Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 141 (2005) 109–128

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# Subroutine Inlining and Bytecode Abstraction to Simplify Static and Dynamic Analysis

Cyrille Artho

*Computer Systems Institute, ETH Zürich, Switzerland*

Armin Biere

*Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria*

Abstract

In Java bytecode, intra-method subroutines are employed to represent code in "finally" blocks. The use of such polymorphic subroutines within a method makes bytecode analysis very difficult. Fortunately, such subroutines can be eliminated through recompilation or inlining. Inlining is the obvious choice since it does not require changing compilers or access to the source code. It also allows transformation of legacy bytecode. However, the combination of nested, non-contiguous subroutines with overlapping exception handlers poses a difficult challenge. This paper presents an algorithm that successfully solves all these problems without producing superfluous instructions. Furthermore, inlining can be combined with bytecode simplification, using abstract bytecode. We show how this abstration is extended to the full set of instructions and how it simplifies static and dynamic analysis.

*Keywords:* Java bytecode analysis, inlining, static analysis, dynamic analysis

## 1 Introduction

Java [12] is a popular object-oriented, multi-threaded programming language. Verification of Java programs has become increasingly important. In general, a program written in the Java language is compiled to Java *bytecode,* a machine-readable format which can be executed by a Java Virtual Machine (VM) [16]. Prior to execution, such bytecode must pass a well-formedness test called *bytecode verification,* which should allow a regular Java program to pass but also has to ensure that malicious bytecode, which could circumvent

doi:10.1016/j.entcs.2005.02.034

security measures, cannot be executed. The Java programming language includes methods, which are represented as such in bytecode. However, bytecode also contains *subroutines,* functions *inside* the scope of a method. A special jump-to-subroutine (`jsr`) instruction saves the return address to the stack. A return-from-subroutine (`ret`) instruction returns from a subroutine, taking a register containing the return address as an argument. This artefact was originally designed to save space for bytecode, but it has three unfortunate effects:

(i) It introduces functionality not directly present in the source language.

(ii) The asymmetry of storing the return address on the stack with `jsr` and retrieving it from a register (rather than the stack) greatly complicates code analysis.

(iii) A subroutine may read and write local variables that are visible within the entire method, requiring distinction of different calling contexts.

The second and third effect have been observed by Stärk et al. [21], giving numerous examples that could not be handled by Sun's bytecode verifier for several years. The addition of subroutines makes bytecode verification much more complex, as the verifier has to ensure that no `ret` instruction returns to an incorrect address, which would compromise Java security [16,21]. Therefore subroutine elimination is a step towards simplication of bytecode, which can be used in future JVMs, allowing them to dispense with the challenge of verifying subroutines.

Correct elimination of subroutines can be very difficult, particularly with nested subroutines, as shall be shown in this paper. Furthermore, considering the entire bytecode instruction set makes for very cumbersome analyzers, because it encompasses over 200 instructions, many of which are variants of a base instruction with its main parameter hard-coded for space optimization [16]. Therefore we introduce a register-based version of abstract bytecode which is derived from [21]. By introducing registers, we eliminate the problem of not having explicit instruction arguments, simplifying analysis further.

JNuke is a framework for static and dynamic analysis of Java programs [2,5]. *Dynamic analysis,* including run-time verification [1] and model checking [13], has the key advantage of having precise information available, compared to classical approaches like *theorem proving* [10]. At the core of JNuke is its VM. Its event-based run-time verification API serves as a platform for various run-time algorithms, including detection of high-level data races [4] and stale-value errors [3].

Recently, JNuke has been extended with static analysis [2], which is usually faster than dynamic analysis but less precise, approximating the set of possible program states. "Classical" static analysis uses a graph representation of the program to calculate a fix point [8]. The goal was to re-use the analysis logics for static and dynamic analysis. This was achieved by a graph-free data flow analysis [20] where the structure of static analysis resembles a VM but allows for non-determinism and uses sets of states rather than single states in its abstract interpretation [2].

Bytecode was the chosen input format because it allows for verification of Java programs without requiring their source code. Recently, even compilers for other languages to Java bytecode have been developed, such as `jgnat` for Ada [7] or `kawa` for Scheme [6]. However, bytecode subroutines and its a very large, stack-based instruction set make static and dynamic analysis difficult. JNuke eliminates subroutines and simplifies the bytecode instruction set.

Section 2 gives an overview of Java compilation and treatment of exception handlers. The inlining algorithm is given in Section 3. Section 4 describes conversion to abstract, register-based bytecode. Section 5 describes differences between our work and related projects, and Section 6 concludes.

## 2 Java Compilation with Bytecode Subroutines

### 2.1 Java Bytecode

Java bytecode [16] is an assembler-like language, consisting of instructions that can transfer control to another instruction, access local variables and manipulate a (fixed-height) stack. Each instruction has a unique address or *code index*. Table 1 describes the instructions referred to in this paper. In this table, $r$ refers to a register or local variable, $j$ to a (possibly negative) integer value, and $a$ to an address. The instruction at that address $a$ will be denoted as code($a$), while the reverse of that function, index($ins$) returns the address of an instruction.

The maximal height of the stack is determined at compile time. The type of instruction argument has to be correct. Register indices must lie within statically determined bounds. These conditions are ensured by any well-behaved Java compiler and have to be verified by the class loader of the Java Virtual Machine (VM) during *bytecode verification* [16], the full scope of which is not discussed here.

### 2.2 Exception Handlers and Finally Blocks

The Java language contains *exceptions,* constructs typically used to signal error conditions. An exception supercedes normal control flow, creates a new exception object $e$ on the stack and transfers control to an *exception handler.*

| Instruction | Description |
|---|---|
| `aload` $r$ | Pushes a reference or an address from register $r$ onto the stack. |
| `iload` $r$ | Pushes an integer from register $r$ onto the stack. |
| `astore` $r$ | Removes the top stack element, a reference or address, storing it in register $r$. |
| `istore` $r$ | Removes the top stack element, an integer, storing it in register $r$. |
| `goto` $a$ | Transfers control to the instruction at $a$. |
| `iinc` $r$ $j$ | Increments register $r$ by $j$. |
| `ifne` $a$ | Removes integer $j$ from the stack; if $j$ is not 0, transfers control to $a$. |
| `jsr` $a$ | Pushes the successor of the current addr. onto the stack and transfers control to $a$. |
| `ret` $r$ | Loads an address $a$ from register $r$ and transfers control to $a$. |
| `athrow` | Removes reference $r$ from the stack, "throwing" it as an exception to the caller. |
| `return` | Returns from the current method, discarding the stack and all local variables. |

Table 1
A subset of Java bytecode instructions.

The range within which an exception can be "caught" is specified by a `try` block. If such an exception $e$ occurs at run-time, execution will continue at the corresponding `catch` block, if present, which deals with the exceptional program behavior. An optional `finally` block is executed whether an exception occurs or not, but always after execution of the `try` and `catch` blocks. Therefore, the presence of a `finally` block creates a dualistic scenario: in one case, an exception occurs, which requires both the `catch` and `finally` blocks to be executed. In the absence of an exception, or if an exception occurs that is not within the type specification of the `catch` block, only the `finally` block has to be executed. Because of this, a default exception handler is required to catch all exceptions that are not caught manually.

In the following text, lower case letters denote single values. Monospaced capital letters such as `C` will denote control transfer targets (statically known). Capitals in italics such as $I$ denote sets or ranges of values. In Java bytecode, an exception handler $h(t, I, \mathtt{C})$ is defined by its type $t$, range $I$, which is an interval $[i_\alpha, i_\omega]$,[1] and handler code at `C`. Whenever an exception of type $t$ or its subtypes occurs within $I$, control is transferred to `C`. If several handlers are eligible for range $I$, the first matching handler is chosen. If, for an instruction index $a$, there exists a handler $h$ where $a$ lies within its range $I$, we say that *$h$ protects $a$*: $\text{protects}(h, a) \leftrightarrow a \in I(h)$.

As specified by the Java language [12], a `finally` block at `F` always has to be executed, whether an exception occurs or not. This is achieved by using an unspecified type $t_{any}$ for a default handler $h_d(t_{any}, I_d, \mathtt{F})$. If a `catch` block is

[1] In actual Java class files, handler ranges are defined as $[i_\alpha, i_\omega[$ and do *not* include the last index of the interval, $i_\omega$. This is only an implementation issue. For simplicity, this paper assumes that handler ranges are converted to reflect the above definition.

present in a `try`/`catch`/`finally` construct, the exception handler $h'(t', I', \mathtt{C}')$ specified by the `catch` clause takes priority over default handler $h_d$. Handler code at $\mathtt{C}'$ is only executed when an exception compatible with type $t'$ is thrown. In that case, after executing the `catch` block, a `goto` instruction is typically used to jump to the `finally` block at `F`. Because this mechanism is a straightforward augmentation of catching any exception by $h_d$, this causes no new problems for subroutine inlining and verification. Hence `catch` blocks are not discussed further in this paper.

## *2.3 Finally Blocks and Subroutines*

A `finally` block can be executed in two modes: either an exception terminated its `try` block prematurely, or no exception was thrown. The only difference is therefore the “context” in which the block executes: it possibly has to handle an exception $e$. This lead to the idea of sharing the common code of a finally block. Thus a Java compiler typically implements `finally` blocks using *subroutines.*[2] A subroutine $S$ is a function-like block of code. In this paper, $S$ will refer to the entire subroutine while `S` denotes the address of the first instruction of $S$. A subroutine can be called by a special jump-to-subroutine instruction `jsr`, which pushes the successor of the current address onto the stack. The subroutine first has to store that address in a register $r$, from which it is later retrieved by a return-from-subroutine instruction `ret`. Register $r$ cannot be used for computations. Java compilers normally transform the entire `finally` block into a subroutine. This subroutine is called whenever needed: after normal execution of the `try` block, after exceptions have been taken care of with `catch`, or when an uncaught exception occurs.

The example in Figure 1 illustrates this. Range $R$ which handler $h(t, R, \mathtt{C})$ protects is marked by a vertical line. The handler code at `C` first stores the exception reference in a local variable $e$. It then calls the `finally` block at `S`. After executing $S$, the exception reference is loaded from variable $e$ and thrown to the caller using instruction `athrow`. If no exception occurs, $S$ is called after the `try` block, before continuing execution at `X`. Note that the subroutine block is inside the entire method, requiring a `goto` instruction to continue execution at `X`, after the `try` block. In the control flow graph, $S$ can be treated as a simple block of code which can be called from the top level

[2] Sun’s J2SE compilers, version 1.4.2 and later, compile `finally` blocks without subroutines. However, in order to ensure backward compatibility with legacy bytecode, the bytecode verifier still has to deal with the complexity of allowing for correct subroutines. This underlines the need for subroutine elimination, as commercial libraries often do not use the latest available compiler but can still be used in conjunction with programs compiled by them. This paper lays the groundwork for inlining subroutines in legacy bytecode, allowing bytecode verifiers in future VMs to ignore this problem.

```
int m(int i) {
  try {
      i++;
  } finally {
      i--;
  }
  return i;
}
```

```
        |    iinc i 1
    (h) |    jsr S
        |    goto X
          C: astore e
             jsr S
             aload  e
             athrow
          S: astore r
             iinc i -1
             ret r
          X: iload i
             ireturn
```

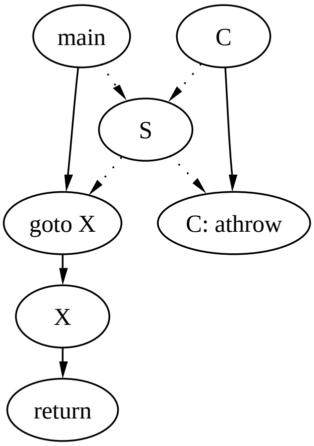


Figure 1. A simple finally block, its bytecode and its control flow graph.

of the method (main) or exception handler code `C`. In the first case, $S$ will return (with `ret`) to instruction `goto X`, otherwise to the second part of the handler ending with `athrow`.

## *2.4 Nested Subroutines*

The example in Figure 2 from [21, Chapter 16] illustrates difficulties when dealing with subroutines. It contains a nested `finally` block with a `break` statement.[3] The compiler transforms this into two exception handlers $h_1(t_1, R_1, \texttt{C}_1)$ and $h_2(t_2, R_2, \texttt{C}_2)$ using two subroutines $S_1$ and $S_2$, where it is possible to return directly to the enclosing subroutine from the inner subroutine, without executing the `ret` statement belonging to the inner subroutine. Letter $e$ denotes a register holding a reference to an exception, $r$ a register holding the return address of a subroutine call.

The corresponding control flow graph in Figure 3 is quite complex. Its two exception handlers $h_1$ and $h_2$ contain one `finally` block each. The first `finally` block contains a while loop with test `W` and loop body `L` . If the loop test fails, $S_1$ returns via `X` to the successor of its caller. This may be the second instruction, or code after $\texttt{C}_1$, which throws exception $e_1$ after having executed $S_1$. Loop body `L` contains in inner `try`/`finally` statement, compiled into

[3] The body of the method does not contain any semantically relevant operations for simplicity. The resulting code, compiled by Sun's J2SE 1.3 compiler, includes a handler protecting a `return` statement, even though that instruction cannot throw an exception. The handler may come into effect if the `try` block contains additional instructions. Therefore it is preserved in this example.

```
static void m(boolean b) {
  try {
    return;
  } finally {
    while (b) {
      try {
        return;
      } finally {
        if (b) break;
      }
    }
  }
}
```

```
(h1) |      jsr S1
     |      return
        C1: astore e1
            jsr S1
            aload e1
            athrow
        S1: astore r1
            goto W
(h2) |   L: jsr S2
     |      return
        C2: astore e2
            jsr S2
            aload e2
            athrow
        S2: astore r2
            iload b
            ifne X
         Y: ret r2
         W: iload b
            ifne L
         X: ret r1
```

Figure 2. Breaking out of a subroutine to an enclosing subroutine.

exception handler $h_2$. Execution of `L` results in calling inner `finally` block at $\mathtt{S}_2$, again prior to the `return` statement. This block will test `b` and break to the outer subroutine, which is represented by connection $\mathtt{S}_2 \rightarrow \mathtt{X}$. If `b` was false, the inner subroutine would return normally using its `ret` instruction at `Y`. There, control will return to the inner `return` statement within `L`, which then returns from the method. Both `try` blocks are also protected by default exception handlers, where the control flow is similar. The main difference is that an exception will be thrown rather than a value returned.

## 3 Inlining Java Subroutines

Once all subroutines with their boundaries have been found, they can be inlined. Inlining usually increases the size of a program only sightly [11] but significantly reduces the complexity of data flow analysis [11,21].

Table 2 defines potential successors of all bytecode instructions covered here. Without loss of generality, it is assumed that instructions are numbered consecutively. Thus $pc + 1$ refers to the successor of the current instruction, $pc - 1$ to its predecessor. Conditional branches (`ifne`) are treated non-deterministically. The `jsr` instruction is modeled to have two successors because control returns to $pc + 1$ after execution of the subroutine at $a$. Certain instructions leave the scope of the current method (`return`, `athrow`) or con-

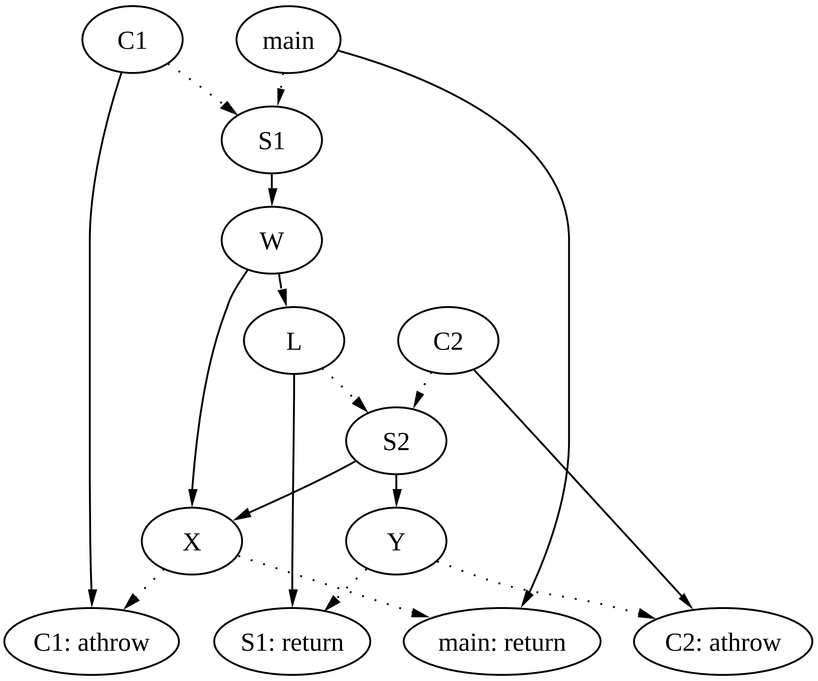


Figure 3. Control flow graph of nested subroutines.

| Instruction (at address $pc$) | Addresses of possible successors |
|---|---|
| `aload, iload, astore, istore, iinc` | $\{pc + 1\}$ |
| `goto` $a$ | $\{a\}$ |
| `ifne` $a$, `jsr` $a$ | $\{a,\ pc + 1\}$ |
| `ret, athrow, return` | $\{\}$ |

Table 2
Potential successors of Java bytecode instructions.

tinue at a special address (`ret`).

The first instruction of a method is assumed to have code index 0. A code index $i$ is *reachable* if there exists a sequence of successors from instruction 0 to $i$. $S$ is a subroutine iff $i$ is reachable and code($i$) is `jsr S`. A code index `X` is a possible *return from a subroutine* if code(`S`) is `astore` $r$, code(`X`) is `ret` $r$, and `X` must be reachable from `S` on a path that does not use an additional `astore` $r$ instruction. A code index $i$ *belongs to subroutine* $S$, $i \in S$, if there exists a possible return `X` from that subroutine `S` such that `S` $\leq i \leq$ `X`. The *end of a subroutine* $S$, eos($S$), is the highest index belonging to $S$. Note that this definition avoids the semantics of nested exception handler ranges, thus covering each nested subroutine individually. For the purpose of inlining, we also need the following definitions: The *body* of a subroutine is the code which belongs to a subroutine $S$, where for each code index $i$, $\mathtt{S} < i < \mathrm{eos}(S)$ holds. This means the body does not include the first instruction, `astore` $r$, and the last instruction, `ret` $r$. A subroutine $S_2$ is *nested* in $S_1$ if for each code index $i$ which belongs to $S_2$, $i \in S_1$ holds. From this, $\mathtt{S}_1 < \mathtt{S}_2$ and $\mathrm{eos}(S_1) > \mathrm{eos}(S_2)$ follows. Furthermore, $\mathrm{code}(\mathtt{S}_2 - 1)$ must be instruction `goto` $\mathrm{eos}(S_2) + 1$. A subroutine $S_1$ is *dependent on* a (possibly nested) subroutine $S_2$, $S_1 \prec S_2$, if there exists an instruction `jsr` $\mathtt{S}_2$ which belongs to subroutine $S_1$, where $S_2 \neq S_1$. Dependencies are transitive.

A subroutine $S_1$ which depends on $S_2$ must be inlined after $S_2$. When $S_1$ is inlined later, the calls to $S_2$ within $S_1$ have already been replaced by the body of $S_2$. Other than that, the order in which subroutines are inlined does not matter. During each inlining step, all calls to one subroutine $S$ are inlined.

### 3.1 Sufficient and Necessary Well-formedness Conditions

Java bytecode can only be inlined if certain well-formedness conditions hold. A set of necessary conditions is given by the specification of bytecode verification, which includes that subroutines must have a single entry point and that return addresses cannot be generated by means other than a `jsr` instruction [16]. Beyond these given conditions, extra conditions have to hold such that a subroutine can be inlined. Note that it is not possible that programs generated by a Java compiler violate these conditions, except for a minor aspect concerning JDK 1.4, which is described below. Furthermore, artificially generated, "malicious" bytecode that does not fulfill these well-formedness criteria will likely be rejected by a bytecode verifier. Bytecode verification is in essence an undecidable problem, and thus verifiers only allow for a subset of all possible bytecode programs to pass [16,21].

One extra condition not described here arises from the current artificial size limit of 65536 bytes per method [16]. Other limitations are structural conditions that bytecode has to fulfill. Given here is an abridged definition taken from [21]:

**Boundary.** Each subroutine $S$ must have an end $\mathrm{eos}(S)$.
If subroutine $S$ does not have a `ret` statement, then all instances of `jsr S` can be replaced with `goto S`, and no inlining is needed.

**No recursion.** A subroutine cannot call itself.

**Correct nesting.** Subroutines may not overlap:
$\nexists S_1, S_2 \cdot \mathtt{S}_1 < \mathtt{S}_2 < \mathrm{eos}(S_1) < \mathrm{eos}(S_2)$.

**No mutual dependencies.** If $S_i \prec S_j$, there must be no dependencies such that $S_j \prec S_i$. Note this property is not affected by nesting.

```
      S: astore r
         ...
         jsr S
         ...
 eos(S): ret r
```

No recursion.

```
     S1: astore r1
         ...
     S2: astore r2
         ...
eos(S1): ret r1
         ...
eos(S2): ret r2
```

Correct nesting.

```
     S1: astore r1
         ...
         jsr S2
         ...
eos(S1): ret r1
         ...
     S2: astore r2
         ...
         jsr S1
         ...
eos(S2): ret r2
```

No mutual dependencies.

```
  | :
h |    ...
  | :
       ...
    S: astore r
       ...
    C:
       ...
eos(S): ret r
```

Exception handler containment.

```
  |   :
  |      ...
h |   S: astore r
  |      ...
  |   :
         ...
eos(S): ret r
         ...
      C:
```

Handler range containment.

```
             jsr S
             ...
    |     :
    |        ...
h   |   S: astore r
    |        ...
eos |(S): ret r
    |        ...
    |     :
```

Subroutine containment in handler range.

Figure 4. Instruction sequences violating well-formedness conditions.

**Exception handler containment.** If code C of a handler $h(t, R, \texttt{C})$ belongs to $S$, then its entire range $R$ must belong to $S$ as well: $\forall h(t, R, \texttt{C}), S \cdot (\texttt{C} \in S \rightarrow R \subseteq S)$.

**Handler range containment.** If any $i \in R$ of a handler $h(t, R, \texttt{C})$ belongs to $S$, then its entire range $R$ must belong to $S$: $\forall h(t, R, \texttt{C}), S \cdot (\exists i \in R \cdot i \in S \rightarrow R \subseteq S)$.

**Subroutine containment in handler range.**
If the entire range $R$ of a handler $h(t, R, \texttt{C})$ belongs to $S$, then any instructions `jsr S` must be within $R$: $\forall h(t, R, \texttt{C}), S \cdot (R \subseteq S \rightarrow (\forall i \cdot \text{code}(i) = \texttt{jsr S} \rightarrow i \in R))$.

For the last six conditions, Figure 4 shows an example violating it. Existing Java compilers do not violate them except as described in Subsection 3.5.

```
     |   iinc i 1
(h)  |   jsr S
     |   goto X
      C: astore e
         jsr S
         aload  e
         athrow
      S: astore r
         iinc i -1
         ret r
      X: iload i
         ireturn
```

```
(h1) |   iinc i 1
         iinc i -1
(h2) |   goto X
      C: astore e
         iinc i -1
         aload  e
         athrow
      X: iload i
         ireturn
```

Figure 5. Inlining a subroutine.

## 3.2 Control Transfer Targets

When inlining subroutines, the body of a subroutine $S$ replaces each subroutine call. This part of inlining is trivial, as shown by the example in Figure 5. The two inlined copies of $S$ which replace the `jsr` instructions are shown in **bold** face. Difficulties arise with jump targets, which have to be updated after inlining. Inlining eliminates `jsr` and `ret` instructions; therefore any jumps to these instructions are no longer valid. Furthermore, there can be jumps inside a subroutine to an enclosing subroutine or the top level of the code, such as shown in Figure 6. Therefore, the inlining algorithm has to update jump targets during several inlining steps and also to consider copies, for each instance of a subroutine body that gets inlined.

The algorithm uses two *code sets, current* set $B$ and *new* set $B'$. During each inlining step, all instructions in $B$ are moved and possibly duplicated, creating a new set of instructions $B'$ which becomes the input $B$ for the next inlining step.

Each address in $B$ must map onto an *equivalent* address $B'$. Each possible execution (including exceptional behavior) must execute the same sequence of operations, excluding `jsr` and `ret`, in $B$ and $B'$. Code indices in $B$ referring to `jsr` or `ret` instructions must be mapped to equivalent indices in $B'$. The most straightforward solution is to update all targets each time after inlining *one* instance of a given subroutine. This is certainly correct, but also very inefficient, because it would require updating targets once for each `jsr` instruction rather than each subroutine.

Instead, our algorithm uses a *mapping* $M$, a relation $I \times I'$ of code indices mapping an index $i \in I$ to a set of indices $\{i'_0, i'_1, \ldots, i'_k\} \in I'$. This relation, initially empty, records how an address in $B$ is mapped to one or several addresses in $B'$. Each time an instruction at index $i$ is moved or copied from the current code set $B$ to the new code set $B'$ at index $i'$, $i \mapsto i'$ is added to the mapping.

Each subroutine is processed inlining *all* its instances in one step, with the innermost subroutines being inlined first. Instructions not belonging to the subroutine which is being inlined and which are not a `jsr S` operation are copied over from $B$ to $B'$. Each occurrence of `jsr S` is replaced with the body of $S$. The key to handling jumps to $ins_j$, the `jsr S` instruction itself, and to $ins_r$, the `ret` instruction in the subroutine, is adding two extra entries to $M$. The first one is $i_j \rightarrow i'_0$ where $i_j = \text{index}(ins_j)$ and $i'_0 = M(\mathtt{S})$, the index where the first instruction of the subroutine has been mapped to. The second one is $i_r \rightarrow i'_r$ where $i_r = \text{index}(ins_r)$ and $i'_r = M(\text{eos}(S) + 1)$, the index of the first instruction *after* the inlined subroutine.

In the following discussion, a *forward jump* is a jump whose target code index is greater than the code index of the instruction. Similarly, a *backward jump* is a jump leading to a smaller code index. If bytecode fulfills the correctness criteria described above, the correctness of the algorithm can be proved as follows:

- A target outside $S$ is mapped to a single target and therefore trivially correct.
- A target in the body of $S$ is mapped to several targets in the inlined subroutines $S'$, $S''$ etc., one for each `jsr S` in $B$. Let the jump instruction in $B$ be at code index $i$ and the target at $a$. Given $i'$, the index of the jump instruction in $B'$, the *nearest* target in the current mapping has to be chosen which still preserves the fact that a jump is either a forward or backward jump.
  For a forward jump, index $\min_{a'} \cdot (a \mapsto a' \in M) \wedge (a' > i')$ is the correct index. This can be shown as follows: Address $a$ is either outside $S$, in which case the *code*$(a)$ has not been duplicated and there is only one $a' \cdot a \mapsto a' \in M$. If $a$ is inside $S$, $a'$ is necessarily the nearest target to $i'$ in that direction: The code at index $a$ has been copied to $a'$ during the inlining of $S$ to $S'$. The first instruction of the inlined copy of $S'$ is at index $\mathtt{S}'_\alpha$ and the last instruction is at $\mathtt{S}'_\omega$. Since $i$ belongs to $S$, $\mathtt{S}'_\alpha \leq i' \leq \mathtt{S}'_\omega$ holds. No other code than $S'$ has been copied to positions inside that interval, and $\mathtt{S}'_\alpha \leq i' < a' \leq \mathtt{S}'_\omega$ holds because $a$ belongs to $S$ and the jump is a forward jump. Any other copies of the instructions at $a$ are either copied to an index $a'' < \mathtt{S}'_\alpha$, and therefore $a'' < i'$, or $a'' > \mathtt{S}'_\omega$, and therefore $a'' > a'$. Backward jumps are treated vice versa.
- A jump to a `jsr S` instruction in $B$ indirectly executes code at `S`. Mapping it to the $\mathtt{S}'_\alpha$ preserves the semantics.

```
(h1) |      jsr S1
     |      return
       C1:  astore e1
            jsr S1
            aload e1
            athrow
       S1:  astore r1
            goto W
(h2) |  L:  jsr S2
     |      return
       C2:  astore e2
            jsr S2
            aload e2
            athrow
       S2:  astore r2
            iload b
            ifne X
        Y:  ret r2
        W:  iload b
            ifne L
        X:  ret r1
```

```
(h1) |      jsr S1
     |      return
       C1:  astore e1
            jsr S1
            aload e1
            athrow
       S1:  astore r1
            goto W
        L:  iload b
            ifne X
(h2) |      return
       C2:  astore e2
            iload b
            ifne X
            aload e2
            athrow
        W:  iload b
            ifne L
        X:  ret r1
```

```
                 goto W1
            L1:  iload b
                 ifne X1
(h2_1) |         return
          C2_1:  astore e2
                 iload b
                 ifne X1
                 aload e2
                 athrow
            W1:  iload b
                 ifne L1
(h1) |      X1:  return
            C1:  astore e1
                 goto W2
            L2:  iload b
                 ifne X2
(h2_2) |         return
          C2_2:  astore e2
                 iload b
                 ifne X2
                 aload e2
                 athrow
            W2:  iload b
                 ifne L2
            X2:  aload e1
                 athrow
```

Figure 6. Inlining a nested subroutine in two steps

- A jump to the last instruction in a subroutine will return to the successor of its `jsr S` instruction. Therefore mapping the code index of the `ret` instruction to the successor of the last inlined instruction of the body of $S$ produces the same effect in the inlined code. Note that there always exists an instruction following a `jsr` instruction [16], such as `return`.

Two of these cases are shown in the second inlining step of Figure 6, the inlining of the subroutines in Figure 2. In both inlined instances of $S_1$, the outer subroutine, there is a jump to `W` inside the subroutine and to `X`, the index of the `ret` instruction of $S_1$. By inlining $S_1$, both code indices are mapped to two new indices, $\{\mathtt{W}_1, \mathtt{W}_2\}$, and $\{\mathtt{X}_1, \mathtt{X}_2\}$, respectively. The semantics of jumps are preserved as described above.

## 3.3 Exception Handler Splitting

If a `jsr S` instruction $ins_j$ is protected by an exception handler $h(t, R, \mathtt{C})$, where $R = [r_\alpha, r_\omega]$ does *not* extend to the subroutine itself, then that handler must *not* be active for the inlined subroutine. A simple example is shown in Figure 5, where the `jsr` instruction is in the middle of the exception handler range. Therefore, to solve this problem, the exception handler must be *split* into two handlers $h_1(t, R_1, \mathtt{C}')$ and $h_2(t, R_2, \mathtt{C}')$. The new ranges are $R_1 =$

$[r'_a, r_\beta]$ and $R_2$, with $r'_\alpha = M(r_\alpha)$ and $r_\beta = M(\mathit{index}(\mathit{ins}_j) - 1)$, the mapped code index of the predecessor of the `jsr` instruction. In $R_2 = [r_\gamma, r'_\omega]$, $r_\gamma = M(\mathit{index}(\mathit{ins}_r))$, the mapped code index of the successor of the last instruction of the inlined subroutine body, and $r'_\omega = M(r_\omega)$.

Splitting handlers is necessary to ensure correctness of the inlined program. There exist cases where $R_1$ or $R_2$ degenerates to an interval of length zero and can be removed altogether. Splitting may increase the number of exception handlers exponentially in the nesting depth of a subroutine. This number is almost never greater than one, though, and only few exception handlers are affected by splitting.

### *3.4 Exception Handler Copying*

If a subroutine $S$, but not the `jsr S` statement, is protected by an exception handler, this protection also has to be ensured for the inlined copy of the subroutine. Therefore, all exception handlers protecting subroutine $S$ have to be *copied* for each inlined instance of $S$. Figure 6 shows a case where inlining the outer subroutine $S_1$ causes the exception handler $h_2$ inside that subroutine to be duplicated.

Note that this duplication does not occur if *both* the `jsr` instruction and the subroutine are protected by the same handler. In this case, the inlined subroutine is automatically included in the mapped handler range. Copying handlers may increase the number of handlers exponentially, which is not an issue in practice because the *innermost* subroutine, corresponding to the innermost `finally` block, is never protected by an exception handler itself, reducing the exponent by one.

### *3.5 Violation of Well-formedness Conditions in JDK 1.4*

The original implementation exhibited problems with some class files compiled with the JDK 1.4 compiler. The reason were changes in the compiler, designed to aid the bytecode verifier of the VM. When compiling the program from Figure 1, the resulting instructions are the same, but the exception handlers are different: The original handler covered three instructions, the initial increment instruction, the `jsr`, and the `goto` which jumps to the end of the program. The handler from the 1.4 compiler omits the `goto`. This does not change the semantics of the code because the `goto` instruction cannot raise an exception.

However, a *second* handler $\overline{h}$ is installed by the newer compiler, which covers the first two instructions of the exception handler code (at label `C`),

| Number of calls | 1 | 2 | 3 | 4 | 5 | 6 – 10 | 11 – 20 | 28 |
|---|---|---|---|---|---|---|---|---|
| Number of subroutines | 1 | 783 | 173 | 23 | 9 | 8 | 3 | 1 |

Table 3
Number of calls per subroutine, determining how often its code is inlined.

`astore` $e$ and the second instance of `jsr S`. The situation is exacerbated by the fact that $\overline{h}$ is recursive; the handler code has the same address as the first instruction protected by it. This could (theoretically) produce an endless loop of exceptions. The result of inlining $\overline{h}$ is a handler covering only the `astore` instruction (since the inlined subroutine is outside the handler range). Fortunately, the `astore` instruction cannot throw an exception, so no changes are needed in the VM to avoid a potentially endless loop.

Newer JDK compilers (1.4.2 and later) generate subroutines in-place. The result is identical to inlined code from JDK 1.4, including spurious handler $\overline{h}$.

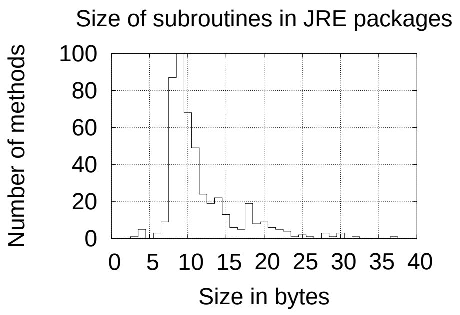
### 3.6 *Costs of Inlining*

Inlining subroutines increases code size only slightly. Subroutines are rare. In Sun's Java run-time libraries (version 1.4.1), out of all 64994 methods from 7282 classes (ignoring 980 interfaces), only 966 methods (1.5 %) use 1001 subroutines. None of them are nested. Table 3 shows that subroutines are usually called two to three times each, with a few exceptions where a subroutine is used more often.

The histogram to the left in Figure 7 shows that most subroutines measure only between 8 and 12 bytes; 626 subroutines were 9 bytes large, hence that entry is off the scale. No subroutine was larger than 37 bytes. Inlining usually results in a modest code growth of less than 10 bytes. This is shown by the histogram to the right where entries with an even and odd number of bytes are summarized in one bucket. Entries off the scale are values 0 (64041 methods, including those without subroutines) and 2, representing 571 methods where code size increased by 2 or 3 bytes. 10 methods grew by more than 60 bytes, 186 bytes being the worst case. Inlining all subroutines of JRE 1.4.1 would result in a code growth of 5998 bytes, which is negligible compared to the entire run-time library, measuring 25 MB.

## 4 Abstract, Register-based Bytecode

Java bytecode contains 201 instructions [16], many of which are variants of the same type. For instance, 25 instructions load a register on the stack. Variants include several instructions for each data type, one generic variant (e.g. `iload` $r$) and short variants like `aload_0`, where $r$ is hard-coded. A re-



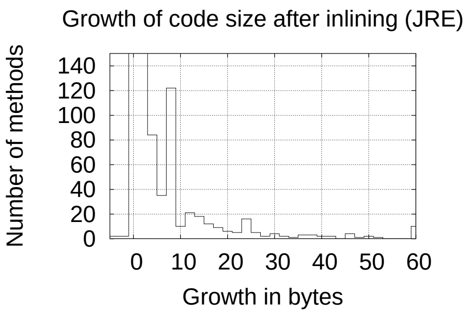


Figure 7. Sizes of subroutines and size increase after inlining.

duction of the instruction set is an obvious simplification. We use abstract bytecode [21] as the reduced format, where argument types and hard-coded indices are folded into the parametrized version of a generic instruction. For instance, `aload_0` becomes `Load` "ref" 0. This reduction is independent of bytecode inlining. The previous section described inlining using normal bytecode to allow for stand-alone inlining algorithms.

Instructions not implemented in [21] include arithmetic instructions, of which implementation is straightforward. Unsupported instructions are `switch` (for control flow), `monitorenter` and `monitorexit` (for multi-threading), and the `wide` instruction that modifies the parameter size of the subsequent instruction. The first three instructions have to be implemented according to the standard bytecode semantics [16] while the `wide` instruction is an artefact of the fact that Java bytecode was initially targetted to embedded systems with little memory for instructions. In our implementation [5] of the abstract bytecode instruction set, we extended the size of any instruction parameters to four bytes and thus could eliminate the wide instruction trivially, by converting all instruction arguments to a four-byte format.

Abstract bytecode only has 31 instructions, which is already a great simplification of the original instruction set. However, the usage of a (fixed-size) stack makes data flow analysis needlessly difficult, since the exact stack height at each index, though known at compile-time, has to be computed first after loading a class file. This computation is normally part of bytecode verification in the class loader. Furthermore, the treatment of stack and local variables (registers) results in pairs of instructions that essentially perform the same task: `load` pops the top element from the stack while `store` pushes a register onto the stack. Finally, 64-bit values are treated as a single stack element, but as a pair of local variables. This creates a need for case distinctions for many instructions [16]. The specification requires that the second slot of the local variables holding a 64-bit value is never used, and that the stack semantics

| Bytecode variant | Java [16] | Abstract [21] | Register-based |
|---|---|---|---|
| Instruction set size | 201 | 31 | 25 |
| Variants (type/index) per instruction | up to 25 | 1 | 1 |
| Bytecode subroutines | yes | yes | no |
| Wide instructions | yes | not impl. | eliminated |
| Special treatment of 64-bit values | yes | not impl. | eliminated |
| Register location | implicit | implicit | explicit |

Table 4
The benefits of register-based bytecode.

are preserved when pushing a 64-bit value onto it.

Because the height of the stack is known for each instruction, we converted the stack-based format of abstract bytecode to an explicit representation where each stack element is converted to a register. When using registers, stack elements and local variables can be treated uniformly, merging `Load` and `Store` into a `Get` instruction, and eliminating more instructions such as `Pop`, `Swap`, or `Dup`. Of all conversions, converting the `Dup` instruction was the only non-trivial one and actually proved to be quite difficult. Some variants of this instruction do not only copy the top element(s) of the stack, but insert it "further down", below the top element. There exist six variants of `Dup` instructions, and the treatment of data flow requires up to four case distinctions per instruction variant, due to 64-bit values [16]. We convert all `Dup` instructions into an equivalent series of `Get` instructions. This unfortunately introduces sequences of instructions that corresponds to only one original instruction, which makes further treatment slightly more complex; but it still eliminates the case distinctions for 64-bit values, which is the greater overhead.

This conversion to register-based bytecode reduces the size of the final instruction set to a mere 25 instructions. The remaining instructions are (refer to [16,21] for their semantics): `ALoad`, `AStore`, `ArrayLength`, `Athrow`, `Checkcast`, `Cond`, `Const`, `Get`, `GetField`, `GetStatic`, `Goto`, `Inc`, `Instanceof`, `InvokeSpecial`, `InvokeStatic`, `InvokeVirtual`, `MonitorEnter`, `MonitorExit`, `New`, `NewArray`, `Prim`, `PutField`, `PutStatic`, `Return`, `Switch`. This instruction set was used in JNuke and has been tested in over 1,000 unit and system tests using static analysis, run-time verification, and software model checking [2,3,5].

## 5 Related Work

Previous work has investigated difficulties in analyzing Java bytecode arising from its large instruction set and subroutines. Inlining bytecode subroutines has been investigated in the context of just-in-time-compilation [15] or as a preprocessing stage for theorem proving [11]. The latter paper also describes an alternative to code duplication for inlining: by storing a small unique integer for each subroutine call instruction in an extra register, subroutines can be emulated without using a `jsr` instruction. However, the size gain by this strategy would be small, and bytecode verifiers would again have to ensure that the content of this extra register is never overwritten inside the subroutine, which would leave one of the major problems in bytecode verification unsolved. Therefore this direction was never pursued further.

Challenges in code analysis similar to those described here occur for *decompilation,* where the structure of subroutines must be discovered to determine the correct scope of `try`/`catch`/`finally` blocks. The Dava decompiler, which is part of the Soot framework, analyzes these structures in order to obtain an output that correctly matches the original source program [19]. Soot also eliminates `jsr` instructions through inlining [22]. However, no algorithm is given. Details on how to handle nested subroutines are missing.

As a part of work on $\mu$Java [14], another project also performs a kind of subroutine inlining called subroutine *expansion* [24]. The main difference is that the expanded code still contains `jsr` instructions, making it easier to ensure correctness of the inlined code, but still posing a certain burden on the bytecode verifier that our work eliminates. The inlining algorithm differs in several points. First, it uses "complex addresses" to track code duplication. Second, it does not inline subroutines in the order of their nesting. This has two side-effects: treatment of nested subroutines creates a very complex special case, and the expanded code may be larger than necessary [24]. Our algorithm uses a simple mapping instead of complex addresses, which, together with inlining subroutines in the order in which they are nested, greatly simplifies the adjustment of branch targets and exception handler ranges. Furthermore, with nesting taken care of by inlining subroutines in nesting order, no special treatment of nested subroutines is necessary in the inner loop that performs the actual inlining.

Instruction set reduction on Java bytecode has been performed in other projects in several ways. The Carmel [17] and Jasmin [18] bytecode instruction sets both use a reduced instruction set similar to abstract bytecode [21]. The Bytecode Engineering Library (BCEL) does not directly reduce the instruction set but features an object-oriented representation of bytecode instructions where super classes combine related instructions [9]. The project most similar to ours with respect to instruction abstraction is Soot. The Jimple language from Soot is a bytecode-like language using 3-address code instead of stack-based instructions, making it suitable for analysis and optimization [23].

## 6 Conclusions

Java bytecode is far from ideal for program analysis. Subroutines, a construct not available in the Java language but only in Java bytecode, make data flow analysis very complex. Eliminating subroutines is difficult because subroutines can be nested, and they can overlap with exception handlers. In practice, inlining does not increase program size much, while greatly simplifying data flow analysis. This is especially valuable as subroutines are disappearing in modern compilers but still have to be supported by virtual machines for backward compatibility.

Abstracting sets of similar instructions to a single instruction greatly reduces the instruction set. Converting the stack-based representation to a register-based one makes computational operands explicit and further reduces the instruction set. Finally, eliminating certain bytecode-specific issues, such as wide instructions and differences of 64-bit variables and stack elements, simplifies the code even further. The resulting instruction set was successfully used in the JNuke framework for static and dynamic analysis, which greatly benefits from the simplified bytecode format.

## Acknowledgements

Many thanks go to Robert Stärk for sharing his profound knowledge of the intricacies of bytecode correctness and his input throughout this work.

## References

[1] *1st, 2nd, 3rd and 4th Workshops on Runtime Verification (RV'01 – RV'04)*, volume 55(2), 70(4), 89(2), 24(2) of *ENTCS*. Elsevier Science, 2001 – 2004.

[2] C. Artho and A. Biere. Combined static and dynamic analysis. In *Proc. AIOOL '05*, ENTCS, Paris, France, 2005. Elsevier Science.

[3] C. Artho, A. Biere, and K. Havelund. Using block-local atomicity to detect stale-value concurrency errors. In Farn Wang, editor, *Proc. ATVA '04*. Springer, 2004.

[4] C. Artho, K. Havelund, and A. Biere. High-level data races. *Journal on Software Testing, Verification & Reliability (STVR)*, 13(4), 2003.

[5] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In R. Alur and D. Peled, editors, *Proc. CAV '04*, Boston, USA, 2004. Springer.

[6] P. Bothner. Kawa — compiling dynamic languages to the Java VM. In *Proc. USENIX 1998 Technical Conference, FREENIX Track*, New Orleans, USA, june 1998. USENIX Association.

[7] E. Briot. JGNAT: The GNAT Ada 95 environment for the JVM. In *Ada France*, France, September 1999.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Symp. Principles of Programming Languages*. ACM Press, 1977.

[9] M. Dahm. BCEL, 2005. http://jakarta.apache.org/bcel.

[10] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI 2002*, pages 234–245, Berlin, Germany, 2002. ACM Press.

[11] S. Freund. The costs and benefits of Java bytecode subroutines. In *Formal Underpinnings of Java Workshop at OOPSLA*, Vancouver, Canada, 1998.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[13] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[14] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, 2003.

[15] S. Lee, B. Yang, S. Kim, S. Park, S. Moon, K. Ebcioglu, and E. Altman. Efficient Java exception handling in just-in-time compilation. In *Proc. ACM 2000 conference on Java Grande*, pages 1–8, USA, 2000. ACM Press.

[16] T. Lindholm and A. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[17] R. Marlet. Syntax of the jcvm language to be studied in the secsafe project. Technical Report SECSAFE-TL-005-1.7, Trusted Logic SA, Versailles, France, 2001.

[18] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly & Associates, Inc., 1997.

[19] J. Miecznikowski and L. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *Proc. 11th CC*, pages 111–127, Grenoble, France, 2002. Springer.

[20] M. Mohnen. A graph-free approach to data-flow analysis. In *Proc. 11th CC*, pages 46–61, Grenoble, France, 2002. Springer.

[21] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer, 2001.

[22] R. Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, Montreal, 2000.

[23] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot – a Java optimization framework. In *Proc. CASCON 1999*, pages 125–135, 1999.

[24] M. Wildmoser. Subroutines and Java bytecode verification. Master's thesis, Technical University of Munich, 2002.