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

Raja Vallée-Rai Etienne Gagnon
Laurie Hendren Patrick Lam
Patrice Pominville Vijay Sundaresan

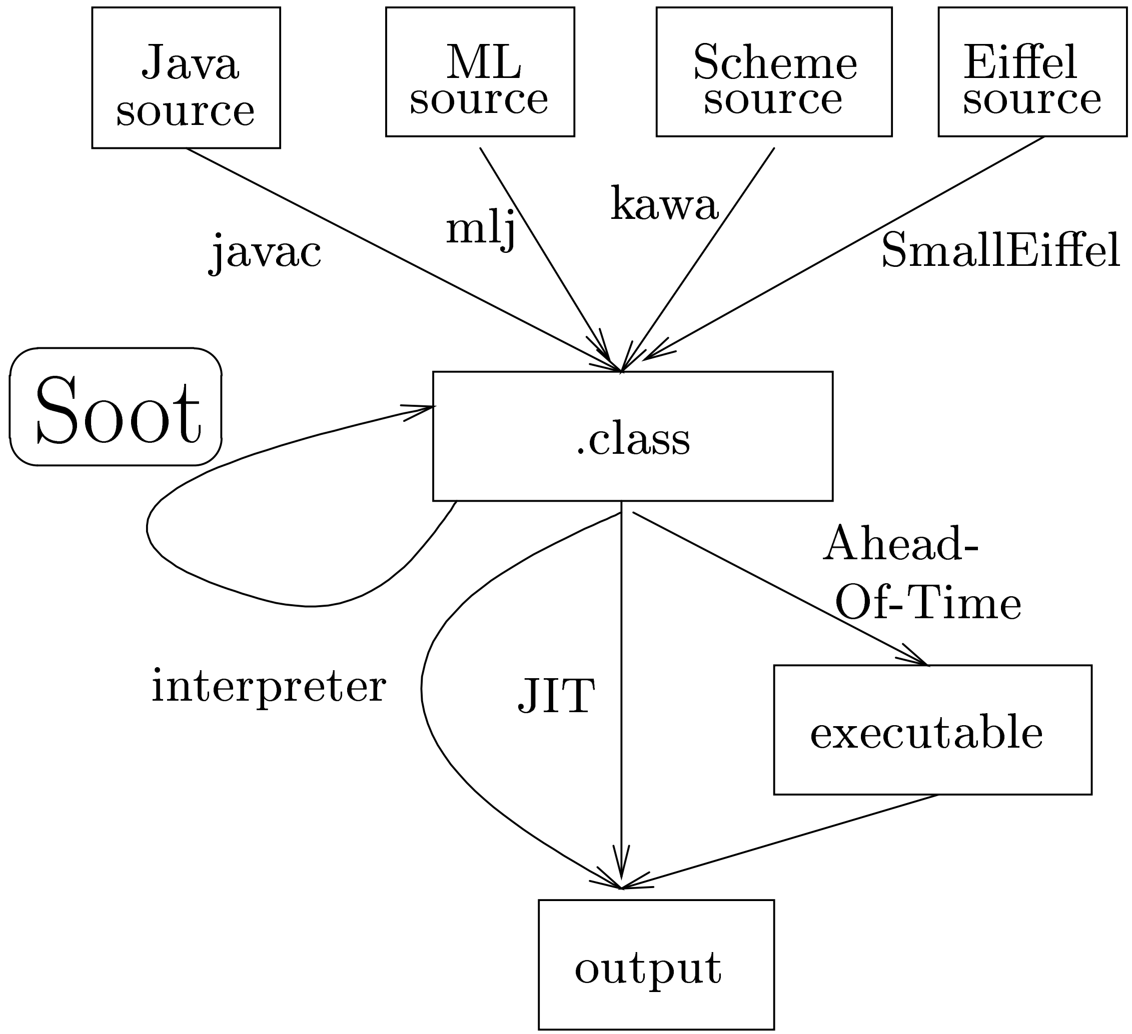
Sable Research Group
School of Computer Science
McGill University
www.sable.mcgill.ca

March 30, 2000

Outline

- Introduction
- Bytecode → 3-address code
- 3-address code optimization
- 3-address code → bytecode via:
  - Aggregating expressions
  - Optimizing stack code
- Experimental Results
- Conclusion

The Java execution model



Emphasize why we do optimization of class files.
In particular, we can point out the experimental questions here.

Java .class files

A Java `class` file contains *fields*, *methods* and *attributes*.

Fields: instance variables or class variables.

Methods: contain Java bytecode

```
// Java source
int cc (int x, int y)
{
  int z;
  z = x * y;
  return z;
}
```

```
// Bytecode
Method int cc (int, int)
   0 iload 1
   1 iload 2
   2 imul
   3 istore 3
   4 iload 3
   5 ireturn
```

Classes, methods and fields also have *attributes*.

Intermediate Representations

What are possible IRs for analysis & transformation?

raw bytecode

- each inst has implicit effect on stack
- no types for local variables
- $>$ 200 kinds of insts

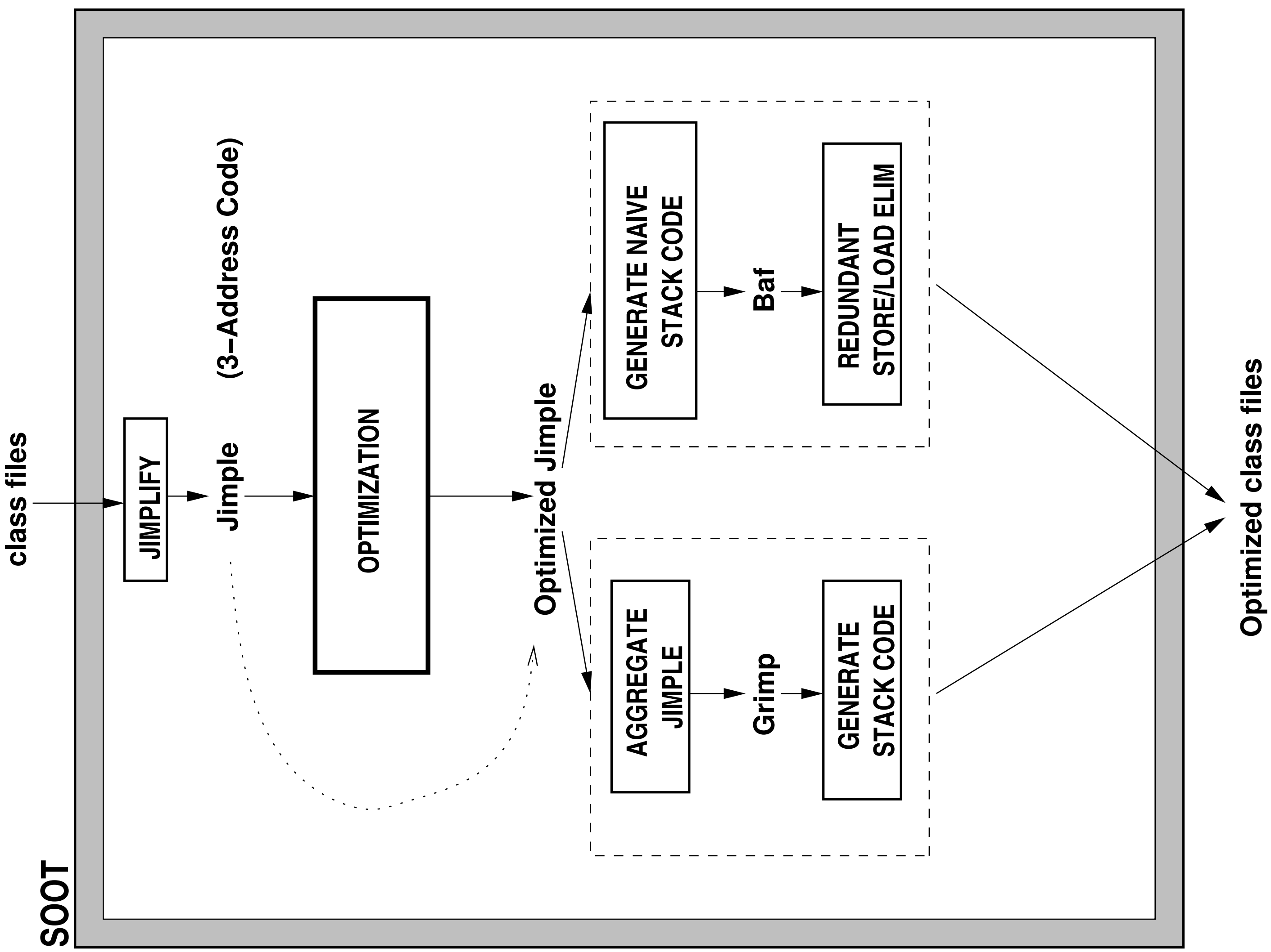
typed 3-address code (Jimple)

- each stmt acts explicitly on named variables
- types for each local variable
- only 15 kinds of stmts

Furthermore, three-address code is familiar territory.

We do analyses on our Jimple IR.

Inside Soot


SOOT
class files
JIMPLIFY
Jimple
(3–Address Code)
OPTIMIZATION
Optimized Jimple
GENERATE NAIVE STACK CODE
Baf
REDUNDANT STORE/LOAD ELIM
AGGREGATE JIMPLE
Grimp
GENERATE STACK CODE
Optimized class files

Bytecode to Jimple I – Naive Jimple

We first produce *naive Jimple*.

```
// Original Java source
int x, y, z;
z = x * y;
return z;
```

From bytecode, we produce naive Jimple:

`.numlocals 3`	`unknown u0, u1, u2;`
`.maxdepth 2`	`unknown $s0, $s1;`
`iload 0`	`$s0 = u0;`
`iload 1`	`$s1 = u1;`
`imul`	`$s0 = $s0 * $s1;`
`istore 2`	`u2 = $s0;`

This still isn't very easy to analyse.

Bytecode to Jimple II – Typing

Want to assign types to locals; but stack variables change types:
$\Rightarrow$ Must split by def-use webs.

After splitting, general typing is still NP-complete (see SAS2000 paper); but transformations and heuristics work for all verifiable bytecode:

```
static java.awt.Component m (boolean param0)
{ unknown_type v0;    // int
  unknown_type v1;    // java.awt.Button

  v0 := param0;
  v1 = new java.awt.Button;
  specialinvoke v1.<init>();
  ...
```

Bytecode to Jimple III – Cleanup

Still have extremely verbose code.

To fix this, use:

- simple aggregation (def/use pair collapsing) within Jimple

```
$i0 = l2;
$i1 = l3;                 ⇒   l1 = l2 * l3;
$i0 = $i0 * $i1;
l1 = $i0;
```

- copy propagation; elimination of any unused variables

This gives clean Jimple code, ready for optimization.

Advantages of Jimple and Soot

Jimple has many advantages:

- Local variables are typed.
- Simple expressions (1 operator / statement).

Furthermore, by being part of the Soot framework, Jimple also has these features:

- Uses and defs are easily available.
- Jimple provides a flow analysis framework.
- Call graph and hierarchy information available.

Applications of Jimple

What we have done with Jimple:

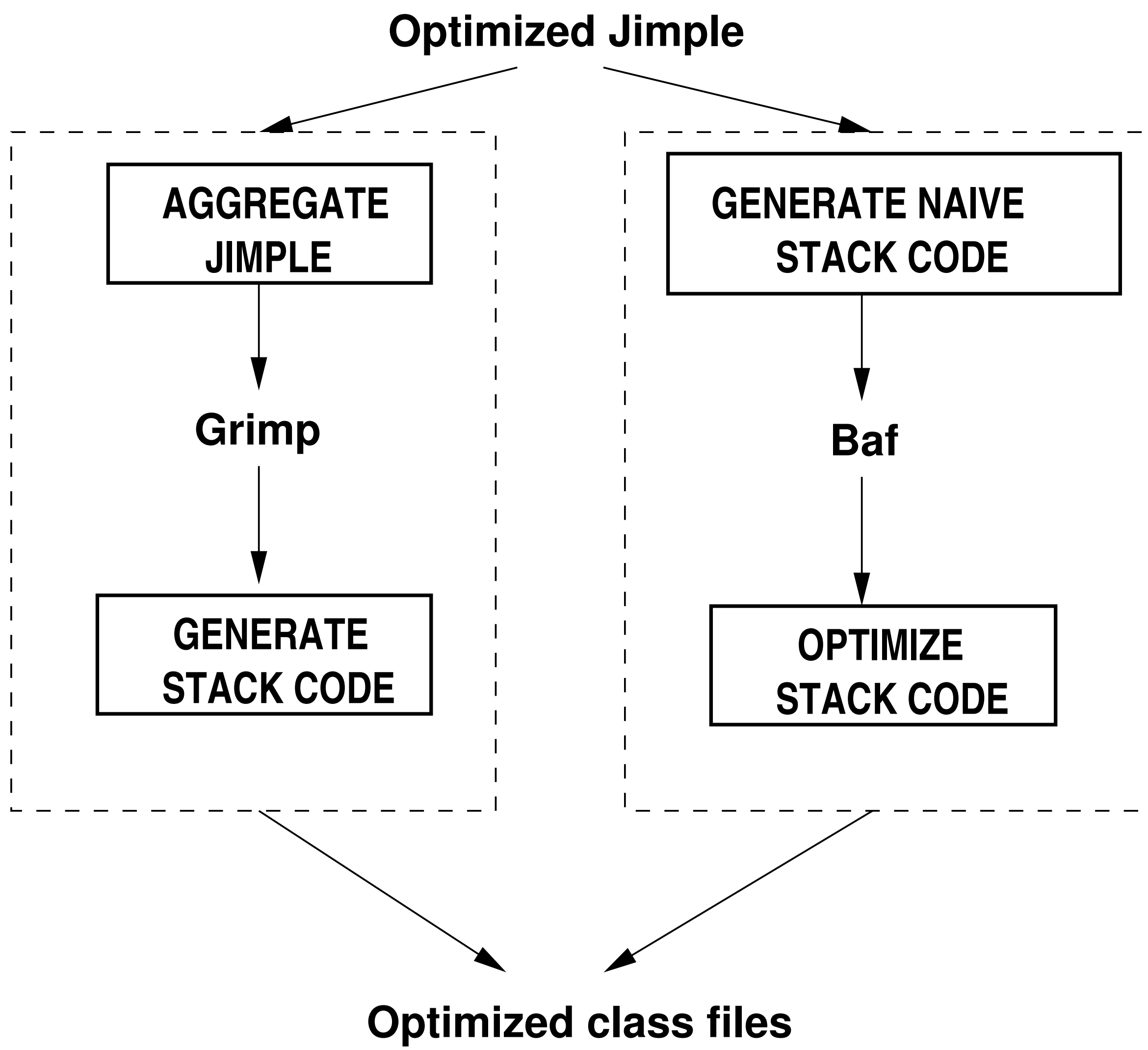
- virtual method resolution
- inlining
- static method binding

What we are doing with Jimple:

- side effect analysis
- partial redundancy elimination
- object inlining
- synchronization elimination
- decompiling Java bytecode
- stack allocating objects (*)
- array bounds checks elimination (*)

(*) - require special attributes and VM support thereof.

Jimple to Bytecode Picture



Jimple to Bytecode

Jimple must be converted back to bytecode.

An efficient translation is vital, and not obvious!

We have investigated two ways of going from Jimple back to bytecode:

Grimp imitate `javac` code generation

Baf create naive bytecode and optimize it

Jimple to Bytecode

Goal: produce tight bytecode.

```
// Java code
x = (y * z) / 2
```

⇒

```
// Jimple code
$i1 = y * z;
$i2 = $i1 / 2;
x = $i2;
```

⇒

```
// Bytecode
iload y
iload z
imul
istore $i1
iload $i1
iconst 2
idiv
istore $i2
iload $i2
istore x
```

⇓

```
// Good bytecode
iload y
iload z
imul
iconst 2
idiv
istore x
```

⇒ Avoid using stack locals.

Motivation: Grimp

Goal: produce tight bytecode.

`javac` can generate tight stack code. Why?

- `javac` has the expression trees.

Grimp is an IR allowing expression trees.

- It is essentially Jimple with arbitrary expressions.

Idea: Recreate expression trees; emit good stack code from them.

Handy bonus for Grimp: produces human-readable code!

Producing Grimp

Basic idea:

```
d: $i1 = y * z;
   /* Jimple stmts */   ?=>   $i2 = (y * z) / 2;
u: $i2 = $i1 / 2;
```

Consider pairs *(def, use)*; *def* has only use *use*, and *use* has unique def *def*. If safe, we move RHS of *def* into *use*.

This is *aggregation*.

Safety:

- no crossing exception boundaries
- no meddling field writes / method calls

Proceeds in reverse pseudo-topological order.

Grimp peepholes

Consider the java statement:

```
a[i++] = new Object();
```

Expressions should not have side effects, so Grimp only allows defs on the LHS of an assignment statement:

```
$i1    = i;
i      = $i1 + 1;
a[$i1] = new Object();
```

In stack code, `$i1` is unneeded.

Result: `compress` benchmark is measurably slower after Grimp.

Grimp peepholes II

Fix: during backend bytecode generation, we look for some patterns:

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

If we match, then emit only $\texttt{s}_3$, but increment `<lvalue>` while using $\texttt{s}_3$.

This works usually, but not always; we have a limited set of peepholes.

The Baf IR and its Optimizations

Goal: Generate efficient bytecode from the Jimple IR.

Observation: Intermediate results need not be stored in locals.

We can keep them on the stack.

Baf Solution: The stack code can be improved after generation.

How?

- Introduce a low-level, stack-based Intermediate Representation.

  Some simple transformations can be done on it.

Baf IR note: we put back slide 13

Problem 1: Redundant store/load pairs

```
// Example Java code
Integer fetchI() {
        Integer x = i;
        return x;
    }
```

```
// Naive Baf
java.lang.Integer fetchI()
{
  word r0;
  staticget Test.i;
  store.r r0;
  load.r r0;
  return.r;
}
```

```
// Improved Baf
  java.lang.Integer fetchI()
  {
    staticget Test.i;
    return.r;
  }
```

Goal: Eliminate redundant store/load pairs.

Observations

- Often produced when translating Jimple statements into naive bytecode.
- Need to check that there are no more uses of that local.

Problem 2: Redundant store/load/load triples

```
// Example Java
   Test t = new Test();
```

```
// Naive Baf
word $r0, r1;
new Test;
store.r $r0;
load.r $r0;
specialinvoke Test.<init>();
load.r $r0;
store.r r1;
return;
```

```
// Improved Baf
word r1;
new Test;
dup1.r;
specialinvoke Test.<init>();
store.r r1;
return;
```

Goal: Eliminate redundant store/load/load triples.

Observations

- No dups produced from Jimple $\Rightarrow$ many store/load/load triples.
- Arbitrary bytecode sequences can separate each instruction of a store/load/load triple.
- Reordering stack code or blocks of stack code might be essential.

Interleaving sequences

We are faced with the following stack code:

```
    push 1729;                     // [1729]
    push 0;                        // [0, 1729]
*   store.i i0;                    // [1729]
    new Test;                      // [Test, 1729]
    dup.r;                         // [Test, Test, 1729]
    specialinvoke Test.<init>();   // [Test, 1729]
    store.r r0;                    // [1729]
*   load.i i0;                     // [0, 1729]
```

We can reduce this store/load pair:

- stack height is 2 before both store and load inst.
- stack height never drops below 1 between store, load.

Between the `store` and `load`, we have a *level sequence*.

Baf rearrangement

```
int x = 0, y = 1;
instance.fct(x);
```

```
    word i0, i1, $r0;
    push 0;          // [0]
2   store.i i0;      // []
    push 1;          // [1]
    store.i i1;      // []
    staticget Test.instance; // [i]
1   store.r $r0;     // []
1   load.r $r0;      // [i]
2   load.i i0;       // [0, i]
    virtualinvoke Test.fct(int); // []
```

```
    word i1;
    push 1;
    store.i i1;
    staticget <Test: Test instance>;
    push 0;
    virtualinvoke <Test: void fct(int)>;
```

Notes on Solution

- Patterns might be interleaved with each other.
- Peepholes are not powerful enough.
- Data dependencies must be taken into account.
- More aggressive optimization of bytecode leads to correct, verifiable code that breaks JITs.

Algorithm Overview

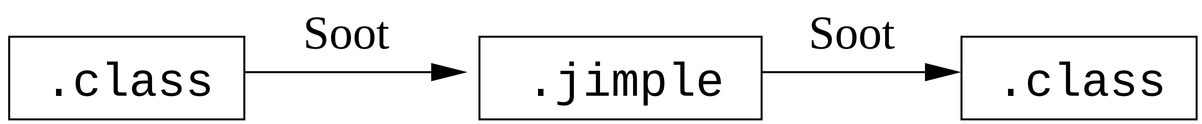
- Perform a fixed point iteration over a method's bytecode, reordering instructions and removing `sl` pairs and `sll` triples separated by level sequences.
- Practically all reducible store/load/load triples and store/load pairs get reduced.

- Algorithm is generic; i.e. not peephole like.
- It can easily be extended to handle other flavours of dups (i.e. dup_x1, dup2_x2, ...).

Experiments

Two questions arise:

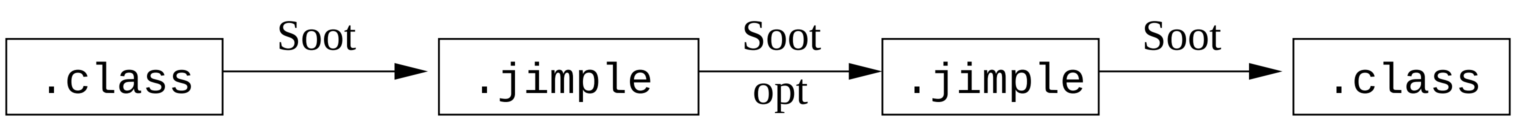
- Does Soot make classfiles slower?



No optimization

GOAL: Same execution speed, robust across platforms.

- Can Soot make classfiles faster?



GOAL: Faster is better.

Experimental Results IA – Straight-Through

Q: Does Soot preserve execution speed?

Ideal result is 1 with low standard deviation.

	Grimp aggregation		Baf stack opts	
	average	std. dev	average	std. dev
Linux Sun Int	1.01	.02	1.00	.02
Linux Sun JIT	1.01	.02	1.00	.01
Linux Bor. JIT	1.01	.02	1.01	.01
NT Sun JIT	1.00	.02	1.00	.01
NT Sun Hotspot	1.00	.02	1.00	.01

Numbers are execution fractions.

Experimental Results IB – Baf vs. Grimp

Numbers are fractions of the original execution speed; lower means faster. * is a failure. ≈ denotes a variation of less than 2 percent.

	Baf stack code optimization					Grimp expression aggregation				
	Linux			NT		Linux			NT	
	Sun Int.	Sun JIT	Bor. JIT	Sun JIT	Sun Hot.	Sun Int.	Sun JIT	Bor. JIT	Sun JIT	Sun Hot.
compress	≈	≈	≈	≈	≈	1.07	1.02	1.04	≈	≈
db	≈	≈	≈	≈	≈	≈	1.05	≈	≈	≈
jack	≈	≈	≈	*	≈	≈	≈	≈	*	≈
javac	≈	.98	≈	≈	.97	≈	1.03	≈	≈	.95
jess	1.02	≈	1.04	≈	≈	≈	1.02	1.04	.97	≈
mpegaudio	1.05	≈	*	*	≈	1.03	≈	*	*	≈
raytrace	≈	1.02	≈	≈	≈	≈	≈	≈	≈	≈
jpat-p	≈	≈	≈	≈	≈	≈	≈	≈	≈	≈
schroeder-s	.97	≈	*	1.03	≈	.98	≈	*	1.03	≈
soot-c	≈	≈	1.02	≈	1.03	≈	≈	≈	≈	≈
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

Experimental Results II – Optimization

Optimizing classfiles can be fruitful; we inlined methods statically using class hierarchy analysis.

	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

Related Work

Bytecode optimizers: Cream and Jax

Bytecode manipulation tools: JTrek, Joie, Bit and JavaClass

Java application packagers: Jax, DashO-Pro, SourceGuard

Java native compilers: Toba, Vortex, Marmot, IBM HPJ, TowerJ, SuperCede

Baf-like Stack code optimization: University of Maryland

Conclusions

- Presented Soot, a Java bytecode analysis framework.
- Described two ways of producing efficient stack code from three-address code.

  Grimp Imitate `javac`'s code generation strategy and avoid emitting redundant instructions.

  Baf Optimize stack code to eliminate redundant instructions.

- Experimentally validated that:
  - Soot doesn't make bytecode slower
  - Soot can make bytecode faster

    (obtained speedup reaching .62 on a benchmark).
- Soot is free software; download it from `www.sable.mcgill.ca`

Grimp bonus

Handy bonus for Grimp: produces human-readable code!

```
int i = 0, j = 0;

while (i < 10)
{
    j = x * (y - j) / 17;
}
return j;
```

```
    i2 = 0; i3 = 0;
    goto label1;

label0:
    $i4 = i1 - i3;
    $i5 = i0 * $i4;
    i3 = $i5 / 17;

label1:
    if i2 < 10 goto label0;

    return i3;
```

Grimp bonus II

Corresponding Grimp:

```
    i2 = 0; i3 = 0;
    goto label1;

 label0:
    i3 = i0 * i1 - i3 / 17;

 label1:
    if i2 < 10 goto label0;

    return i3;
```