Structural Encoding of Static Single Assignment Form

Andreas Gal Christian W. Probst Michael Franz [1]

Donald Bren School of Information and Computer Science
University of California
Irvine, CA, 92697, USA

Abstract

Static Single Assignment (SSA) form is often used as an intermediate representation during code optimization in Java Virtual Machines. Recently, SSA has successfully been used for bytecode verification. However, constructing SSA at the code consumer is costly. SSA-based mobile code transport formats have been shown to eliminate this cost by shifting SSA creation to the code producer. These new formats, however, are not backward compatible with the established Java class-file format. We propose a novel approach to transport SSA information implicitly through structural code properties of standard Java bytecode. While the resulting bytecode sequence can still be directly executed by traditional Virtual Machines, our novel VM can infer SSA form and confirm its safety with virtually no overhead.

Key words: mobile code, verification, optimization

1 Introduction

Java programs are shipped in a platform-independent bytecode format. To execute such programs, a Virtual Machine (VM) can choose to simply interpret the bytecode instruction by instruction. This results in a significant loss of execution performance in comparison to native machine code execution. Just-in-time (JIT) compilers are used to dynamically translate portable bytecode into native machine

[1] Email: {gal,probst,franz}@uci.edu
This research effort was partially funded by the National Science Foundation (NSF) under grants TC-0209163 and ITR-0205712 and by the Office of Naval Research (ONR) under agreement N00014-01-1-0854. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, the Office of Naval Research, or any other agency of the U.S. Government.

code, which is directly executed by the underlying physical CPU, eliminating most of the overhead that interpretation causes.

Code optimization is an essential part of many dynamic code-generation systems. Many optimizations cannot be applied ahead of time by the code producer due to the wide range of possible target architectures the bytecode has to run on. Performing aggressive common-subexpression elimination, for example, while likely beneficial on a RISC architecture with many registers, might severely degrade performance on a CISC architecture such as x86 by increasing register pressure and introducing unneeded spills to memory. As a consequence, optimization has to be performed by the code consumer—the JIT compiler.

Being stack-based, bytecode is not well suited to perform code optimizations on. Instead, it often is translated into an intermediate representation such as Static Single Assignment (SSA) form [5]. In SSA form, variables are split into multiple instances such that every new variable instance is defined exactly once. At control-flow merge points special ϕ-instructions are inserted to merge variable instances and to assign the proper value to a new and unique instance of that variable. Through this transformation, SSA form embeds definition-use information into the program representation.

We recently have shown how to use SSA for bytecode verification [9,10]. To enable this approach it is important to make the SSA form of an incoming program available efficiently. From an algorithmic perspective, transforming bytecode into SSA requires finding dominators in a flow graph [3,12], and the calculation of iterated dominance frontiers [2]. While both problems have been shown to be linear in theory [20], they still incur a not-negligible runtime overhead. Approaches such as SafeTSA [1] avoid this overhead by transforming bytecode into SSA at the code producer. However, to ship the mobile code, SafeTSA defines a new and incompatible class-file format. While it is possible to avoid such compatibility problems by shipping the SSA-based representation as an optional annotation, this option severely inflates the size of class files as code is effectively represented twice. It would also allow inconsistencies between the two formats. The lack of a compatible transport format has hampered the adoption of SSA-based mobile code formats.

We propose a novel approach to SSA-based mobile-code representation. Instead of introducing a new and incompatible bytecode format, we use the *existing* bytecode format and transport SSA information through the *modification of structural code properties* such as local-variable mappings and basic-block ordering. The resulting bytecode is fully compatible to the Java Standard and can be executed by traditional Java VMs. A specially crafted VM, however, can directly infer SSA form and verify the correctness of the deducted SSA-based representation in linear time.

The remainder of this paper is organized as follows: In Section 2 we give a brief overview of the Java bytecode format. Section 3 describes the bytecode transformation process we use to encode SSA-information. Section 4 discusses the encoding, decoding, and verification of dominator information. Related work is discussed in Section 5. Section 6 contains our conclusions and discusses future work.

2 Java Bytecode

JVML instructions read and store intermediate values in two locations: the operand stack and local variables. The types of these locations are flow sensitive in that the same stack cell or local variable can hold values of different types during program execution. Verification ensures that locations are used consistently and intermediate values are always read back with the same type that they were originally written as.

Verification also ensures control-flow safety, but this is a comparatively trivial task. Conversely, verifying that the data flow is *well-typed* is rather complex. The JVM bytecode verifier [13,14,23] uses an iterative data-flow analysis and an abstract interpreter for JVML instructions. Unlike in the JVM, the stacks and local variables of the abstract interpreter used for verification store *types*, rather than *values*. From the perspective of the verifier, JVM instructions are operations that execute on types and not on values.

JVML verification works at method level. If every method is verifiable, the whole program is verifiable. For the remainder of this paper, we use program and method interchangeably, and only consider a subset of JVML that does not contain the subroutine construct. Subroutines are a significant complication when dealing with Java bytecode [8,11,18,21] and have been shown to be a very ineffective way of reducing code size [7]. The compiler in the current Java version 1.5 does no longer generate subroutines. Our prototype implementation resolves the rare occurrence of a subroutine by inlining it into the body of the calling method[2] .

3 Bytecode in Static Single Assignment Form

Traditional JVML bytecode does not comply with SSA form. Values can be written to local variables or into stack cells, and there is no requirement to use fresh local variables or stack cells for each new definition. However, there is also nothing that would stop a code producer to emit a Java bytecode program in which values are held in local variables with each definition being assigned its own local variable. In this section we describe a simple transformation that takes regular Java bytecode and translates it into a form that permits the code consumer to infer SSA form even though the code is still transported as pure JVML bytecode. To ensure that the code consumer can not only *extract* SSA information, but also *verify* that it is safe to use it, we additionally encode dominator-tree information by re-arranging the sequence of basic blocks. Using the dominator tree we can traverse the code and type-check uses and their corresponding definition in linear time and in a single sweep.

[2] We have studied numerous bytecode applications including the Eclipse framework, different Java APIs, and the SPEC benchmarks. Of approximately 5.4 million instructions we only found 0.24% to be in subroutines. The average size of a subroutine was 7 instructions, and it was only called 2 times.

3.1 Static Single Assignment Form

Most JVML instructions use the stack to access operands and store calculated values. To permit the code consumer to infer Static Single Assignment form for such instructions, we encapsulate them with local variable access instructions such as `iload` and `istore`.

The expression $a = b+c*d$, for example, can be calculated in JVML as follows:

```
iload 1    // b
iload 2    // c
iload 3    // d
imul       // c * d
iadd       // b + c * d
```

In this example, b, c, and d are read from local variable 1, 2, and 3 respectively After multiplying c and d and adding the result to a, the final value is left on the stack.

In our enriched transport format, we use Shaylor's approach [19] to eliminate the operand stack. In the transformed code, each instruction reads its arguments directly from a local variable and the stack is always empty between instructions:

```
iload 2    // c
iload 3    // d
imul       // c * d
istore 5   // temp = c * d
iload 1    // b
iload 5    // temp
iadd       // b + temp
istore 4   // a = b + c * d
```

Instead of passing the temporary result of $c*d$ through the stack, it is assigned to a fresh local variable 5 to ensure that the stack is empty between `imul` and `iadd`. Effectively, we turn the stack-based JVML representation into a register-based representation in SSA form. A traditional JVM would obviously calculate the same result for these two code fragment, albeit taking slightly more time to complete the operations of the transformed fragment, as the code is more verbose. An aware JVM, however, can easily detect that each local variable is assigned exactly once, allowing to skip the renaming phase, directly obtaining SSA form for the code.

By transforming the program into a register-based format, a number of JVML instructions become obsolete. The JVML instruction set can be divided in two kinds of instructions: *core* instructions and *data-flow* instructions. Core instructions operate on values stored on the operand stack, while data-flow instructions such as `dup`, `dup_2`, `iload_`x, and `istore_`x only facilitate the flow of values between core instructions by manipulating the state of the operand stack and exchanging values between operand stack and variables.

Values are produced by core instructions and can be consumed by other core instructions. During the lifetime of a value it can reside on the operand stack or in variables. Values can reside in multiple locations at the same time. Data-flow instructions neither produce nor consume values, but merely transport values between stack locations and variables [9,10].

During the transformation, the code producer eliminates all data-flow instructions and replaces them with direct references to SSA variables. The following code fragment calculates $2 * 2$:

```
iconst_2
dup
imul
istore 1
```

After the transformation, the value generated by `iconst_2` is stored in local variable 2. The `dup` instruction is removed, and the `imul` instruction is transformed to directly point to the instruction that defines its operands, which is the `istore_2` instruction newly introduced for `iconst_2`. Any future use of the result is replaced with a direct reference to local variable 1, which holds the result of the multiplication:

```
iconst_2
istore 2
iload 2
iload 2
imul
istore 1
```

After the transformation, the code does not contain any more data-flow instructions except for load/store instructions encapsulating core instructions.

3.2 Control-Flow Merges

Besides the single-assignment property, ϕ-instructions are the most important feature of SSA form. ϕ-instructions are used to merge definitions along multiple incoming control-flow edges. JVML does not have a ϕ-instruction, thus we need an alternative way to represent them. Adding a new instruction is not an option, because we want to maintain backward compatibility.

Instead, we use the JVM operand stack to hand over values between basic blocks at control-flow merges. The ϕ-operands are pushed onto the stack at the end of each basic block that targets a basic block with multiple predecessor blocks, and each such merge block pops the ϕ-operands from the stack and stores them in appropriate (fresh) local variables. Thus, the ϕ-instruction $l_3 = \phi(l_1, l_2)$, which joins the definitions of local variables 1 and 2 and leaves the result in local variable 3 is represented as:

```
  iload 1
  goto L1
  ...
  iload 2
  goto L1
  ...
L1:
  istore 3
```

This approach works for regular control-flow edges, but cannot be applied to exception handlers. Exception handlers automatically purge the stack upon invocation and push the exception object on the emptied stack. Instead, we use temporary local variables to communicate ϕ-operands from regular code to exception handlers. To ensure that the code consumer can easily recognize ϕ-operands of exception edges, each instruction that can trigger an exception is preceded by corresponding local variable load and store instructions to prepare for a potential exception exit:

```
iload 1
istore 9
iload 2
istore 10
iload 1
iload 2
idiv
istore 3
```

Here, local variables 9 and 10 are used as temporaries to hold the ϕ-operands for the exception handler guarding the `idiv` instruction. For a traditional JVML this representation incurs a slight runtime overhead during interpretation. Optimizing JIT compilers are likely to detect the unused variables and eliminate the extra `istore` instructions (dead code elimination).

To evaluate the impact of the transformation on the size of bytecode programs, we used the encoder to transform 670 classes from JDK 1.4.2 into SSA-inferable form. On average, the class-file size was increased by 30%.

4 Dominator Information

To construct the SSA-form for a program, the code consumer needs access to dominator information. This is also needed for linear-time verification of JVML bytecode in SSA-form. While multiple approaches exist to compute the dominator tree efficiently from a control-flow graph [12,3,4], our approach uses the fact that the code producer either already has the dominator tree or can easily obtain it from the control-flow graph. The dominator tree is then used to rearrange the basic blocks in such a way that the code consumer can reconstruct the dominator tree instead of computing it from scratch.

We first describe the decoding of the dominator relation from the ordering of basic blocks. The decoding process essentially governs how basic blocks have to be arranged by the encoder. Section 4.2 then describes the process of actually encoding the basic blocks, Section 4.3 shows some performance measurements for a prototype, and Section 4.4 presents how to verify the computed dominator information.

4.1 Decoding Dominator Information

The decoder constructs the dominator tree solely from the information available in the program it receives—the ordering of basic blocks as computed by the encoder and the control-flow edges between basic blocks. Decoding the dominator information from the received basic-block stream works in two phases. First, an initial approximation to the dominator tree is constructed. The second phase corrects erroneous and missing edges.

To limit the problem space and for efficiency reasons we restrict the way in that the approximation computed in the first phase may deviate from the correct dominator relation. Since the immediate-dominator relation forms a tree, each node has exactly *one* predecessor, but possibly *several* successors. This property can be used to limit the search space during correcting the approximation in the second phase. When moving nodes that have been misplaced in the approximation *upwards*, it is always clear which edge to follow. In contrast, when moving nodes *downwards*, the second phase would have to determine which edge to chose—this would require to visit the subtrees reachable via these edges.

Before we describe the two phases in more detail, we note a special relation between a node, its predecessors, and its immediate dominator. If d is the immediate dominator of n, then each predecessor of n is either d or is dominated by d.

Theorem 1 *Let G be a graph with nodes N and edges $E \subseteq N \times N$. Let S be the entry node of G and* DOM $\subset N \times N$ *be the immediate dominator relation of nodes in G with the usual representation $(n_1, n_2) \in$* DOM *if n_1 is the immediate dominator of n_2. dominates $: N \to 2^N$ maps a node n to all nodes in N that are dominated by n. Then the following statements are equivalent:*

- $(n_1, n_2) \in$ DOM
- $\forall n \in N$, *with* $n \neq n_1, (n, n_2) \in E$: $n \in dominates(n_1)$

Proof. For an edge between nodes n_1 and n_2 we write $n_1 \to n_2$ and for a (possibly empty) path $n_1 \rightsquigarrow n_2$.
We prove both directions by indirection. $\Rightarrow$. Assume that there exists a node $n \in N$ with $n \neq n_1$, $(n, n_2), (n_1, n_2) \in E$ but $n \notin dominates(n_1)$. Then, since n_1 does not dominate n, there is a path $S \rightsquigarrow n$ that does not lead through n_1, and since $(n, n_2) \in E$ there exists a path $S \rightsquigarrow n \to n_2$ from S to n_2 that does not lead through n_1. Therefore, $(n_1, n_2) \notin$ DOM. $\Leftarrow$. Assume that n_1 is not the immediate dominator of n_2, that is it exists a path $S \rightsquigarrow n_2$ that does not lead through n_1. Assume that the last step of this path is the edge (n, n_2) with $n \neq n_2$. Since the path does not go through n_1, n is a predecessor of n_2 that is not immediately dominated by n_1. □

These facts directly lead to a way to compute an approximation of the dominator tree in the first phase of the decoder (Figure 1). The input to the decoder is a sequence of basic blocks. The decoder constructs the dominator tree bottom-up, by inserting nodes always above the nodes that have already been inserted. The set *top* contains all nodes in the current dominator tree that have not yet been assigned an immediate dominator. Whenever a basic block n is read, the decoder determines its control-flow successors. If the set *top* contains a successors s of n, a dominator edge is inserted between n and s, and s is removed from *top*. Finally, n is added to *top*. Due to Theorem 1, the immediate dominator of s is either n or dominates n. Thus, s is inserted below its immediate dominator, which we make use of by moving nodes only upwards in the second phase.

The second phase keeps track of all nodes in the approximation that have a control-flow predecessor that is not their immediate dominator. Following Theorem 1, each of their control-flow predecessors must be dominated by their immediate dominator. For each such node n, the decoder walks up in the approximated

```
decode()
    // Phase 1
    $top = \emptyset$
    read next basic block $bb$
    while (basic blocks available)
        $dominates(bb) = \{bb\}$
        for all nodes $n \in succs(bb)$ that are in $top$
            add $n$ to $dominates(bb)$ and add $(bb, n)$ to DOM
            remove $n$ from $top$
        read next basic block $bb$
    $\forall n \in top$ add $n$ to $dominates(bb)$ and set $n$'s IDOM to $bb$
    // Phase 2
    append all nodes $n$ to $worklist$ that have a predecessor that is not their IDOM
    while ($worklist$ not empty)
        $bb = worklist.removeFirst()$
        find $bb$'s predecessor $n_1$ in the dominator tree that dominates
            all of $bb$'s control-flow predecessors (stops at the entry node)
        recalculate $dominates$ for all nodes in the dominator tree between $bb$ and $n_1$
        set $bb$'s IDOM to $n_1$
        append all nodes $n$ to $worklist$ that are dominated by $bb$ and
            have a predecessor that is not their IDOM
```

Fig. 1. Algorithm to decode the stream of basic blocks. The first phase constructs an approximated dominator tree, while the second phase moves nodes upwards in the dominator tree. We use *IDOM* as abbreviation for *immediate dominator*.

dominator tree until it finds a node n_1 that dominates all of n's control-flow predecessors. The decoder then sets n_1 to be the immediate dominator of n and checks for each successor s of n that Theorem 1 is still fulfilled. Each s that does have a predecessor that is *not* dominated by the dominator of s is added to the worklist. This is necessary since moving n around may have changed the information for s.

4.2 Encoding Dominator Information

The encoder is responsible to facilitate the decoding process described above. To ensure that a node is placed *below* its predecessors in the approximation, it must be encoded *before* these. Following Theorem 1 this also ensures that each node is placed below its immediate dominator.

To construct the encoding, the encoder takes a subgraph of the control-flow graph as input. This subgraph contains exactly the control-flow edges (n_1, n_2) for which n_1 does *not* dominate n_2. Figure 2 shows the control-flow graph and dominator tree for an example program, as well as the extracted subgraph. Based on the number of incoming and outgoing edges, the encoder determines the initial encoding of basic blocks. Nodes are sorted based on fewer incoming edges and more outgoing edges, and the entry node is always encoded last. The motivation for this heuristic is how the first decoding phase inserts edges into the approximated dominator tree. Whenever an immediate predecessor p of an unhandled node n

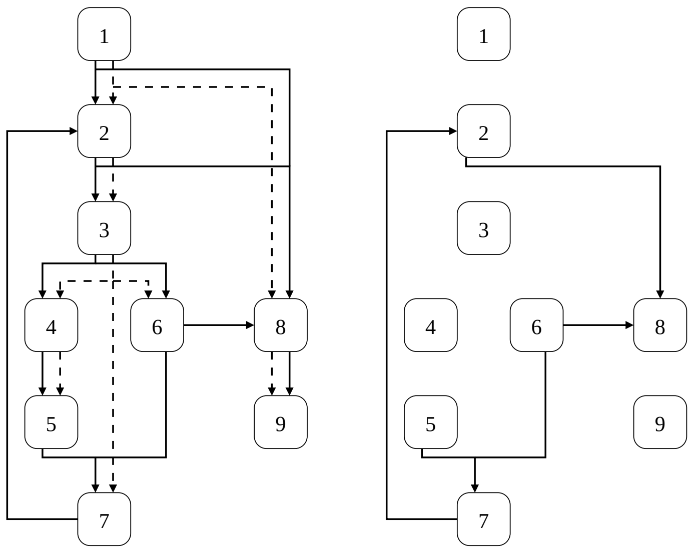

Fig. 2. The left graph is the combined control-flow graph and dominator graph for an example program. The solid edges are control-flow edges, the dashed edges are dominator edges. The right graph shows the subgraph that is actually used to construct the encoding.

```
encode()
    $\forall n, n' \in N$ with $(n, n') \in E \wedge (n, n') \notin$ DOM:
        OUT$(n) + +$; IN$(n') + +$;
    sort $N$ by increasing IN$(n)$ and decreasing OUT$(n)$
    place each node $n$ before its immediate dominator
    $\forall n, n' \in N$ with $(n, n') \in$ DOM $\wedge (n, n') \notin E$:
        place $n'$ before $n''$ with $n'' \in dominates(n) \wedge (n'', n') \in E$
```

Fig. 3. Algorithm to compute the encoding of basic blocks at the code producer. G is a graph with nodes N and edges $E \subseteq N \times N$. DOM $\subset N \times N$ is the immediate dominator relation in G, and $dominates(n)$ gives the set of nodes that are dominated by n. The first phase computes the number of in- and outgoing edges for nodes that are connected by control-flow edges but not by dominator edges. Using these numbers, the nodes are initially sorted. The second phase places all nodes before their immediate dominator. The final phase makes sure that nodes are moved further to the front if there is no control-flow edge between the immediate dominator and the node.

is found, a dominator edge (p, n) is added to the decoded tree. Each control-flow edge (n_1, n_2) that is not a dominator edge will lead to a faulty edge in the approximated dominator tree. We can avoid this insertion by encoding n_1 first, so that the first phase will not have n_2 in *top*. For the example program this gives an initial encoding of

$$[6, 5, 3, 4, 9, 7, 2, 8, 1]$$

However, as described in Section 4.1, the decoder has two properties that impose additional restrictions on the encoding chosen. First, since the decoder at the code consumer can move nodes only upwards, each node must be encoded *before* its immediate dominator. For the initial encoding of the example program, this property

node read	CFG succs	top	inserted edges
7	2	$\emptyset$	
6	$7, 8$	7	$(6, 7)$
5	7	6	
4	5	$5, 6$	$(4, 5)$
3	$4, 6$	$4, 6$	$(3, 4)$, $(3, 6)$
9		3	
8	9	$3, 9$	$(8, 9)$
2	$3, 8$	$3, 8$	$(2, 3)$, $(2, 8)$
1	$2, 8$	2	$(1, 2)$

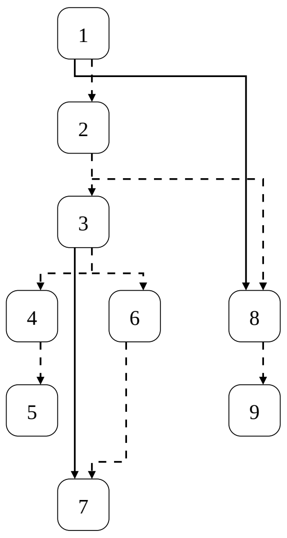

Fig. 4. Decoding of the encoded example program and the computed first approximation to the dominator tree. The solid edges are edges that are missing from the computed dominator tree, that is have not been decoded correctly.

is violated for nodes 7 and 4 (both dominated by 3). Thus, the encoder walks over the initial encoding from back to front and moves nodes towards the front until they are encoded before their dominators. For the example program, this leads to the encoding

$$[6, 5, 7, 4, 3, 9, 2, 8, 1]$$

As shown above, during phase 1 the decoder keeps track of nodes for which no dominator has been decoded yet. All nodes that have not yet been connected when the entry node is reached are assumed to be dominated by that node. While this approach works fine for nodes with a direct edge between the immediate dominator and the node, special care must be taken for nodes where this edge does not exist. In the example graph, $(3, 7)$ is the only pair of nodes that is connected by a dominator tree edge but not by a CFG edge. To make sure that the decoder will place 7 *below* its immediate dominator 3, 7 must be encoded *before* its control-flow predecessors on the shortest path from its dominator. For the example, node 7 must be encoded before node 6. This leads to the final encoding

$$[7, 6, 5, 4, 3, 9, 8, 2, 1]$$

4.3 Performance

This section shows how the decoding of the example program works and reports on our prototype implementation.

As shown above, the encoding for the example program from Figure 2 is

$$[7, 6, 5, 4, 3, 9, 8, 2, 1]$$

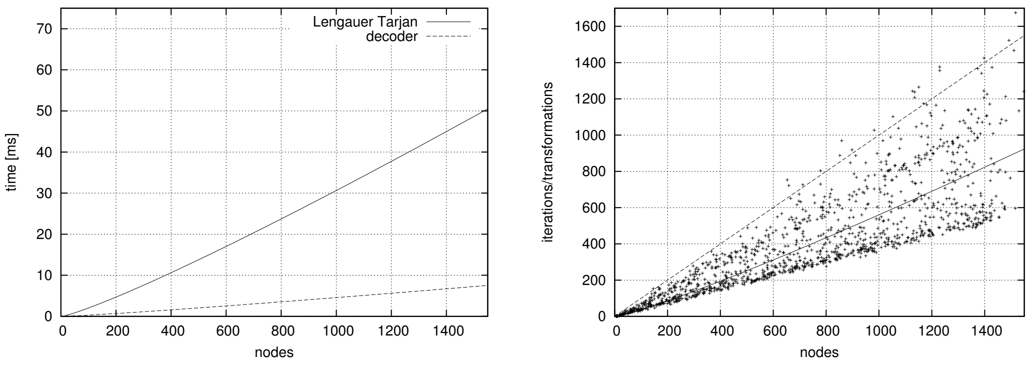

Fig. 5. Comparison of a prototype implementation of the decoding phase to an implementation of the dominator algorithm by Lengauer and Tarjan [12]. The left graph shows the time needed to construct the dominator tree for a randomly constructed graph with x nodes. The right graph shows the number of iterations performed during decoding. Due to the selection heuristic used when putting nodes in the worklist, the algorithm performs a transformation in each iteration. The dashed line is equality. The measurements have been performed on a Pentium 4 with 2.66Ghz and 512MB RAM.

Figure 4 shows the information computed during the first phase of the decoder (Figure 1) and the resulting approximation. As can be seen, all dominator edges but 2 have been decoded correctly.

The second phase starts with the worklist $(2, 7, 8)$, all the nodes that have a predecessor that is not their immediate dominator. For node 2 the only predecessor in the dominator tree is the entry node 1 that dominates all other nodes, so no action is taken. For node 7 the decoder picks node 3, because it dominates nodes 5 and 6, the predecessors of 7. Thus the immediate dominator of 7 is set to 3, reducing the number of incorrect edges to 1. Since 7 has no child nodes in the dominator tree, no new nodes are added to the worklist. For the next node (8) the decoder picks node 1 as immediate dominator, because it dominates its predecessors 1, 2, and 6. The only child node of 8 does not require adding new nodes to the worklist, so phase two finishes with a correct dominator tree.

We have measured the performance of a prototype implementation of this algorithm on randomly generated reducible and irreducible graphs. Figure 5 compares the results in terms of time for the construction of the dominator tree with an implementation of the algorithm from [12]. The right-hand graph shows the number of iterations needed in the second phase of the decoding step, when nodes are moved upwards in the approximated dominator tree. The number of iterations is almost linear in the number of nodes.

4.4 Verification

Even if an aware code consumer recognizes all these code patterns and constructs an SSA-based intermediate representation, it needs to verify that the code is actually in valid SSA form before the IR is safe to be used for code optimization. For

$$\textsc{Dom}(S) = \{S\}$$
$$\textsc{Dom}(n) = \left\{ \bigcap_{p \in preds(n)} \textsc{Dom}(p) \right\} \cup \{n\}$$

Fig. 6. Data-flow equations for the DOM sets for a graph G with nodes N, edges E, and start node S.

this, the code consumer has to verify the following three properties:

- Variables are assigned exactly once as required in SSA form.
- Variables are defined before their first use.
- Variables are used in a type-safe manner.

As type-checking is performed directly on the SSA representation, traditional bytecode verification based on data-flow analysis is obsolete and is no longer performed if the SSA representation is found to be safe.

Interestingly, the code consumer does not need to verify adequate placement of ϕ-instructions. While placing too few or too many ϕ-instructions can lead to programs that do not calculate a meaningful result, they will never result in unsafe code as long as each ϕ-instruction is type-safe (which we do check).

Verifying that each local variable is assigned exactly once is trivial. This is the first action performed by the code consumer once the program has been loaded. Traditional JVMs use an iterative data-flow analysis to verify type-safety and proper variable initialization. While the same approach could be applied to verify that the code is indeed in SSA, it is much more elegant and efficient to perform verification directly in SSA form [9,10]. For this, we have to first recover dominator information from the code, after which the code is traversed in dominator-tree order to type-check uses with their corresponding definition.

The decoder does not guarantee that the resulting graph is the proper dominator tree for the program. A malicious program could be constructed by rearranging basic blocks to make the code consumer believe certain basic blocks are dominating others, while this is actually not the case. Would the code consumer blindly trust the basic-block ordering, it would be vulnerable to such exploits. Fortunately, we can easily verify the obtained dominator information by rephrasing the dominator-tree problem as a data-flow equation.

DOM(b) is defined as a set containing every basic block that dominates *b*. Instead of using iterative data-flow analysis, we initialize each set DOM(b) according to the dominator tree produced by the decoder. If the code was transmitted in a decodable basic-block sequence, the data-flow equations will be satisfied in a single iteration, confirming the decoded dominator tree. If they are not satisfied after one iteration, the code consumer falls back to the standard solution of computing the dominator tree from scratch using approaches like [3,12].

Once we have recovered the dominator tree, we traverse the code in dominator-tree order and and determine the distinct type for each variable definition. This type is then matched to the respective uses. As variables are assigned exactly once and we visit dominating blocks first, definitions will automatically appear before their uses. If the decoder runs into a missing definition, the code is rejected.

Furthermore, each variable, except for variables defined by ϕ- instructions, has one unique type, as it is assigned exactly once (SSA). This greatly simplifies type checking. While traversing the code, the type of each definition is recorded and for each use this table is consulted to verify that definition and use have compatible types. As we have discussed above, the program does not contain any more data-flow instructions such as `dup`. The remaining core instructions are self-typed, i.e. the expected types of any consumed operands and the types of any produced values are known statically. The only exception from this are ϕ-instructions, for which the return type has to be formed through type inference over their operands.

A more detailed description of SSA-based bytecode verification can be found in [9,10].

5 Related Work

Finding the dominator tree in a control-flow graph is an essential problem for program analysis and transformation, e.g. Static Single Assignment form construction. While multiple algorithms with differing average and worst-case complexity have been proposed [15,12,3,4], all these algorithms work in a single phase on the control-flow graph. Based on these prior works we have split the construction into the two phases described in this paper.

Static Single Assignment form is used as intermediate representation in most of the current high-performance JIT-based virtual machines. However, SSA is rarely used as transport format.

SafeTSA [1], an inherently safe mobile code representation format, uses SSA as encoding and transport format. This has the additional advantage of eliminating the need for verification as mobile code is stored in a self-consistent format that cannot represent anything but well-formed and well-typed programs. This comes at the price of abandoning the existing Java class-file format, which is not always acceptable. Our approach and SafeTSA have in common that they both make the code available to the JIT in SSA-form, which can be used to speed up code generation.

An example for using SSA-based representations for compilation of bytecode is Marmot [6], a research platform for studying the implementation of high-level programming languages. The main difference to our work is that Marmot, like many other similar frameworks, focuses only at the code consumer side and does not generate code-producer side hinting such as program reordering.

Annotating mobile code with proofs that can be checked by a code consumer is a well explored area. *Proof-carrying code* (PCC) [17,16] addresses this problem by relieving the code consumer of the burden to verify the code. Instead, the code producer computes a verification condition based on a public safety policy and proves it to be true for the program. This proof is shipped to the code consumer along with the code. Upon receipt, the code consumer recomputes the verification condition and can then check whether the attached proof indeed establishes the verification condition as claimed by the code producer. Just like using SSA as a transport format, shipping additional proof information along with the code requires the abandonment of the original Java bytecode format.

The split verifier approach [22] is very similar to PCC. It annotates the JVML with the fixed-point of the data-flow analysis otherwise performed by the JVM during class loading. For annotated class files the verification is reduced to confirming that the annotation is indeed a valid fixed-point, which can be completed in near-linear time. This idea has been used in our approach for the verification of the computed dominator tree. Otherwise, the split verifier just like PCC requires additional annotations to be shipped with the code.

6 Conclusions and Future Work

We have presented a novel approach to transport SSA information in Java bytecode through *structural annotation*: Instead of introducing new bytecode instructions or adding explicit annotations, the bytecode is rearranged and transformed to allow the code consumer to infer SSA form without actually having to calculated dominator tree and iterated dominance frontiers. The code consumer has to merely run some simple tests to ensure that the encoded information is valid.

The code consumer can not only avoid having to perform these analyses, but can also use a more efficient type-checking method operating directly on the SSA representation. Instead of the worst-case quadratic data-flow analysis, verification runs in linear time and a single sweep over the program.

The presented research is work in progress. While we have implemented a prototype encoder and decoder system, the decoder is not fully integrated with a virtual machine and thus we are not reporting any performance numbers at this point. From previous work [9] we know that SSA-based verification is on average approximately 15% faster than traditional data-flow analysis based verification. Disregarding the time it takes to calculate the dominator tree and iterated dominance frontiers for ϕ-instruction placement, the speedup is 45%. We expect to achieve a similar reduction in verification time, with the added benefit that SSA form is immediately available to the JIT compiler without any additional computation.

As far as future work is concerned, we are currently working on a thorough evaluation of the impact of our approach on legacy VMs. While the code executes on legacy VMs, a certain slowdown can be expected. The most noticeable impact of the transformation is an increase in code size. While this has a significant negative impact on interpretation, we expect it to have only a limited impact on JIT-compiled code. A second side-effect of the proposed structural annotation is a significant increase in number of local variables used per method. In contrast to the code size increase, this seems to affect code generation, in particular when the dynamic compiler uses a simplistic register allocator.

References

[1] Amme, W., N. Dalton, J. von Ronne and M. Franz, *SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form*, in: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001, pp. 137–147, *SIGPLAN Notices*, 36(5), May 2001.

[2] Bilardi, G. and K. Pingali, *Algorithms for Computing the Static Single Assignment Form*, Journal of the ACM (JACM) **50** (2003), pp. 375–425.

[3] Buchsbaum, A. L., H. Kaplan, A. Rogers and J. R. Westbrook, *A new, simpler linear-time dominators algorithm*, ACM Transactions on Programming Languages and Systems **20** (1998), pp. 1265–1296.

[4] Cooper, K. D., T. J. Harvey and K. Kennedy, *A Simple, Fast Dominance Algorithm*, Software Practice & Experience **4** (2001), pp. 1–10.

[5] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM Transactions on Programming Languages and Systems **13** (1991), pp. 451–490.

[6] Fitzgerald, R., T. B. Knoblock, E. Ruf, B. Steensgaard and D. Tarditi, *Marmot: an optimizing compiler for Java*, Software—Practice and Experience **30** (2000), pp. 199–232.

[7] Freund, S. N., *The costs and benefits of java bytecode subroutines*, in: *Proceedings of the Formal Underpinnings of Java Workshop at OOPSLA*, 1998.

[8] Freund, S. N. and J. C. Mitchell, *Specification and verification of java bytecode subroutines and exceptions*, Technical Report CS-TN-99-91, Standford University (1999).

[9] Gal, A., C. W. Probst and M. Franz, *Proofing: Efficient SSA-based Java Verification*, Technical Report 04-10, University of California, Irvine, School of Information and Computer Science (2004).

[10] Gal, A., C. W. Probst and M. Franz, *Integrated Java Bytecode Verification*, in: *Proceedings of the First International Workshop on Abstract Interpretation of Object Oriented Languages*, 2005.

[11] Hagiya, M. and A. Tozawa, *On a new method for dataflow analysis of Java Virtual Machine Subroutines*, in: G. Levi, editor, *Static Analysis, 5th International Symposium, SAS'98* (1998), pp. 17–32.

[12] Lengauer, T. and R. E. Tarjan, *A fast algorithm for finding dominators in a flowgraph*, Transactions of Programming Languages and Systems (TOPLAS) **1** (1979), pp. 121–141.

[13] Leroy, X., *Java Bytecode Verification: Algorithms and Formalizations*, Journal of Automated Reasoning **30** (2003), pp. 235–269.

[14] Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley, 1996.

[15] Lowry, E. and C. W. Medlock, *Object Code Optimization*, Communications of the ACM **12** (1971), pp. 13–22.

[16] Necula, G. C., *Proof-carrying code*, in: *Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Paris, France, 1997.

[17] Necula, G. C. and P. Lee, *Safe Kernel Extensions Without Run-Time Checking*, in: USENIX, editor, *Proceedings of the 2^{nd} Symposium on Operating Systems Design and Implementation (OSDI '96)* (1996), pp. 229–243.

[18] O'Callahan, R., *A Simple, Comprehensive Type System for Java Bytecode Subroutines*, in: *Proceedings of the 26^{th} ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, San Antonio, Texas, 1999, pp. 70–78.

[19] Shaylor, N., *A Just-in-Time Compiler for Memory-Constrained Low-Power Devices*, in: *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (JVM-02)* (2002), pp. 119–126.

[20] Sreedhar, V. C. and G. R. Gao, *A Linear Time Algorithm for Placing ϕ-nodes*, in: *Proceedings of the 22^{nd} ACM SIGPLAN Symposium on Principles of Programming Languages*, San Francisco, California, 1995, pp. 62–73.

[21] Stata, R. and M. Abadi, *A Type System for Java Bytecode Subroutines*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 90–137.

[22] Sun Microsystems, *JSR-000139 Connected Limited Device Configuration 1.1*. `http://www.jcp.org/en/jsr/detail?id=139`.

[23] Yellin, F., *Low level security in Java*, in: O'Reilly and Associates and Web Consortium (W3C), editors, *World Wide Web Journal: The Fourth International WWW Conference Proceedings* (1995), pp. 369–380.