# Soot - a Java Bytecode Optimization Framework*

Raja Vallée-Rai Phong Co Etienne Gagnon
Laurie Hendren Patrick Lam Vijay Sundaresan

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 bytecode which is simple to manipulate; Jimple, a typed 3-address intermediate representation suitable for optimization; and Grimp, an aggregated version of Jimple suitable for decompilation. We describe the motivation for each representation, and the salient points in translating from one representation to another.

In order to demonstrate the usefulness of the framework, we have implemented intraprocedural and whole program optimizations. To show that whole program bytecode optimization can give performance improvements, we provide experimental results for 12 large benchmarks, including 8 SPECjvm98 benchmarks running on JDK 1.2 for GNU/Linux$_{\mathrm{tm}}$. These results show up to 8% improvement when the optimized bytecode is run using the interpreter and up to 21% when run using the JIT compiler.

## 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. Using a Just-In-Time compiler[1], or a Way-Ahead-Of-Time Java compiler[21] [20], to convert the bytecodes to native instructions is the most often used method for improving performance. There are other types of optimizations, however, which can have a substantial impact on performance:

- *Optimizing the bytecode directly:* Some bytecode instructions are much more expensive than others. For example, loading a local variable onto the stack is inexpensive; but virtual methods calls, interface calls, object allocations, and catching exceptions are all expensive. Traditional C-like optimizations, such as copy propagation, have little effect because they do not target the expensive bytecodes. To perform effective optimizations at this level, one must consider more advanced optimizations such as method inlining, and static virtual method call resolution, which directly reduce the use of these expensive bytecodes.
- *Annotating the bytecode:* Java's execution safety feature guarantees that all potentially illegal memory accesses are checked for safety before execution. In some situations it can be determined at compile-time that particular checks are unnecessary. For example, many array bound checks can be determined to be completely unnecessary[14]. Unfortunately, after having determined the safety of some ar-

*The authors' e-mail addresses are: {rvalleerai, pco, gagnon, hendren, plam, vijay}@sable.mcgill.ca, respectively. This research was supported by IBM's Centre for Advanced Studies (CAS), NSERC and FCAR. Java is a trademark of Sun Microsystems Inc.

ray accesses, we can not eliminate the bounds checks directly from the bytecode, because they are implicit in the array access bytecodes and can not be separated out. But if we can communicate the safety of these instructions to the Java Virtual Machine by some annotation mechansim, then the Java Virtual Machine could speed up the execution by not performing these redundant checks.

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[24] framework which provides a set of intermediate representations and a set of Java APIs for optimizing Java bytecode directly. The optimized bytecode can be executed using any standard Java Virtual Machine (JVM) implementation, or it could be used as the input to a bytecode→C or bytecode→native-code compiler.

Based on the Soot framework we have implemented both intraprocedural optimizations and whole program optimizations. The framework has also been designed so that we will be able to add support for the annotation of Java bytecode. We have applied our tool to a substantial number of large benchmarks, and the best combination of optimizations implemented so far can yield a speed up reaching 21%.

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 intraprocedural and whole program optimizations to a set of real Java applications.

The rest of the paper is organized as follows. Section 2 gives an overview of the framework. Section 3 describes the three intermediate representations used, in detail. Section 4 describes the steps required to transform bytecode from one intermediate representation to another. Section 5 describes the current optimizations present in the Soot framework. Section 6 presents and discusses our experimental results. Section 7 covers the conclusions and related work.

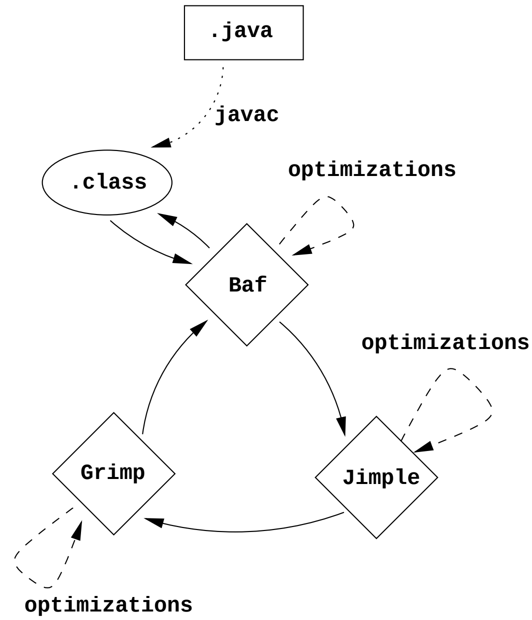


Figure 1: The Soot Optimization Framework consists of three intermediate representations: Baf, Jimple and Grimp.

## 2 Framework Overview

The Soot framework has been designed to simplify the process of developing new optimizations for Java bytecode. To do this we have developed three intermediate representations: Baf, a streamlined representation of bytecode which is simple to manipulate; Jimple, a typed 3-address intermediate representation suitable for optimization; and Grimp, an aggregated version of Jimple suitable for decompilation. Section 3 gives more detailed descriptions of each intermediate representation.

Optimizing Java bytecode in Soot consists of transforming Java bytecode subsequently to Baf, Jimple, Grimp, back to Baf, and then to bytecode, and while in each representation, performing some appropriate optimization (see figure 1.)

## 3 Intermediate Representations

Baf, Jimple, and Grimp are three unstructured representations for Java bytecode. They were developed to allow optimizations and analyses to be performed on Java bytecode at the most appropriate level. Each representation is discussed in more detail below, and figures 2, 3, 4 and 5 provide an example program in each form.

## 3.1 Baf

### Motivation

Although the primary goal of the Soot framework is to avoid having to deal with bytecode as stack code, it is still sometimes necessary to analyze or optimize bytecode in this form. We use it in two ways. First, in order to produce Jimple code, it is necessary to perform abstract interpretation on the stack. Second, before producing new bytecode, it is convenient to perform some peephole optimizations and stack manipulations to eliminate redundant load/stores. These analyses and transformations could be performed on Java bytecode directly, but this is tedious for two reasons: encoding issues, and untyped bytecodes.

One of the many encoding issues to deal with is the constant pool. Bytecode instructions must refer to indices in this pool in order to access fields, methods, classes, and constants, and this constant pool must be tediously maintained. Baf abstracts away the constant pool, and thus it is easier to manipulate Baf code.

Another problem with Java bytecode is the presence of untyped bytecodes. A large proportion of the bytecodes are fully typed, in the sense that their effect on the stack is made explicit by the opcode. For example, the `iload` instruction indicates that an integer is loaded on to the stack. A few instructions, however, have been left untyped. Two examples are `dup` and `swap`. In order to determine their exact effect one must already know what types are on the stack, and thus typed stack interpretation must be performed. This step can be avoided by using Baf because each Baf instruction has an explicit type.

### Description

Baf is a bytecode representation which is stack based, but without the complications that are present in Java bytecode. Baf consists of a set of orthogonal instructions which are fully typed to simplify analyses and transformations. We have also developed an assembler format for Baf to allow easy modifications to class files in this form. An example of a Baf program is given in figure 3. Note that the local variables are given explicit names, and that there are some identity instructions at the beginning of the method to mark the local variables with pre-defined meanings, such as `this`.

## 3.2 Jimple

### Motivation

Optimizing stack code directly is awkward for multiple reasons, even if the code is in a streamlined form such as Baf. 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[30]. For example, a simple instruction such as `and` 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 confuses some analyses which expect explicitly typed variables. A fourth problem is the `jsr` contained in the bytecode instruction set. The `jsr` bytecode is difficult to handle because it is essentially a interprocedural feature which is inserted into a traditionally intraprocedural context.

### Description

Jimple is a 3-address code representation of bytecode, which is typed and does not include the 3-address code equivalent of a `jsr` (`jsr` instructions are allowed in the input bytecode, and are eliminated in the generated Jimple code). It is an ideal form for performing optimizations and analyses, both traditional optimizations such as copy propagation and more advanced optimizations such as virtual method resolution that object-oriented languages such as Java require. Essentially, the stack has been eliminated and replaced by additional local variables. Additionally, the typed nature of the bytecode and the untyped nature of the local and stack area variables have been reversed; the operators in Jimple are untyped, but the local variables are given explicit primitive, class, or interface types. And finally the `jsr` bytecode has been eliminated by subroutine replication, a technique which causes in practice very little code growth, but greatly simplifies analyses and transformations.

```
public int stepPoly(int x)
{
    if(x < 0)
    {
        System.out.println("foo");
        return -1;
    }
    else if(x <= 5)
        return x * x;
    else
        return x * 5 + 16;
}
```

Figure 2: `stepPoly` in its original Java form.

```
public int 'stepPoly'(int)
{
    word r0, i0

    r0 := @this
    i0 := @parameter0
    load.i i0
    ifge label0
    staticget java.lang.System.out
    push "foo"
    virtualinvoke println
    push -1
    return.i

 label0:
    load.i i0
    push 5
    ifcmpgt.i label1
    load.i i0
    load.i i0
    mul.i
    return.i

 label1:
    load.i i0
    push 5
    mul.i
    push 16
    add.i
    return.i
}
```

Figure 3: `stepPoly` in Baf form.

```
public int 'stepPoly'(int)
{
    Test r0;
    int i0, $i1, $i2, $i3;
    java.io.PrintStream $r1;

    r0 := @this;
    i0 := @parameter0;
    if i0 >= 0 goto label0;

    $r1 = java.lang.System.out;
    $r1.println("foo");
    return -1;

 label0:
    if i0 > 5 goto label1;

    $i1 = i0 * i0;
    return $i1;

 label1:
    $i2 = i0 * 5;
    $i3 = $i2 + 16;
    return $i3;
}
```

Figure 4: `stepPoly` in Jimple form. Dollar signs indicate local variables representing stack positions.

```
public int stepPoly(int)
{
    Test r0;
    int i0;

    r0 := @this;
    i0 := @parameter0;
    if i0 >= 0 goto label0;

    java.lang.System.out.println("foo");
    return -1;

 label0:
    if i0 > 5 goto label1;

    return i0 * i0;

 label1:
    return i0 * 5 + 16;
}
```

Figure 5: `stepPoly` in Grimp form.

Figure 4 shows the Jimple code generated for the example program. Note that all local variables are given explicit types, and that variables representing stack positions are prefixed with $.

### 3.3 Grimp

#### Motivation

One of the common problems in dealing with intermediate representations is that they are difficult to read because they do not resemble structured languages. In general, they contain many goto's and expressions are extremely fragmented. Another problem is that despite its simple form, for some analyses, 3-address code is sometimes harder to deal with than complex structures. For example, we found that generating good stack code was simpler when large expressions were available.

#### Description

Grimp is an unstructured representation of Java bytecode which allows trees to be constructed for expressions as opposed to the flat expressions present in Jimple. In general, it is much easier to read than Baf or Jimple, and for code generation, especially when the target is stack code, it is a much better source representation. It also has a representation for the `new` operator in Java which combines the `new` bytecode instruction with the `invokespecial` bytecode instruction. Essentially, Grimp looks like a partially decompiled Java code and we are also using Grimp as the foundation for a decompiler. Figure 5 gives the example program in Grimp format. Note that the Grimp code is extremely similar to the original Java code, and that the statements `$i2 = i0 * 5; $i3 = $i2 + 16; return $i3;` have been collapsed down to `return i0 * 5 + 16`.

## 4 Transformations

This section describes the steps necessary to transform Java bytecode from one intermediate representation to another (including to and from the native Java bytecode representation.)

### 4.1 bytecode $\longrightarrow$ Baf

Most of the bytecodes correspond directly to equivalent Baf instructions. The only difficulty lies in giving types to the `dup` and `swap` instructions, which is required because Baf is fully typed. This can always be performed because the Java Virtual Machine guarantees that the types on the stack at every program point can be determined statically[18]. This is done by performing a simple stack simulation.

### 4.2 Baf $\longrightarrow$ Jimple

Producing Jimple code from Baf is a multistep operation:

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 which refers to the local variables explicitly. This is a standard technique and is covered in detail in [21] and [20].

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. So the next step is to split the uses and definitions of local variables according to webs[19]. This produces, in almost all cases, Jimple code whose local variables can be given a primitive, class, or interface type. To do this, we invoke an algorithm described in [12]. The complete solution to this typing problem is non-trivial as it is NP-complete, but in practice heuristics 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[8].

3. *Clean up the code*: Now that the Jimple code is typed, it remains to be compacted because step 1 produces extremely verbose code[21] [20]. Although constant propagation, copy propagation, and even back copy propagation[23] are suggested techniques to be used at this point, we have found that simple aggregation (collapsing single def/use

pairs) to be sufficient to eliminate almost all redundant stack variables.

### 4.3 Jimple ⟶ Grimp

Producing Grimp code from Jimple is straightforward given that Grimp is essentially Jimple with arbitrarily nested expressions instead of references to locals.

1. *Aggregate expressions*: for every single def/use pair, attempt to move the right hand side of the definition into the use. Currently we only consider def-use pairs which reside in the same extended basic block, but our results indicate that almost all pairs are caught. Some care must be taken to guard against violating data dependencies or producing side effects when moving the right hand side of the definition.

2. *Fold constructors*: pairs consisting of `new` and `specialinvoke` are collapsed into one Grimp expression called `newinvoke`.

3. *Aggregate expressions*: folding the constructors usually exposes additional aggregation opportunities. These are taken advantage of in this second aggregation step.

The Grimp code generated by these three steps is extremely similar to the original Java source code; almost all introduced stack variables have been eliminated. Statements from Java which have multiple local variable definitions, however, cannot be represented as compactly, and this complicates Baf code generation. An example is given below.

```
                    $stack = j;
  a[j++] = j        j = j + 1;
                    r1[$stack] = j;

 Java code          Grimp code
```

### 4.4 Grimp ⟶ Baf

Generating Baf code from Grimp is straightforward because Grimp consists of tree statements and Baf is a stack-based representation. Standard code generation techniques for stack machines are used here[2].

The code generated in some cases by this tree traversal is inefficient compared to the original Java bytecode. This usually occurs when the original Java source contained compact C-like constructs such as in the example above. This inefficiency may have a significant impact on the program execution time if it occurs in loops. We delegate the optimization of these cases to the next transformation.

### 4.5 Baf ⟶ bytecode

Producing bytecode from Baf requires 4 steps:

1. *Pack local variables*: the local variables produced from Grimp assumed that they had to be typed as in Jimple. Baf has however only two types (`word`, `dword`) and so many less variables can actually be used by making some variables with different types share the same name. To rename these local variables, we use a simple register allocation scheme based on interference graphs[19].

2. *Optimize load/stores*: the Baf code produced from Grimp may have some redundant load and stores. These come from bytecode which originate from Java statements which have multiple local variable definitions, as in the previous example.

3. *Compute maximum stack height*: the Java Virtual Machine requires that the maximum stack height for each method be given. This can be computed by performing a simple depth first traversal of the Baf code.

4. *Produce the bytecode*: every Baf instruction is then converted to the corresponding bytecode instruction.

## 5 Optimizations

We have implemented a set of traditional scalar optimizations. They are described in [2], [19], and [3], and consist of:

- constant propagation and folding;
- conditional and unconditional branch elimination;
- copy propagation;
- dead assignment and unreachable code elimination;
- expression aggregation.

Based on our framework we plan to add further optimizations such as common sub-expression elimination and loop invariant removal. For best effect these optimizations need side-effect analysis, and two varieties of side-effect analysis are currently being developed.

At the heart of whole program optimization for object oriented languages is the call graph. The call graph contains information about the possible targets of virtual method calls. In general, a call graph with more precision enables more effective whole program optimization of an application. We have 3 methods for constructing the call graph which involve different time/space trade-offs:

- class hierarchy analysis[11];
- rapid type analysis[5];
- variable type analysis[26].

After building the call graph, we currently perform only one optimization: method inlining. We have also experimented with static method binding, where virtual dispatches are replaced with static dispatches, whenever possible. Unfortunately, preliminary results produced slowdowns and are undergoing investigation.

## 6 Experimental Results

### Methodology

In order to demonstrate the usefulness of our framework, we have optimized a collection of large Java applications. The experimental results that we present in this paper are speed ups: the ratio of original execution time to the execution time after an application has been processed with Soot. All experiments were performed on dual 400Mhz Pentium $\mathrm{II}_{\mathrm{tm}}$ machines running GNU/Linux, with the blackdown release of Linux JDK1.2 pre-release version 1[1]. Execution times were measured by taking the best of five runs on an unloaded machine, and we report results for both the interpreter and the JIT compiler. All executions were verified for correctness by comparing the output of the modified program to the output of the original application. Although not built for speed, Soot is very usable; optimizing class files averages about 2.5 times the execution time of javac to compile them.

[1] http://www.blackdown.org

The benchmarks used consist of the SPECjvm98[2] suite, plus two additional applications, each performing two different runs. The benchmarks *sablecc-j* and *sablecc-w* are runs of the SableCC[22] compiler compiler, and *soot-c* and *soot-j* are benchmarks based on an older version of Soot. Figure 6 consists of benchmark characteristics and speed-up results. As indicated by the figure, all of the benchmarks have non-trivial execution times, and are reasonably sized applications, represented by thousands of Jimple statements. We also include, as a benchmark characteristic, the speed-up obtained by using the JIT over the interpreter. Note that the four benchmarks that we have added to the JVM98 Suite are not sped up by the JIT nearly as much as the others, and clearly have different performance bottlenecks.

We present three columns of speed up results, which result from processing the application classfiles in different ways:

$\longrightarrow$: Processes the files through Soot without attempting to optimize them. The ideal result is 1.00.

**-O**: Performs our set of intraprocedural optimizations on each method in isolation.

**-W**: Transforms all the application class files and performs whole program optimizations. Essentially, class hierarchy analysis is used to build the call graph, and then ilining is performed. Then -O is performed on each method, exploiting the optimization opportunities that inlining may have exposed.

### Observations

$\longrightarrow$: Currently, processing class files with Soot, without performing any optimizations does not produce bytecode of the same caliber as the original bytecode. As mentioned in section 4.3, the current technique used to generate code from Grimp may introduce redundant loads and stores. This is particularly problematic with *_201_compress*, because redundant loads and stores are introduced into a critical loop. Interestingly, the JIT is unaffected by the verbose code; this suggests that it performs some form of copy propagation after it has converted the bytecode to register code.

**-O**: Turning on intraprocedural optimizations manages to produce files which have nearly identical execution time as the originals. Since these are scalar optimizations that are orthogonal to the

[2] http://www.spec.org/

| | # Jimple Stmts | Base Execution Int. | JIT | Int./JIT | Speed up: ⟶ Int. | JIT | Speed up: -O Int. | JIT | Speed up: -W Int. | JIT |
|---|---|---|---|---|---|---|---|---|---|---|
| *_201_compress* | 3562 | 441s | 67s | 6.6 | 0.86 | 0.97 | 1.00 | 1.00 | 0.98 | 1.21 |
| *_202_jess* | 13697 | 109s | 48s | 2.3 | 0.97 | 0.99 | 0.99 | 0.98 | 1.03 | 1.03 |
| *_205_raytrace* | 6302 | 125s | 54s | 2.3 | 0.99 | 0.99 | 1.00 | 0.97 | 1.08 | 1.10 |
| *_209_db* | 3639 | 229s | 130s | 1.8 | 0.98 | 1.03 | 1.01 | 1.02 | 1.00 | 1.03 |
| *_213_javac* | 26656 | 135s | 68s | 2.0 | 0.99 | 1.01 | 1.00 | 1.00 | 1.01 | 1.00 |
| *_222_mpegaudio* | 15244 | 374s | 54s | 6.9 | 0.94 | 0.97 | 0.99 | 1.00 | 0.96 | 1.05 |
| *_227_mtrt* | 6307 | 129s | 57s | 2.3 | 0.99 | 1.01 | 1.00 | 0.99 | 1.07 | 1.10 |
| *_228_jack* | 13234 | 144s | 61s | 2.4 | 0.99 | 0.97 | 0.99 | 0.99 | 1.00 | 0.98 |
| *sablecc-j* | 25344 | 45s | 30s | 1.5 | 0.98 | 1.01 | 0.99 | 0.99 | 1.00 | 1.04 |
| *sablecc-w* | 25344 | 70s | 38s | 1.8 | 1.00 | 1.00 | 1.00 | 1.01 | 0.98 | 1.04 |
| *soot-c* | 39938 | 85s | 49s | 1.7 | 0.98 | 0.99 | 0.98 | 1.00 | 1.03 | 0.96 |
| *soot-j* | 39938 | 184s | 126s | 1.5 | 0.98 | 0.99 | 0.99 | 0.99 | 1.02 | 1.01 |

Figure 6: Benchmark characteristics and speed-up results. ⟶, -O and -W represent no optimizations, intraprocedural optimizations, and whole program optimizations, respectively. The programs were executed on a 400Mhz dual Pentium II machine running GNU/Linux with Linux JDK1.2, pre-release version 1.

elimination of the redundant loads and stores, we expect that -O will produce a net speed up of 2-3% when our Baf optimizations are ready. Perhaps the effect will be greater for *_201_compress*, since the scalar optimizations seem to compensate for a slowdown of .14 due to redundant loads and stores. On *_201_compress* the optimizations have practically no effect on the JIT, suggesting that the JIT is already performing these optimizations. In general, intraprocedural optimizations have very little effect on Java bytecode, which is what we expect, given that our current intraprocedural optimizations can only work on scalar operations which are relatively inexpensive.

**-W**: Performing whole program optimizations does produce significant speed ups. In general, the effect of devirtualization and of inlining is more pronounced under the JIT than the interpreter, due to the increased relative cost of virtual dispatches, and the fact that larger method bodies are available to the JIT optimizer. This is particularly noticeable for *_201_compress* for which inlining yields no speed up under the interpreter, but exhibits a 21% speed up under the JIT. *soot-j* illustrates the danger of code expansion with JIT compilers, as inlining produces more code which must be translated by the JIT without incurring any speed benefits.

## 7 Related Work

Related work falls into five different categories:

*Bytecode optimizers:* There are only 2 Java tools that we are aware of which perform significant optimizations on bytecode and produce new class files: Cream[6] and Jax[28]. 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 compressed. They also are interested in speed optimizations, but at this time their current published speed up results are more limited than those presented in this paper. It would be interesting, in the future, to compare the results of the three systems on the same set of benchmarks.

*Bytecode annotators:* Tools of this category analyze bytecode and produce new class files with annotations which convey information to the virtual machine on how to execute the bytecode faster. We are aware of one such system[4] which passes register allocation information to a JIT compiler in this manner. They obtain speed-ups between 17% to 41% on a set of four scientific benchmarks.

*Bytecode manipulation tools*: There are a number of Java tools which provide frameworks for manipulating bytecode: JTrek[16], Joie[7], Bit[17] and JavaClass[15]. 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[28], DashO-Pro[9] and SourceGuard[25]. Application 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[21] which produces unoptimized C code and relies on GCC to produce the native code. Slightly more sophisticated, Harissa[20] also produces C code but performs some method devirtualization and inlining first. The most sophisticated systems are Vortex[10] and Marmot[13]. 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. Each of these systems include their customized intermediate representations for dealing with Java bytecode, and produce native code directly. There are also numerous commercial Java native compilers, such as the IBM (R) High Performance Compiler for Java, Tower Technology's TowerJ[29], and SuperCede[27], but they have very little published information. The intention of our work is to provide a publically available infrastructure for bytecode optimization. The optimized bytecode could be used as input to any of these other tools.

## 8 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.

Our contributions in this paper are the intermediate representations, the transformations between the intermediate representations, and a comprehensive set of speed up results of processing class files through Soot with an increasing level of optimization.

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 are actively engaged in further work on Soot on many fronts. Baf-level optimizations are being pursued for eliminating the redundant loads and stores. We are also completing the set of basic traditional optimizations by adding side-effect analyses and optimizations such as loop invariant removal and common sub expression elimination. We have also begun the design of an annotation mechanism.

## 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] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, UK, 1998.

[4] Ana Azevedo, Alex Nicolau, and Joseph Hummel. Java Annotation-Aware Just-In-Time (AJIT) Compilation System. *Proc. of the ACM 1999 Java Grande Conference, San Francisco.*, June 1999.

[5] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 324–341, New York, October 6–10 1996. ACM Press.

[6] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice & Experience*, 9(11):1031–1045, November 1997.

[7] 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.

[8] 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.

[9] DashOPro. . http://www.preemptive.com/products.html.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(4):135–150, March 1993.

[15] JavaClass. . http://www.inf.fu-berlin.de/ dahm/JavaClass/ .

[16] Compaq JTrek. . http://www.digital.com/java/download/jtrek .

[17] 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.

[18] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[19] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[20] 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.

[21] 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.

[22] SableCC. . http://www.sable.mcgill.ca/sablecc/.

[23] Tatiana Shpeisman and Mustafa Tikir. Generating Efficient Stack Code for Java. Technical report, University of Maryland, 1999.

[24] Soot - a Java Optimization Framework. . http://www.sable.mcgill.ca/soot/.

[25] 4thpass SourceGuard. . http://www.4thpass.com/sourceguard/.

[26] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, and Étienne Gagnon. Practical Virtual Method Call Resolution for Java . Sable Technical Report 1999-2, Sable Research Group, McGill University, April 1999.

[27] SuperCede, Inc. SuperCede for Java. . http://www.supercede.com/.

[28] 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.

[29] Tower Technology. Tower J.
. http://www.twr.com/.

[30] 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.