Soot
A Java Bytecode Optimization Framework

Sable Research Group
School of Computer Science
McGill University
Goal

• Provide a Java framework for optimizing and annotating bytecode
• provide a set of API’s easy to use and efficient enough for developing competitive optimizers
Outline

- Contribution
- Framework overview
- Intermediate representations
- Transformations
- Optimizations
- Conclusion
Currently used methods for improving Java performance

- JIT compilers
- Way-Ahead-Of-Time Java compilers
- Optimizing bytecode directly
  - Must address expensive bytecode operations: virtual method call, interface call, object allocation
- Annotating the bytecode
  - Statically checking the safety of memory accesses and annotating the bytecode (e.g., array bounds)
Contributions

• 3 intermediate representations used for bytecode optimizations and de-compilations
• Support for both intra-procedural and whole program optimizations
• Able to add future support for bytecode annotation
Framework overview

- Baf - streamlined representation of bytecode
- Jimple - typed 3-address code suitable for optimizations
- Grimp - aggregated version of Jimple suitable for decompilation and bytecode codification

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

• Baf
  – motivation
    • easier to manipulate than the bytecode (abstracts away the constant pool)
    • some bytecode instructions are untyped (dup, swap): difficult to estimate their effect and therefore optimize
  – description
    • stack-based
    • fully typed instructions
    • untyped variables
Intermediate representations (cont.)

• Jimple
  – motivation
    • stack code optimization is difficult
    • 2 types of variables : locals and stack locations
    • untyped nature of stack
  – description
    • 3 address code
    • stack replaced by local variables
    • untyped instructions
    • typed local variables
    • ideal for optimizations
Intermediate representations (cont.)

• Grimp
  – motivation
    • IR difficult to read
    • 3 address code difficult to deal in some cases (eg. Generating good stack code)
  – description
    • compacted version of Jimple: flattened expressions, new and invokespecial compacted to new
    • looks like a partially decompiled Java code
public int stepPoly(int n) {
    if (n > 0) {
        System.out.println("foo");
        return -1;
    } else if (n == 5) {
        return n * n;
    } else {
        return n * 5 + 16;
    }
}

Figure 2: stepPoly in its original Java form.

public int "stepPoly"(int) {
    word e0, i0
    e0 := @this
    i0 := $parameters0
    load i0
    if goto label0
    statictype java.lang.System.out
    push "foo"
    virtualinvoke println
    push -1
    return i0
}

label0:
    load i0
    push 5
    ifgoto i0 label1
    load i0
    load i0
    mul i0
    return i0

label1:
    load i0
    push 5
    mul i0
    push 16
    add i0
    return i0

Figure 3: stepPoly in Baf form.

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

public int stepPoly(int) {
    Test e0;
    int i0;
    e0 := @this;
    i0 := $parameters0;
    if i0 := 5 goto label0;
    java.lang.System.out.println("foo");
    return -1;
}

label0:
    if i0 := 5 goto label1;
    return i0;

label1:
    return i0 * 5 + 16;

Figure 5: stepPoly in Grasp form.
Transformations

• **Bytecode -> Baf**
  – stack simulation : types of untyped instructions
  – distributing the constant pool

• **Baf -> Jimple**
  – produce naïve 3 address code
  – type the local variables (paper)
  – clean up the code (simply collapsing def-use pairs)
Transformations (cont.)

- **Jimple -> Grimp**
  - aggregate expressions
  - fold constructors
  - aggregate expressions

- **Grimp -> Baf**
  - expression trees converted to stack based code

- **Baf -> Bytecode**
  - Pack local variable for placing onto Frame
  - Optimize load/stores (eliminate redundancies)
  - Compute maximum stack height (required by JVM)
  - Produce the bytecode
Optimizations

• Scalar optimizations (implemented)
  – constant propagation and folding
  – conditional and unconditional branch elimination
  – copy propagation
  – dead assignment and unreachable code elimination
  – expression aggregation

• Scalar optimizations (future)
  – common sub-expression elimination
  – loop invariant removal
Optimizations (cont.)

- Whole program optimizations (OOP)
  - call graph based
  - methods for constructing the call graph
    - class hierarchy analysis
    - rapid type analysis
    - variable type analysis (*)
  - methods inlining
Experimental results

<table>
<thead>
<tr>
<th></th>
<th>Jimple Starts</th>
<th>Base Execution</th>
<th>Speed up: →</th>
<th>Speed up: -O</th>
<th>Speed up: -W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Int.</td>
<td>JIT</td>
<td>Int.</td>
</tr>
<tr>
<td>.201_cassidy</td>
<td>3562</td>
<td>441s</td>
<td>67s</td>
<td>6.6</td>
<td>0.86</td>
</tr>
<tr>
<td>.202_jess</td>
<td>13697</td>
<td>109s</td>
<td>48s</td>
<td>2.3</td>
<td>0.97</td>
</tr>
<tr>
<td>.205_interpreter</td>
<td>6302</td>
<td>125s</td>
<td>54s</td>
<td>2.3</td>
<td>0.99</td>
</tr>
<tr>
<td>.209_db</td>
<td>3639</td>
<td>229s</td>
<td>130s</td>
<td>1.8</td>
<td>0.98</td>
</tr>
<tr>
<td>.213_javac</td>
<td>26656</td>
<td>135s</td>
<td>68s</td>
<td>2.0</td>
<td>0.99</td>
</tr>
<tr>
<td>.222_sageaudio</td>
<td>15244</td>
<td>374s</td>
<td>54s</td>
<td>6.9</td>
<td>0.94</td>
</tr>
<tr>
<td>.227_javac</td>
<td>6307</td>
<td>129s</td>
<td>57s</td>
<td>2.3</td>
<td>0.99</td>
</tr>
<tr>
<td>.228_jack</td>
<td>13234</td>
<td>141s</td>
<td>61s</td>
<td>2.4</td>
<td>0.99</td>
</tr>
<tr>
<td>sublee-j</td>
<td>25344</td>
<td>45s</td>
<td>30s</td>
<td>1.5</td>
<td>0.98</td>
</tr>
<tr>
<td>sublee-w</td>
<td>25344</td>
<td>70s</td>
<td>38s</td>
<td>1.8</td>
<td>1.00</td>
</tr>
<tr>
<td>sooij-c</td>
<td>39938</td>
<td>85s</td>
<td>49s</td>
<td>1.7</td>
<td>0.98</td>
</tr>
<tr>
<td>sooij-j</td>
<td>39938</td>
<td>184s</td>
<td>126s</td>
<td>1.5</td>
<td>0.98</td>
</tr>
</tbody>
</table>

Figure 6: Benchmark characteristics and speed-up results. →, -O and -W represent no optimizations, intra-procedural optimizations, and whole program optimizations, respectively. The programs were executed on a 400MHz dual Pentium II machine running GNU/Linux with Linux JDK 1.2, pre-release version 1.
Conclusions

• Soot: framework for optimizing bytecode
• 3 IRs
• transformations between IRs
• useful for optimizing and decompilation
• Speedup for both interpreter and JIT
• Present work
  – Eliminating redundant loads/stores from baff
  – Adding new optimizations (loop invariant removal, common subexpression elimination)
Intra-procedural Inference of Static Types for Java Bytecode

Sable Technical Report No. 1999-1
McGill University
Sable Research Group
Outline

• Introduction
• Challenges of Types
• Typing algorithm
• Extending the algorithm for arrays
• Program transformations
• Experimental results
• Conclusions
Introduction

• **Bytecode**
  - target IR for a variety of compilers (Ada, ML, Scheme, Eiffel)
  - “well-behaved” (verifiable) - checked by bytecode verifier (eg. each method invocation has the correct number of arguments, arguments are well-typed)
  - bytecode verification
    - static : flow analysis for local type estimation (well typed instructions) and not global
    - dynamic (eg. Array bounds checks)
  - local variables of each method kept on the frame of the method and accessed by index (are not typed)
Introduction (cont.)

• Drawbacks of bytecode
  – stack-based
    • complicates the analysis
    • doesn’t map nicely on existing architectures
    • not easy readable
  – local variables not typed (could be used for both analysis and decompilation)

• Addressing the drawbacks
  – IR representations (Jimple)
  – this paper: typing Jimple
Figure 1: Simple example of static typing
Introduction (cont.)

• Problem definition
  – Given: an untyped Jimple method
  – Find: static types of local variables

• Modeled as a graph problem
  – hard nodes: types in the declared hierarchy
  – soft nodes: type variables (to be determined)
  – directed edges: constraints between 2 nodes
    • constraint: denoted $a<-b$ if $b$ is assignable to $a$
Challenges of Types

- Declared types versus types at program points
  - verifier checks the type at each local point (are the operands of the instruction right?)
  - at control flow merge verifier takes LCAC (least common ancestor class of the types of the branch)
  - program from fig.2 will verify, but there is no static solution (a solution where copies are introduced will be presented later)
void m() {
  <unknown> a;
  if (...) {
    a = new A();
    s1: a.f(); // invokevirtual A.f()
  }
  else {
    a = new B();
    s2: a.g(); // invokevirtual B.g()
  }
  s3: a.toString();
  // invokevirtual Object.toString()
}
(a) untyped method

class Object {
  public String toString() { .... }
  ...
}

class A extends Object {
  public void f() { .... }
  ...
}

class B extends Object {
  public void g() { .... }
  ...
}
(b) hierarchy

Figure 2: Different types needed at different program points
Challenges of Types (cont.)

- Type problems due to interfaces (multiple inheritance)
  - LCAC strategy to resolve types from different branches breaks for multiple inheritance
  - Java verifier checks at run-time
  - Hierarchy I - statically typeable (but can be expensive in the presence of many ancestors)
  - Hierarchy II - not statically typeable (extra-copies can solve the problem)
```java
class CC implements IC {
    void f() {}
    void g() {}
}

class CD implements ID {
    void f() {}
    void g() {}
}

class Hard {
    IC getCC() { return new CC(); }
    ID getCD() { return new CD(); }

test() {
    <untyped> a;
    if (...) a1: a = getCC();
    else a2: a = getCD();
    a.f(); // invoke interface IA.f
    a.g(); // invoke interface IB.g
}
```

(a) Untyped program

(b) Hierarchy I and II

Figure 3: Typing interfaces
Typing algorithm

• Algorithm overview
  – abstract problem into a constraint system (directed graph problem)
  – restrict the problem to programs w/o arrays
  – apply simplifying transformations on the graph
  – if no solution found so far perform an exponential search (algorithm is shown to be NP-hard)
Typing algorithm (cont.)

• Building the constraint system
  – a<-b , where a,b - nodes and b is assignable to a
    • simple assignment a=b => T(a)<-T(b)
    • binary expression assignment a=b+3 => T(a)<-T(b), T(a)<-int and T(b)<-int
    • method invocation a=b.equals© => T(a)<-int, java.lang.Object<-T(b) and java.lang.Object<-T(c)
Typing algorithm (cont.)

- Transformations
  - connected components
  - merging primitive types
  - transitive constraints
Typing algorithm (cont.)

– merging single constraints
  • *single parent constraint* \( x \) if \( y < -x \) and \( x \) is not parent of anybody else
  • *single child constraint* \( y \) if \( x < -y \) and \( x \) is not child of anybody else
  • transformations
    – merge all single child constrains
    – merge all soft parent constraints
    – merge with LCA
    – merge all remaining parent constraints
– if no solution found perform an exhaustive search
Extending the algorithm to arrays

• Definitions
  – *array constraint* $a \rightarrow b$ means $a$ is an array whose type is $b$
  – *array depth*: number of indirections necessary to get a non-array type (e.g. $A[][]$ has depth 2)
Extending the algorithm to arrays (cont.)

- **Algorithm**
  - starting from hard nodes
    - follow parent constraints: modify the parent depths s.t. they are <= than child’s depth
    - follow array constraints: assign to element type array depth -1
  - propagate array constraints on arrays
    - propagate a constraint between 2 nodes at equal depth to a constraint between their depth 0 element types
    - change a constraint between 2 nodes of different depth to a constraint between the depth 0 element type of lowest depth node and java.lang.Clonoable
  - find a solution using the non-array algorithm and only 0-depths nodes
  - propagate the solution back to array depths
Figure 9: Solving Array Constraints
Program transformations

– Performed when there is no static type solution

• Type casts

  s3: ((IA)a).f();
  – makes fig. 3 program typeable
  – but adds run-time overhead

• Copy statements

  – introducing copy statements following new statements to take care of the common case of the creation of instances on 2 branches
  – well known techniques can get rid of extra copy statements (copy propagation)
```java
void m() {
    <unknown> a;
    <unknown> b;
    <unknown> c;
    if (...) {
        a = new A();
    s1:    a.f(); // invokevirtual A.f()
          c = a; // Extra copy
    }
    else {
        b = new B();
    s2:    b.g(); // invokevirtual B.g()
          c = b; // Extra copy
    }
}
```

```java
class Object {
    public String toString() { .... }
    ...
}

class A extends Object {
    public void f() { .... }
    ...
}

class B extends Object {
    public void g() { .... }
    ...
}

(a) untyped method
(b) hierarchy

Figure 2: Different types needed at different program points
# Experimental results

## Typing Java bytecode

<table>
<thead>
<tr>
<th>Language</th>
<th>Benchmark</th>
<th># methods</th>
<th># transf.</th>
<th>conn. comp.</th>
<th>single cons.</th>
<th>exhaust.</th>
</tr>
</thead>
<tbody>
<tr>
<td>java:</td>
<td>javac</td>
<td>1179</td>
<td>3</td>
<td>383</td>
<td>796</td>
<td>0</td>
</tr>
<tr>
<td>java:</td>
<td>jdk1.1</td>
<td>5060</td>
<td>14</td>
<td>2832</td>
<td>2228</td>
<td>0</td>
</tr>
<tr>
<td>ada:</td>
<td>kalman</td>
<td>736</td>
<td>10</td>
<td>473</td>
<td>262</td>
<td>0</td>
</tr>
<tr>
<td>eiffel:</td>
<td>compile_to_c</td>
<td>7521</td>
<td>0</td>
<td>1558</td>
<td>5950</td>
<td>0</td>
</tr>
<tr>
<td>ml:</td>
<td>lexgen</td>
<td>209</td>
<td>0</td>
<td>140</td>
<td>69</td>
<td>0</td>
</tr>
<tr>
<td>scheme:</td>
<td>boyer</td>
<td>2255</td>
<td>2</td>
<td>820</td>
<td>1433</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Required steps
Experimental results

Improving Class Hierarchy analysis
–receiver type more accurately determined

<table>
<thead>
<tr>
<th>source language</th>
<th>program name</th>
<th>call-graph edges untyped Jimple (#)</th>
<th>call-graph edges typed Jimple (#)</th>
<th>Reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>java:</td>
<td>jack</td>
<td>10580</td>
<td>10228</td>
<td>3</td>
</tr>
<tr>
<td>java:</td>
<td>javac</td>
<td>26320</td>
<td>23626</td>
<td>10</td>
</tr>
<tr>
<td>java:</td>
<td>jimple</td>
<td>51350</td>
<td>33464</td>
<td>36</td>
</tr>
<tr>
<td>ada:</td>
<td>rudstone</td>
<td>8161</td>
<td>7806</td>
<td>4</td>
</tr>
<tr>
<td>eiffel:</td>
<td>illness</td>
<td>3966</td>
<td>3778</td>
<td>5</td>
</tr>
<tr>
<td>ml:</td>
<td>nucleic</td>
<td>5009</td>
<td>4820</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: Call Graph reduction
Conclusion

• Static type inference algorithm for typing Java bytecode
• emphasized the difference between well-behaved and well-typed bytecode
• experimental results show how the algorithm improves the results of further analysis