Sable
Research Group
http://www.sable.mcgill.ca

Soot - a Java Bytecode Optimization and Annotation Framework

McGill
University

Overview of Soot framework

Why is Java slow?

Useful Features...

Java is a nice programming language for application development because of the useful features it provides:

* Platform independence: a compiled Java application can run without modifications on any operating system which supports Java.

* Execution safety: Java applications can not corrupt their memory space, making them significantly easier to debug.

* Garbage collection: the Java Virtual Machine automatically manages memory use, so memory leaks never occur.

* Object orientation: provides convenient abstractions for programming.

...But Costly Features

Unfortunately, these features are expensive to support; applications written in Java are usually slower than their counterparts written in C or C++. In order to support these features efficiently, a high level of optimization is required.

Core Features

* The Soot framework consists of a set of Java APIs designed to facilitate the optimization and annotation of Java bytecode.

* Soot provides a choice of 3 intermediate representations of varying levels for Java bytecode: Jimple (medium), Grimp (medium-high) and Baf (low). This allows the programmer to write analyses and optimizations on bytecode at the most appropriate level.

* Jimple and Baf also exist as stand alone textual representations, providing two convenient assembly languages for Java bytecode.

* Soot is a Java framework, and so benefits from the advantages of the Java programming language (mainly platform independence and ease of application development.)

* Soot is freed source, licensed under the GNU Library General Public License.

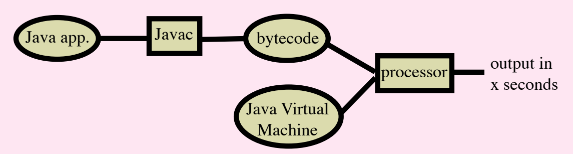
* By targeting Java bytecode, Soot can analyze and optimize the application code for Java programs, as well as for other high level languages which can compile to Java bytecode, such as Eiffel, Scheme, ML and Ada.

* By targeting Java bytecode, the optimizations and annotations performed by Soot can potentially benefit all Java Virtual Machines implementations.

The Java Execution Model

The Java Virtual Machine

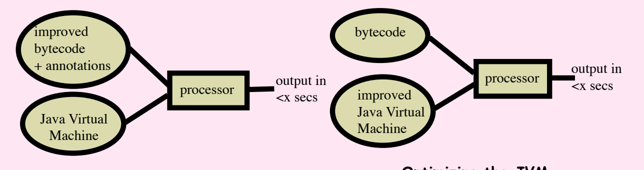
The key to Java's platform independence is the Java Virtual Machine (JVM). Compiling Java applications does not produce binary code which can directly be executed on a processor. Instead, it produces Java bytecodes, which are instructions for this Java Virtual Machine. To execute the program, you must run the Java Virtual Machine, itself a program, which in turn will execute the bytecode.



Optimizing Java

This split architecture gives rise to two levels for optimizing Java execution: optimizing the applications themselves (a series of Java bytecodes), or optimizing the program which executes applications (the Java Virtual Machine, usually written in C.)

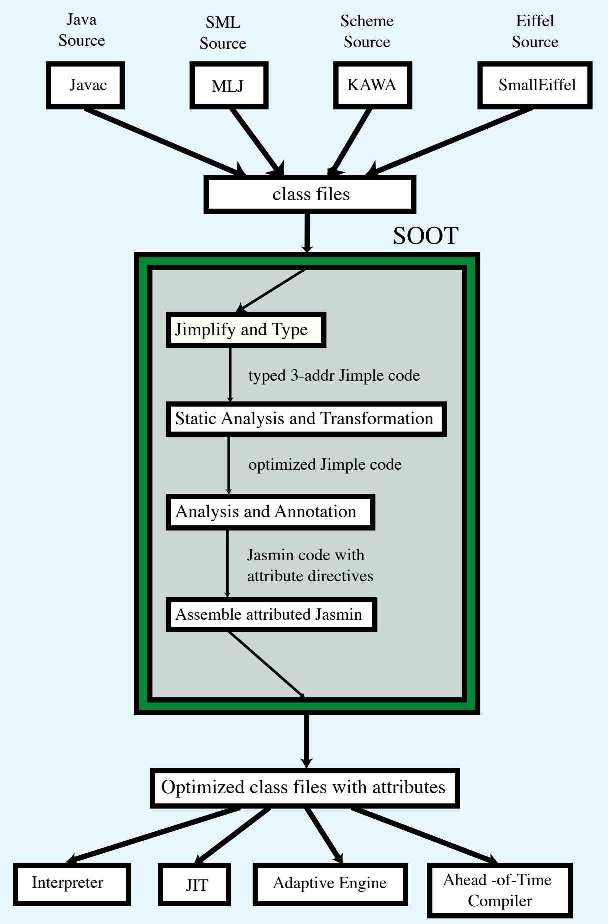
Optimizing the bytecode has the advantage benefiting all Java Virtual Machine implementations, whereas optimizing a specific JVM will usually only improve one given platform (like Linux/x86). Soot focuses on the former approach.



Optimizing Bytecode

Optimizing the JVM

Framework Overview



Soot Optimizations

3 Representations for Java Bytecode : Jimple, Grimp & Baf

Why Bytecode is Cumbersome

Optimizing or annotating bytecode directly is awkward because it is stack based code; computed expressions are not explicit because they are spread over several instructions. Simple analyses and transformations thus become complicated. Furthermore, the bytecode format is laden with encoding issues, such as the constant pool, which make it awkward to manipulate.

```
if(x + y != z)
    return;
else
    System.out.println("foo");
```

Running Example: Original Java Code

Jimple

Jimple is a simple intermediate representation for Java bytecode with the following features:

* Resembles simple Java: instructions are in 3-address code form.
* Unstructured: while's, for's, if-then-else's are broken down into multiple statements. Goto's are allowed.
* Typed: Like Java, Jimple's local variables are typed.

Because of these features, Jimple is well suited for implementing general analyses and optimizations such as copy propagation, devirtualization and method inlining.

```
    t = x + y;
    if t == z goto label0;
    return;

label0:
    ref = System.out;
    ref.println("foo");
```

In Jimple Form

Grimp

The Grimp intermediate representation is similar to Jimple, except that it represents statements as trees. This is extremely useful in situations where Jimple's fractured nature is inappropriate. Grimp is currently being used for decompilation and bytecode generation.

```
    if x + y == z goto label0;
    return;

label0:
    System.out.println("foo");
```

In Grimp Form

Baf

Sometimes it is necessary to deal with bytecode as stack code. The Baf intermediate representation attempts to simplify this task by hiding the encoding issues in Java bytecode, such as the constant pool and the multiple variants of virtually the same instructions. Baf is currently used for peephole optimizations and for a final reordering phase.

```
    iload x
    iadd
    iload z
    icmpge label0
    return

label0:
    getfield System.out
    push "foo"
    invokevirtual println
```

In Baf Form

Improving Java Performance using Soot: via Optimizations

The first approach is to reduce the total bytecode execution cost by directly reducing the number of bytecode instructions executed, or replacing expensive bytecodes with inexpensive ones. The techniques for optimizing standard languages like C and C++ can be used here. The main difference, however, is that Java bytecode instructions have a wide range of costs. The instructions which usually dominate execution time are unaffected by simple optimizations such as common sub-expression elimination or copy propagation. Thus higher level optimizations, such as virtual method call resolution and method inlining, must be performed to yield a significant speed-up.

```
x = 1;
y = x * z;
use(x);
```

$\xrightarrow{\text{optimized becomes}}$

```
y = z;
use(1);
```

Optimization Possibilities

Traditional Optimizations

1) copy propagation*
2) constant propagation and folding*
3) conditional and unconditional branch folding*
4) dead assignment elimination*
5) unreachable code elimination*

Complex Optimizations (interprocedural/side-effect analysis)

1) common sub-expression elimination*
2) loop invariant removal
3) class file splitting
4) object inlining
5) method devirtualization and inlining*

* we have implemented these optimizations in Soot.

Experimental Results: via Optimizations

The table gives the results of performing inlining using class hierarchy analysis to resolve the call graph. The numbers given are fractional execution times with respect to the original execution time of the benchmarks for a given platform. For the Linux Sun JIT, the 'average' ratio is 0.92 which indicates that the average running time is reduced by 8%.

	Linux			NT	
	Sun Int.	Sun JIT	Borland JIT	Sun JIT	Sun Hot.
compress	1.02	0.78	1.00	1.01	0.99
db	0.99	1.01	1.00	1.00	1.00
jack	1.00	0.98	0.99	-	0.97
javac	0.97	0.96	0.97	1.11	0.93
jess	0.93	0.93	1.01	0.99	1.00
Jpat-p	0.99	0.99	1.00	1.00	1.00
mpegaudio	1.04	0.96	-	-	0.97
raytrace	0.76	0.62	0.74	0.89	1.01
schroeder-s	0.97	1.00	0.97	1.02	1.06
soot-c	0.94	0.94	0.96	1.03	1.05
average	0.96	0.92	0.96	1.01	1.00
std-dev	0.07	0.12	0.08	0.06	0.04

Bytecode Annotations

Improving Java Performance using Soot: via Annotations

The second approach is to lessen the cost of certain bytecode instructions by communicating short-cuts to the Java Virtual Machine about their execution. For example, JVMs must always perform checks before executing potentially dangerous code. Static analyses at compile time can determine that some checks at unnecessary. This stage performs analyses and annotates particular instructions so that JVMs can take safe short-cuts when executing them.

```
int[] n = new int[2];

n[0] = 5;
n[1] = 8;
```

$\xrightarrow{\text{annotated becomes}}$

```
int[] n = new int[2];

n[0] = 5; // safe access!
n[1] = 8; // safe access!
```

Annotation Possibilities

Annotations for Optimization

1) Array bounds checks*
2) Null pointer checks*
3) Register allocation
4) Stack allocation of objects
5) Runtime static method binding
6) Parallel computations
7) Exception handling as control flow

Annotations for Profiling

1) Hot methods
2) Persistent objects
3) Garbage collection
4) Branch prediction annotation
5) Hot data

* we have implemented these analyses in Soot.

Experimental Results: via Annotations

(Array bounds check and null pointer check elimination)

Experimental results show the running time and improvement of four benchmarks on modified Java environments: the KaffeVM(tm) and the IBM's High Performance Compiler for Java (HPCJ). The 'normal' column represents normal execution time without any check removal; the 'nocheck' gives the running time if the VM performs no checks (not safe); and the 'with attributes' gives the running time if the compiler eliminates checks as indicated in attributes (safe).

KaffeVM runtime results

	normal	nocheck	with attributes
mpegaudio	80.83s	62.83s(22.3%)	72.57s(10.2%)
FFT	51.44s	48.84s(5.1%)	50.01s(2.8%)
LU	81.10s	81.88s(-0.9%)	78.15s(3.6%)
SOR	46.46s	41.23s(11.3%)	43.19s(7.0%)

HPCJ* runtime results: without optimizations

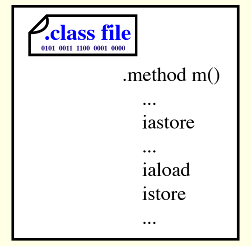
	normal	nocheck	with attributes
mpegaudio	58.88s	29.96s(41.1%)	39.14s(23.1%)
FFT	28.22s	25.09s(1 1.1%)	26.59s(5.8%)
LU	39.99s	28.83s(27.9%)	32.33s(19.2%)
SOR	24.16s	15.46s(36.0%)	15.55s(35.6%)

HPCJ runtime results: with optimizations

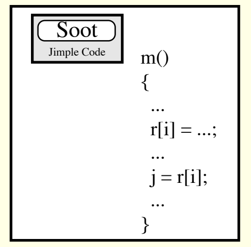
	normal	nocheck	with attributes
mpegaudio	21.27s	15.93s(25.1%)	20.33s(4.4%)
FFT	17.39s	15.34s(1 1.8%)	19.45s(-11.8%)
LU	21.50s	14.84s(30.8%)	21.27s(1.1%)
SOR	11.93s	8.88s(25.6%)	8.88s(25.6%)

Annotations Details

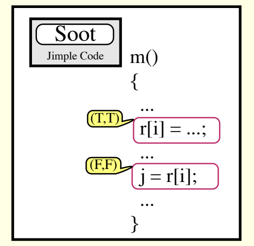
Soot has several steps to add attributes into the class file. The following figures show the process of annotating the class file with array bounds check attributes. Soot takes bytecode from a class file and converts it to typed Jimple code, which is the representation used by the bounds check analysis. The analysis results are represented as tag objects which are attached to the instruction units that have array references. Tags from one method body are aggregated as a list of (unit, tag) pairs, and the list is attached to the method. When the attribute list is written into the class file with the method, each (unit, tag) pair is translated to a (PC, value) entry respectively.



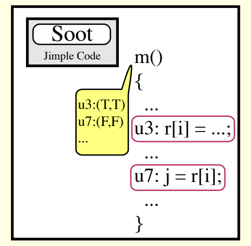
(a) The original class file



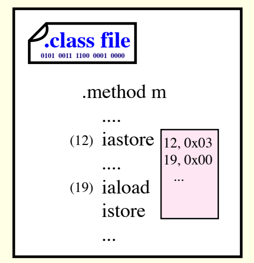
(b) Typed Jimple code



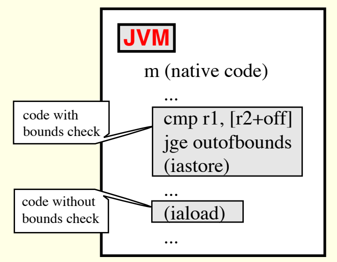
(c) Jimple code with tags attached to instruction units



(d) Soot method with aggregated (unit, tag) pairs



(e) Annotated class file with attribute table of (PC, value) entries.



(f) Attribute-aware VM produces efficient native code.

Credits

Soot is being designed and implemented by
the Sable Research Group, namely:
Raja Vallée-Rai, Patrick Lam,
Patrice Pominville, Feng Qian,
Etienne Gagnon and Laurie Hendren

The HPCJ experiment was done by:
Clark Verbrugge
IBM Toronto Lab.

This work is funded in part by
IBM's Centre for Advanced Studies, FCAR and NSERC

The Sable Research Group belongs to
the McGill University School of Computer Science.

Visit our web site at www.sable.mcgill.ca