3 Convert Jimple back to Stack Code

Producing bytecode naively from Jimple code produces highly inefficient code. Even the best JIT that we tested can not make up for this inefficiency. And it is very important that we do not introduce inefficiencies at this point that would negate the optimizations performed on Jimple.

We have investigated two alternatives for producing stack code, labeled Option I and Option II in Figure 1. These two options are discussed in more detail in Section 3, and they are experimentally validated in Section 4.

2.4 Generate Jasmin Assembler

After converting Jimple to Baf stack code, the final phase is to generate Jasmin [11] assembler files from the internal Baf representation. Since Baf is a relatively direct encoding of the Java bytecode, this phase is quite simple.

[2] Other cases must be handled by introducing type casts and/or introducing extra copy statements.

```
Object[] a;
int x;

public void f(int i, int c)
{ g(x *= 10);
  while(i * 2 < 10)
    { a[i++] = new Object();
    }
}
```

(a) Original Java Source

```
.method public f(II)V
    aload_0
    aload_0
    dup
    getfield A/x I
    bipush 10
    imul
    dup_x1
    putfield A/x I
    invokevirtual A/g(I)V
    goto Label1

Label0:
    aload_0
    getfield A/a [Ljava/lang/Object;
    iload_1
    iinc 1 1
    new java/lang/Object
    dup
    invokenonvirtual
         java/lang/Object/<init>()V
    aastore

Label1:
    iload_1
    iconst_2
    imul
    bipush 10
    if_icmplt Label0

    return
```

(b) Bytecode

```
public void f(int, int)
{
  Example this;
  int i, c, $i0, $i1, $i2, $i3;
  java.lang.Object[] $r1;
  java.lang.Object $r2;

  this := @this;
  i := @parameter0;
  c := @parameter1;
X:$i0 = this.x;
X:$i1 = $i0 * 10;
  this.x = $i1;
  this.g($i1);
  goto label1;

label0:
Y:$r1 = this.a;
  $i2 = i;
  i = i + 1;
  $r2 = new java.lang.Object;
  specialinvoke $r2.<init>();
Y:$r1[$i2] = $r2;

label1:
Z:$i3 = i * 2;
Z:if $i3 < 10 goto label0;

  return;
}
```

(c) Jimple

Fig. 2. Translating bytecode to Jimple

3 Transformations

After optimizing Jimple code, it is necessary to translate back to efficient Java bytecode. Figure 3(a) illustrates the bytecode produced by a naive translation from Jimple.[3] Note that this naive code has many redundant store and load instructions reflecting the fact that Jimple computation results are stored to local variables, while stack-based bytecode can use the stack to store many intermediate computations.

[3] Note that our Baf representation closely mirrors Java bytecode. However, we do streamline the representation somewhat and preserve the variable names as much as possible.

3.1 Option I: Produce Naive Baf and Optimize

The main idea behind Baf optimizations is to identify and eliminate redundant store/load computations. Figure 3(b) shows the results of applying the Baf optimizations on the naive code given in Figure 3(a).

The optimizations currently in use are all performed on discrete basic blocks within a method. Inter-block optimizations that cross block boundaries have also been implemented, but to date these have not yielded any appreciable speedups on the runtime and are not described here. In the following discussion it is assumed that all instructions belong to the same basic block.

In practice, the majority of the redundant store and load instructions present in naive Baf code belong to one of the following code patterns:

store/load (sl pair) : a store instruction followed by a load instruction referring to the same local variable with no other uses. Both the store and load instructions can be eliminated, and the value will simply remain on the stack.

store/load/load (sll triple) : a store instruction followed by 2 load instructions, all referring to the same local variable with no other uses. The 3 instructions can be eliminated and a `dup` instruction introduced. The dup instruction replaces the second load by duplicating the value left on the stack after eliminating the store and the first load.

We can observe these patterns occurring in Figure 3(a) where labels A, B, C and E each identify distinct sl pairs, and where labels D and G identify sll triples.

To optimize a Baf method our algorithm performs a fixed point iteration over its basic blocks identifying and reducing these patterns whenever possible. By reducing, we mean eliminating an sl pair or replacing an sll triple to a dup instruction.

Reducing a pattern is trivial when all its instructions directly follow each other in the instruction stream. For example, the sl pairs at labels A and E in Figure 3(a) are trivial ones. However, often the store and load instructions are far apart (like at labels B and C.) To identify pairs of statements to reduce, we must determine the effect on the stack of the intervening bytecode. These bytecodes are called the *interleaving sequences*.

We compute two pieces of information as a means to analyse interleaving sequences and their effects/dependencies on the stack. The net effect on the stack height after executing a sequence of bytecode is referred to as the *net stack height variation* or *nshv*. The minimum stack height variation attained while executing a sequence of bytecode is referred to as the *minimum stack height variation* or *mshv*. A sequence of instructions having both $nshv = 0$ and $mshv = 0$ is referred to as a *level sequence*.

If interleaving sequences are level sequences, then the target patterns can be reduced directly. This is because one can just leave the value on the stack, execute the interleaving instructions, and then use the value that was left on the stack. This is the case for the sl pair labeled B, and later the sl pair labeled C, once B has been reduced. Finally, the sll triple D can be reduced, once both B and C have been reduced. In general, however, many such interleaving sequences will not be level and no reductions will be possible without some reordering of the block's bytecode.

```
public void f(int, int)
{ word this, i, c, $i2, $r2;

  this := @this: Example;
  i := @parameter0: int;
  c := @parameter1: int;
  load.r this;
  fieldget <Example: int x>;
A:store.i c;
A:load.i c;
  push 10;
  mul.i;
G:store.i c;
F:load.r this;
G:load.i c;
  fieldput <Example: int x>;
F:load.r this;
G:load.i c;
  virtualinvoke
     <Example: void g(int)>;
  goto label1;

label0:
  load.r this;
  fieldget
     <Example: java.lang.Object[] a>;
B:store.r c;
  load.i i;
C:store.i $i2;
  inc.i i 1;
  new java.lang.Object;
D:store.r $r2;
D:load.r $r2;
  specialinvoke
     <java.lang.Object: void <init>()>;
B:load.r c;
C:load.i $i2;
D:load.r $r2;
  arraywrite.r;

label1:
  load.i i;
  push 2;
  mul.i;
E:store.i $r2;
E:load.i $r2;
  push 10;
  ifcmplt.i label0;

  return;
}
```

(a) naive Baf generated from Jimple

```
public void f(int, int)
{ word this, i, c;

  this := @this: Example;
  i := @parameter0: int;
  c := @parameter1: int;
  load.r this;
F:load.r this;
F:load.r this;
  fieldget <Example: int x>;
  push 10;
  mul.i;
G:store.i c;
G:load.i c;
  fieldput <Example: int x>;
G:load.i c;
  virtualinvoke
     <Example: void g(int)>;
  goto label1;

label0:
  load.r this;
  fieldget
     <Example: java.lang.Object[] a>;
  load.i i;
  inc.i i 1;
  new java.lang.Object;
D:dup1.r;
  specialinvoke
     <java.lang.Object: void <init>()>;
  arraywrite.r;

label1:
  load.i i;
  push 2;
  mul.i;
  push 10;
  ifcmplt.i label0;

  return;
}
```

(b) optimized Baf

Fig. 3. Optimizing Baf

If an interleaving sequence has $nshv > 1$ for a sll triple or $nshv > 0$ for a sl pair, then our algorithm will try to lower the $nshv$ value by relocating a bytecode having a positive $nshv$ to an earlier location in the block. This is illustrated by the movement of instructions labeled by F in Figures 3(a) and 3(b) in an attempt to reduce the pattern identified by G. Another strategy used when $nshv < 0$ is to move level subsequence ending with the pattern's store instruction past the interleaving sequence. Of course, this can only be done if no data dependencies are violated.

Applying these heuristics produces optimized Baf code which becomes Java bytecode and is extremely similar to the original bytecode. We observe that except for two minor differences, the optimized Baf code in Figure 3(b) is the same as the original bytecode found in Figure 2(b). The differences are: (1) the second load labeled by F is not converted to a `dup`; and (2) the pattern identified by G is not reduced to a `dup_x1` instruction. We have actually implemented these patterns, but our experimental results did not justify enabling these extra transformations. In fact, introducing bytecodes such as `dup_x1` often yields non-standard, albeit legal, bytecode sequences that increase execution time and cause many JIT compilers to fail.

3.2 Option II: Build Grimp and Traverse

In this section, we describe the second route for translating Jimple into bytecode. The compiler `javac` is able to produce efficient bytecode because it has the structured tree representation of the original program, and the stack based nature of the bytecode is particularly well suited for code generation from trees[2]. Essentially, this phase attempts to recover the original structured tree representation, by building Grimp, an aggregated form of Jimple, and then producing stack code by standard tree traversal techniques.

Grimp is essentially Jimple but the expressions are trees of arbitrary depth. Figure 4(a) shows the Grimp version of the Jimple program in Figure 2(c).

Aggregation of bytecode The basic algorithm for aggregation is as follows. We consider pairs *(def, use)* in extended basic blocks, where *def* is an assignment statement with sole use *use*, and *use* has the unique definition *def*. We inspect the path between *def* and *use*, to determine if *def* can be safely moved into *use*. This means checking for dependencies and not moving across exception boundaries. We perform this algorithm iteratively, and the pairs are considered in reverse pseudo-topological order to cascade the optimizations as efficiently as possible. Examples of these aggregation opportunities are shown in figure 2(c) at the labels X, Y and Z. X and Z are trivial cases because their aggregation pairs are adjacent, but the pair at Y are a few statements apart. Thus before producing the aggregated code in 4(a) the interleaving statements must be checked for writes to `this.a`.

Peephole optimizations In some cases, Grimp cannot concisely express Java idioms. For example, the increment operation in `a[i++] = new Object();` cannot be expressed, because Grimp only allows a definition to occur as the left-hand

```
public void f(int, int)
{
  Example this;
  int i, c, $i1, $i2;

  this := @this;
  i := @parameter0;
  c := @parameter1;
X:$i1 = this.x * 10;
  this.x = $i1;
  this.g($i1);
  goto label1;

label0:
  $i2 = i;
  i = i + 1;
Y:this.a[$i2] =
        new java.lang.Object();

label1:
Z:if i * 2 < 10 goto label0;

  return;
}
```

(a) Grimp

```
public void f(int, int)
{ word this, i, c, $i2;

  this := @this: Example;
  i := @parameter0: int;
  c := @parameter1: int;
  load.r this;
  fieldget <Example: int x>;
  push 10;
  mul.i;
  store.i c;
  load.r this;
  load.i c;
  fieldput <Example: int x>;
  load.r this;
  load.i c;
  virtualinvoke
      <Example: void g(int)>;
  goto label1;

label0:
  load.r this;
  fieldget
      <Example: java.lang.Object[] a>;
  load.i i;
  inc.i i 1;
  new java.lang.Object;
  dup1.r
  specialinvoke
       <java.lang.Object: void <init>()>;
  arraywrite.r;

label1:
  load.i i;
  push 2;
  mul.i
  push 10;
  ifcmplt.i label0;

  return;
}
```

(b) Baf generated from Grimp

Fig. 4. Generating Baf from Grimp

side of an assignment statement, not as a side effect.[4] To remedy this problem, we use some peephole optimizations in the code generator for Grimp.

For example, for the increment case, we search for Grimp patterns of the form:

```
s1:    local = <lvalue>;
s2:    <lvalue> = local/<lvalue> + 1;
s3:    use(local)
```

and we ensure that the local defined in s_1 has exactly two uses, and that the uses in s_2, s_3 have exactly one definition. Given this situation, we emit code for only s_3. However, during the generation of code for s_3, when `local` is to be emitted,

[4] This design decision was made to simplify analyses on Grimp.

we also emit code to duplicate `local` on the stack, and increment `<lvalue>`. An example of this pattern occurs just after `Label0` in Figure 4(a).

This approach produces reasonably efficient bytecode. In some situations the peephole patterns fail and the complete original structure is not recovered. In these cases, the Baf approach usually performs better. See the section 4 for more details.

4 Experimental Results

Here we present the results of two experiments. The first experiment, discussed in Section 4.3, validates that we can pass class files through the framework, without optimizing the Jimple code, and produce class files that have the same performance as the original ones. In particular, this shows that our methods of converting from Jimple to stack-based bytecode are acceptable. The second experiment, discussed in Section 4.4, shows the effect of applying method inlining on Jimple code and demonstrates that optimizing Java bytecode is feasible and desirable.

4.1 Methodology

All experiments were performed on dual 400Mhz Pentium II^{TM} machines. Two operating systems were used, Debian GNU/Linux (kernel 2.2.8) and Windows NT 4.0 (service pack 5). Under GNU/Linux we ran experiments using three different configurations of the Blackdown Linux JDK1.2, pre-release version 2.[5] The configurations were: interpreter, Sun JIT, and a public beta version of Borland's JIT[6]. Under Windows NT, two different configurations of Sun's JDK1.2.2 were used: the JIT, and HotSpot (version 1.0.1)

Execution times were measured by running the benchmarks ten times, discarding the best and worst runs, and averaging the remaining eight. All executions were verified for correctness by comparing the output to the expected output.

4.2 Benchmarks and Baseline Times

The benchmarks used consist of seven of the eight standard benchmarks from the SPECjvm98[7] suite, plus three additional applications from our collection. See figure 5. We discarded the *mtrt* benchmark from our set because it is essentially the same benchmark as *raytrace*. The *soot-c* benchmark is based on an older version of Soot, and is interesting because it is heavily object oriented, *schroeder-s* is an audio editing program which manipulates sound files, and *jpat-p* is a protein analysis tool.

Figure 5 also gives basic characteristics such as size, and running times on the five platforms. All of these benchmarks are real world applications that are

[5] http://www.blackdown.org
[6] http://www.borland.com
[7] http://www.spec.org/

reasonably sized, and they all have non-trivial execution times. We used the Linux interpreter as the base time, and all the fractional execution times are with respect to this base.

Benchmarks for which a dash is given for the running time indicates that the benchmark failed validity checks. In all these cases, the virtual machine is to blame as the programs run correctly with the interpreter with the verifier explicitly turned on. Arithmetic averages and standard deviations are also given, and these automatically exclude those running times which are not valid.

For this set of benchmarks, we can draw the following observations. The Linux JIT is about twice as fast as the interpreter but it varies widely depending on the benchmark. For example, with *compress* it is more than six times faster, but for a benchmark like *schroeder-s* it is only 56% faster. The NT virtual machines also tend to be twice as fast as the Linux JIT. Furthermore, the performance of the HotSpot performance engine seems to be, on average, not that different from the standard Sun JIT. Perhaps this is because the benchmarks are not long running server side applications (i.e. not the kinds of applications for which HotSpot was designed).

4.3 Straight through Soot

Figure 6 compares the effect of processing applications with Soot with Baf and Grimp, without performing any optimizations. Fractional execution times are given, and these are with respect to the original execution time of the benchmark for a given platform. The ideal result is 1.00. This means that the same performance is obtained as the original application. For *javac* the ratio is .98 which indicates that *javac*'s execution time has been reduced by 2%. *raytrace* has a ratio of 1.02 which indicates that it was made slightly slower; its execution time has been increased by 2%. The ideal arithmetic averages for these tables is 1.00 because we are trying to simply reproduce the program as is. The ideal standard deviation is 0 which would indicate that the transformation is having a consistent effect, and the results do not deviate from 1.00.

On average, using Baf tends to reproduce the original execution time. Its average is lower than Grimp's, and the standard deviation is lower as well. For the faster virtual machines (the ones on NT), this difference disappears. The main disadvantage of Grimp is that it can produce a noticeable slowdown for benchmarks like *compress* which have tight loops on Java statements containing side effects, which it does not always catch.

Both techniques have similar running times, but implementing Grimp and its aggregation is conceptually simpler. In terms of code generation for Java virtual machines, we believe that if one is interested in generating code for slow VMs, then the Baf-like approach is best. For fast VMs, or if one desires a simpler compiler implementation, then Grimp is more suitable.

We have also measured the size of the bytecode before and after processing with Soot. The sizes are very similar, with the code after processing sometimes slightly larger, and sometimes slightly smaller. For the seven SPECjvm benchmarks the total bytecode size was 0.5% larger after processing with Soot. This is not a significant increase in size.

4.4 Optimization via Inlining

We have investigated the feasibility of optimizing Java bytecode with Soot by implementing method inlining. Although inlining is a whole program optimization, Soot is also suitable for optimizations applicable to partial programs. Our approach to inlining is simple. We build an invoke graph using class hierarchy analysis[8] and inline method calls whenever they resolve to one method. Our inliner is a bottom-up inliner, and attempts to inline all call sites subject to the following restrictions: 1) the method to be inlined must contain less than 20 Jimple statements, 2) no method may contain more than 5000 Jimple statements, and 3) no method may have its size increased more than by a factor of 3.

After inlining, the following traditional intraprocedural optimizations are performed to maximize the benefit from inlining: copy propagation, constant propagation and folding, conditional and unconditional branch folding, dead assignment elimination and unreachable code elimination. These are described in [2].

Figure 7 gives the result of performing this optimization. The numbers presented are fractional execution times with respect to the original execution time of the benchmark for a given platform. For the Linux virtual machines, we obtain a significant improvement in speed. In particular, for the Linux Sun JIT, the average ratio is .92 which indicates that the average running time is reduced by 8%. For raytrace, the results are quite significant, as we obtain a ratio of .62, a reduction of 38%.

For the virtual machines under NT, the average is 1.00 or 1.01, but a number of benchmarks experience a significant improvement. One benchmark, *javac* under the Sun JIT, experiences significant degradation. However, under the same JIT, *raytrace* yields a ratio of .89, and under HotSpot, *javac*, *jack* and *mpegaudio* yield some improvements. Given that HotSpot itself performs dynamic inlining, this indicates that our static inlining heuristics sometimes capture opportunities that HotSpot does not. Our heuristics for inlining were also tuned for the Linux VMs, and future experimentation could produce values which are better suited for the NT virtual machines.

These results are highly encouraging as they strongly suggest that a significant amount of improvement can be achieved by performing aggressive optimizations which are not performed by the virtual machines.

5 Related Work

Related work falls into five different categories:

Java bytecode optimizers: There are only two Java tools of which we are aware that perform significant optimizations on bytecode and produce new class files: Cream[3] and Jax[23]. Cream performs optimizations such as loop invariant removal and common sub-expression elimination using a simple side effect analysis. Only extremely small speed-ups (1% to 3%) are reported, however. The main goal of Jax is application compression where, for example, unused methods and fields are removed, and the class hierarchy is

	# Jimple Stmts	Linux Sun Int. (secs)	Linux Sun JIT	Linux Bor. JIT	NT Sun JIT	NT Sun Hot.
compress	7322	440.30	.15	.14	**.06**	.07
db	7293	259.09	.56	.58	.26	**.14**
jack	16792	151.39	.43	.32	**.15**	.16
javac	31054	137.78	.52	.42	**.24**	.33
jess	17488	109.75	.45	.32	.21	**.12**
jpat-p	1622	47.94	1.01	.96	.90	**.80**
mpegaudio	19585	368.10	.15	-	**.07**	.10
raytrace	10037	121.99	.45	.23	.16	**.12**
schroeder-s	9713	48.51	.64	.62	.19	**.12**
soot-c	42107	85.69	.58	.45	**.29**	.53
average			.49	.45	.25	.25
std. dev.			.23	.23	.23	.23

Fig. 5. Benchmarks and their characteristics.

	Baf Linux Sun Int.	Baf Linux Sun JIT	Baf Linux Bor. JIT	Baf NT Sun JIT	Baf NT Sun Hot.	Grimp Linux Sun Int.	Grimp Linux Sun JIT	Grimp Linux Bor. JIT	Grimp NT Sun JIT	Grimp NT Sun Hot.
compress	1.01	1.00	.99	.99	1.00	1.07	1.02	1.04	1.00	1.01
db	.99	1.01	1.00	1.00	1.00	1.01	1.05	1.01	1.01	1.02
jack	1.00	1.00	1.00	-	1.00	1.01	.99	1.00	-	1.00
javac	1.00	.98	1.00	1.00	.97	.99	1.03	1.00	1.00	.95
jess	1.02	1.01	1.04	.99	1.01	1.01	1.02	1.04	.97	1.00
jpat-p	1.00	.99	1.00	1.00	1.00	.99	1.01	1.01	1.00	1.00
mpegaudio	1.05	1.00	-	-	1.00	1.03	1.00	-	-	1.01
raytrace	1.00	1.02	1.00	.99	1.00	1.01	1.00	.99	.99	1.00
schroeder-s	.97	1.01	-	1.03	1.01	.98	.99	-	1.03	1.00
soot-c	.99	1.00	1.02	.99	1.03	1.00	1.01	1.00	1.01	1.01
average	1.00	1.00	1.01	1.00	1.00	1.01	1.01	1.01	1.00	1.00
std. dev.	.02	.01	.01	.01	.01	.02	.02	.02	.02	.02

Fig. 6. The effect of processing classfiles with Soot using Baf or Grimp, without optimization.

compressed. They also are interested in speed optimizations, but at this time their current published speed up results are extremely limited.

Bytecode manipulation tools: There are a number of Java tools which provide frameworks for manipulating bytecode: JTrek[13], Joie[4], Bit[14] and JavaClass[12]. These tools are constrained to manipulating Java bytecode in their original form, however. They do not provide convenient intermediate representations such as Baf, Jimple or Grimp for performing analyses or transformations.

Java application packagers: There are a number of tools to package Java applications, such as Jax[23], DashO-Pro[6] and SourceGuard[21]. Application

	Linux			NT	
	Sun Int.	Sun JIT	Bor. JIT	Sun JIT	Sun Hot.
compress	1.01	.78	1.00	1.01	.99
db	.99	1.01	1.00	1.00	1.00
jack	1.00	.98	.99	-	.97
javac	.97	.96	.97	1.11	.93
jess	.93	.93	1.01	.99	1.00
jpat-p	.99	.99	1.00	1.00	1.00
mpegaudio	1.04	.96	-	-	.97
raytrace	.76	.62	.74	.89	1.01
schroeder-s	.97	1.00	.97	1.02	1.06
soot-c	.94	.94	.96	1.03	1.05
average	.96	.92	.96	1.01	1.00
std. dev.	.07	.12	.08	.06	.04

Fig. 7. The effect of inlining with class hierarchy analysis.

packaging consists of code compression and/or code obfuscation. Although we have not yet applied Soot to this application area, we have plans to implement this functionality as well.

Java native compilers: The tools in this category take Java applications and compile them to native executables. These are related because they all are forced to build 3-address code intermediate representations, and some perform significant optimizations. The simplest of these is Toba[18] which produces unoptimized C code and relies on GCC to produce the native code. Slightly more sophisticated, Harissa[17] also produces C code but performs some method devirtualization and inlining first. The most sophisticated systems are Vortex[7] and Marmot[10]. Vortex is a native compiler for Cecil, C++ and Java, and contains a complete set of optimizations. Marmot is also a complete Java optimization compiler and is SSA based. There are also numerous commercial Java native compilers, such as the IBM (R) High Performance Compiler for Java, Tower Technology's TowerJ[24], and SuperCede[22], but they have very little published information.

Stack code optimization: Some research has been previously done in the field of optimizing stack code. The work presented in [15] is related but optimizes the stack code based on the assumption that stack operations are cheaper than manipulating locals. This is clearly not the case for the Java Virtual Machines we are interested in. On the other hand, some closely related work has been done on optimizing naive Java bytecode code at University of Maryland[19]. Their technique is similar, but they present results for only toy benchmarks.

6 Conclusions and Future Work

We have presented Soot, a framework for optimizing Java bytecode. Soot consists of three intermediate representations (Baf, Jimple & Grimp), transformations between these IRs, and a set of optimizations on these intermediate representations.

In this paper we have given an overview of the structure of Soot, concentrating on the mechanisms for translating Java stack-based bytecode to our typed three-address representation Jimple, and the translation of Jimple back to bytecode. Jimple was designed to make the implementation of compiler analyses and transformations simple. Our experimental results show that we can perform the conversions without losing performance, and so we can effectively optimize at the Jimple level. We demonstrated the effectiveness of a set of intraprocedural transformations and inlining on five different Java Virtual Machine implementations.

We are encouraged by our results so far, and we have found that the Soot APIs have been effective for a variety of tasks including the optimizations presented in this paper.

We, and other research groups, are actively engaged in further work on Soot on many fronts. Our group is currently focusing on new techniques for virtual method resolution, pointer analyses, side-effect analyses, and various transformations that can take advantage of accurate side-effect analysis.

Acknowledgements

This research was supported by IBM's Centre for Advanced Studies (CAS), NSERC and FCAR.

References

1. Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast and effective code generation in a just-in-time Java compiler. *ACM SIGPLAN Notices*, 33(5):280–290, May 1998.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, 1986.
3. Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice & Experience*, 9(11):1031–1045, November 1997.
4. Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178, Berkeley, USA, June 15–19 1998. USENIX Association.
5. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.
6. DashOPro. http://www.preemptive.com/products.html.
7. Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. VORTEX: An optimizing compiler for object-oriented languages. In *Proceedings OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31 of *ACM SIGPLAN Notices*, pages 83–100. ACM, October 1996.
8. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, 7–11 August 1995. Springer.

9. Étienne Gagnon and Laurie Hendren. Intra-procedural Inference of Static Types for Java Bytecode. Sable Technical Report 1999-1, Sable Research Group, McGill University, March 1999.
10. Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an Optimizing Compiler for Java. Microsoft technical report, Microsoft Research, October 1998.
11. Jasmin: A Java Assembler Interface. http://www.cat.nyu.edu/meyer/jasmin/.
12. JavaClass. http://www.inf.fu-berlin.de/ dahm/JavaClass/ .
13. Compaq JTrek. http://www.digital.com/java/download/jtrek .
14. Han Bok Lee and Benjamin G. Zorn. A Tool for Instrumenting Java Bytecodes. In *The USENIX Symposium on Internet Technologies and Systems*, pages 73–82, 1997.
15. Martin Maierhofer and M. Anton Ertl. Local stack allocation. In Kai Koskimies, editor, *Compiler Construction (CC'98)*, pages 189–203, Lisbon, 1998. Springer LNCS 1383.
16. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
17. Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Berkeley, June 16–20 1997. Usenix Association.
18. Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A way ahead of time (WAT) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 41–54, Berkeley, June 16–20 1997. Usenix Association.
19. Tatiana Shpeisman and Mustafa Tikir. Generating Efficient Stack Code for Java. Technical report, University of Maryland, 1999.
20. Soot - a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.
21. 4thpass SourceGuard. http://www.4thpass.com/sourceguard/.
22. SuperCede, Inc. SuperCede for Java. http://www.supercede.com/.
23. Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical Experience with an Application Extractor for Java . IBM Research Report RC 21451, IBM Research, 1999.
24. Tower Technology. Tower J. http://www.twr.com/.
25. Raja Vallée-Rai and Laurie J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations. Sable Technical Report 1998-4, Sable Research Group, McGill University, July 1998.