

Soot: A Java Bytecode Optimization Framework

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

Ahmad Shamail

March 05, 2025

- Background
- Motivation
- Framework and Methodology
 - Code Intermediate Representations
 - Code type Conversion
- Experiment and Results
- Prior/Related Work
- Q&A

- JVM is an **abstract machine** that runs Java bytecode.
- Acts as a bridge between Java programs and the underlying OS.
- Provides features like:
 - Automatic memory management (Garbage Collection).
 - Security - isolates code from OS.
 - Platform independence.

What is Java Bytecode?

- Intermediate representation of Java code - stack based.
- Stored in .class files after compilation.
- Not human-readable, but JVM understands it.

Example: Java Code vs. Bytecode

Java Code:

```
public class Addition {  
    public static void main(String[] args) {  
        int a = 5;  
        int b = 3;  
        int sum = a + b;  
        System.out.println(sum);  
    }  
}
```

Bytecode:

```
iconst_5 // Push 5 onto stack  
istore_1 // Store in variable  
iconst_3 // Push 3 onto stack  
istore_2 // Store in variable  
iload_1 // Load a (5)  
iload_2 // Load b (3)  
iadd // Perform addition (5 + 3)  
...  
...  
return
```

Motivation

JAVA

Pros:

- Platform independent (due to JVM and bytecode)
- Does garbage collection
- Provides object orientation

Cons:

- Java applications suffer performance overhead due to:
 - **Bytecode Interpretation:** Java code is compiled to bytecode, which is executed on the JVM rather than directly by the hardware.
 - **Expensive Operations:**
 - Virtual method calls and interface calls.
 - Object allocations and exception handling.
 - Array bounds checking.
- Optimizations are needed to make Java competitive with C and C++.

Common Performance Optimizations

- **Just-In-Time (JIT) Compilation:**

- Converts bytecode to native machine code at runtime.
- Caching of compiled code minimizes lag on future execution.

- **Ahead-Of-Time (AOT) Compilation:**

- Compiles bytecode into native code before execution.
- Reduces startup time and enhances performance.

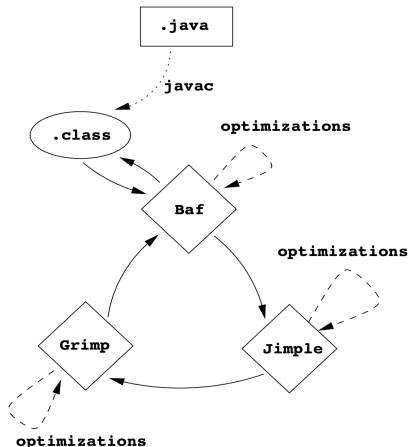
Optimizing Bytecode Directly

- Some bytecode instructions are more expensive than others.
- Simple optimizations (like copy propagation) do not significantly improve performance.
- More advanced techniques are required:
 - **Method Inlining**: Replace method calls with their actual code.
 - **Annotation**: Annotate the bytecode to inform the JVM when (array bounds) checks can be skipped.
- These optimizations reduce execution overhead and improve runtime efficiency.

Research Goal

- Develop tools and techniques for optimizing Java bytecode.
- Enable efficient execution without sacrificing Java's functionality and portability.
- Introduce **Soot** as a framework to aid Java optimization.

- Soot provides three **Intermediate Representations** :
 - **Baf** – A streamlined bytecode representation, easy to manipulate.
 - **Jimple** – A typed 3-address representation, suitable for optimizations.
 - **Grimp** – An aggregated version of Jimple, useful for decompilation.



Motivation for Baf

- Java bytecode is difficult to analyze and modify due to:
 - Stack-based execution model – Each instruction operates on an implicit stack.
 - Constant pool complexity – Managing class references is tedious.
 - Untyped bytecodes – Some instructions require runtime type resolution.
- Baf (Bytecode Abstract Form) is introduced to simplify bytecode manipulation:
 - Preserves stack-based execution but eliminates constant pool complexities (uses direct names).
 - Fully typed representation – Makes type information explicit.

Baf Example

Java Code

```
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;
}
```

Baf Representation

```
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
}
```

Motivation for Jimple

- Stack-based bytecode makes optimization hard:
 - Each operation modifies an implicit stack, making dependencies hard to track.
 - Complex control flow due to 'goto' and exception handling.
- Jimple provides a simpler, three-address representation:
 - Replaces stack operations with explicit local variables.
 - Uses typed expressions for easier analysis.
 - Makes control flow clear, simplifying optimizations.

Jimple Example

Java Code

```
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;
}
```

Jimple Representation

```
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;
}
```

Motivation for Grimp

- Jimple is optimized for analysis, but not readability:
 - Many low-level three-address code instructions.
 - Difficult to read for decompilation.
- Grimp is designed for structured decompilation:
 - Groups expressions into trees, making them more readable.
 - Collapses unnecessary variables into expressions.
 - Reconstructs structured control flow.

Grimp Example

Java Code

```
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;
}
```

Grimp Representation

```
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;
}
```

Summary: Baf, Jimple, and Grimp

- **Baf (Bytecode Abstract Form)**

- Removes the constant pool complexity and uses fully typed instructions.
- Reduces stack complexity, making transformations and optimizations easier.

- **Jimple (Three-Address Code Representation)**

- Converts Java bytecode into typed three-address code.
- Eliminates the stack and uses explicit local variables.
- Simplifies control flow, making it easier for compiler optimizations.

- **Grimp (Aggregated Jimple for Decompilation)**

- Reconstructs structured Java-like expressions from Jimple.
- Groups fragmented Jimple expressions into trees, improving readability.
- Useful for decompilation and Java source reconstruction.

Transformations in Soot

- **Goal:** Convert Java bytecode into simpler forms for easier optimization.
- **Stages of Transformation:**
 - **Bytecode → Baf:** Removes constant pool complexity and preserves stack-based execution.
 - **Baf → Jimple:** Converts to three-address code, eliminating implicit stack operations.
 - **Jimple → Grimp:** Aggregates expressions to resemble structured Java code.
 - **Grimp → Optimized Baf:** Simplifies and optimizes bytecode for efficient execution.
 - **Optimized Baf → Bytecode:** Generates final JVM-compatible bytecode with performance improvements.

- **Intraprocedural Optimizations** (Within a method)
 - **Constant propagation and folding** – Replaces variables with known constant values.
 - **Branch elimination** – Removes redundant conditional checks.
 - **Copy propagation** – Replaces unnecessary variable copies with their original values.
 - **Dead code elimination** – Removes code that has no effect on execution.
 - **Expression aggregation** – Merges multiple small expressions into fewer, larger ones.
- **Whole-Program Optimizations** (Across multiple methods/classes)
 - **Call graph analysis** – Identifies which methods are actually used (static virtual method binding).
 - **Method inlining** – Replaces small method calls with their actual body to reduce function call overhead.

• **Testing Environment:**

- 12 Java applications tested, including SPECjvm98 benchmarks.
- Ran on a dual 400MHz Pentium II machine with GNU/Linux, JDK 1.2.
- Measured execution time using both:
 - Interpreter mode (without JIT).
 - JIT-compiled mode (optimized native execution).

• **Optimization Levels Compared:**

- → (No optimizations, baseline performance).
 - -O (Intraprocedural optimizations applied).
 - -W (Whole-program optimizations applied).
-
- Performance measurements were averaged over five runs.
 - Correctness verified by comparing program outputs before and after optimizations.

- **Whole-program optimizations (-W) provided the most improvement:**
 - Up to 21% speedup in JIT execution.
 - Up to 8% improvement in interpreted mode.
 - Method inlining had the largest impact
- **Intraprocedural optimizations (-O) had minor effects:**
 - Around 2-3% performance improvement.
 - JIT already performs many of these optimizations, limiting additional gains.
- **Some benchmarks showed performance decreases:**
 - Compress benchmark saw a 14% slowdown due to redundant load/store instructions introduced to a critical loop.
 - Some optimizations conflicted with JIT optimizations, resulting in no improvement.

Performance Comparison Table

Performance Results of Soot Optimizations

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

How Soot Improves Over Existing Approaches

Category	Limitations	How Soot is Better
Bytecode Optimizers	Limited speed-ups (1-3%), focus on compression	Uses whole-program optimizations, structured IRs, better performance gains
Bytecode Annotators	Only assists JIT, limited to scientific benchmarks	Directly modifies bytecode, works for all applications
Bytecode Manipulation Tools	No structured IRs, only low-level bytecode edits	Introduces Baf, Jimple, Grimp IRs, enables advanced transformations
Java Application Packagers	Focuses on size reduction, no performance improvements	Optimizes execution, could be extended for compression
Java Native Compilers	Compiles to native code, doesn't optimize bytecode	Optimizes bytecode while keeping JVM compatibility

Thank you! But ... Questions to Think About

- **Why is this relevant to security?**
 - How can bytecode optimizations impact security vulnerabilities?
- **How could attackers exploit bytecode-level transformations?**
 - Could obfuscation or optimizations introduce security risks?
- **What role does Soot play in malware detection?**
 - Can we use Soot to analyze and detect malicious bytecode?

- **Relevance to Security**

- Optimized bytecode could unintentionally remove critical security checks.
- Certain transformations might introduce side-channel vulnerabilities.

- **Attacker Exploits**

- Attackers could use bytecode optimizations to obfuscate malicious code.
- Control flow transformations might bypass static security checks.

- **Soot & Malware Detection**

- Soot can help deobfuscate and simplify malware-infected bytecode.
- Call graph analysis in Soot can detect injected malicious methods.

Thank you!