ICS

TECHNICAL REPORT

Enabling Efficient Program Analysis for Dynamic Optimization of a Family of Safe Mobile Code Formats

Ning Wang *Niall Dalton* *Michael Franz*

Technical Report 02-24
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

November 2002

Information and Computer Science
University of California, Irvine

Enabling Efficient Program Analysis for Dynamic Optimization of a Family of Safe Mobile Code Formats

Ning Wang
wangn@uci.edu

Michael Franz
franz@uci.edu

Niall Dalton
ndalton@ics.uci.edu

Department of Information and Computer Science
University of California
Irvine, CA 92697-3425

ABSTRACT

Modern and likely future architectures require compilers to perform extensive restructuring of programs during optimization. We have been building a system in which JVM bytecode is compiled off-line into an alternative, enhanced mobile-code format. This alternative format is still fully target-machine independent but can be more easily verified and compiled into native code. In particular, our approach permits shifting of analyses and optimizations to the code producer that, because of the necessity to perform bytecode verification, could only occur on the code consumer if JVM bytecode were used. Our approach naturally encompasses irreducible control flow, which can result from the use of bytecode optimizers, obfuscators and compilers for source languages other than Java. Our techniques are applicable beyond JVM bytecode.

Although some optimizations can be moved to the code producer, we believe that it will still be unavoidable to perform some restructuring optimizations on the target machine. For example, loop transformations, code scheduling, and parallelization are vital to achieve high performance on EPIC and multithreaded architectures.

In this paper, we introduce the Augmented Dominator Tree (ADT) as a candidate mobile code format enabling efficient program analysis. The ADT may be quickly validated on the target machine; we give an $O(E + V)$ algorithm for this. However, our algorithm not only verifies the validity of the data structure, but also reconstructs the control flow graph, and computes the dominance frontier. Furthermore, we show how to quickly compute the post-dominator tree, using a more efficient variant of Cooper, Harvey, and Kennedy's iterative dominator algorithm. We intend the ADT to form the basis of simple and efficient algorithms for performing sophisticated program analysis in a just-in-time compiler for high-performance architectures.

1. INTRODUCTION

When mobile code is transported using Java bytecode, the code needs to be *verified* by the code consumer before it can be executed. This verification step requires a considerable effort that imposes a delay before execution can commence.

The need for verification also severely limits producer-side optimization of the bytecode, as many of the optimizations that compilers typically employ would not be indistinguishable from malicious modifications at the code consumer's site.

Just-in-time compilation (from bytecode to native code) imposes an additional delay, in addition to the verification delay. In an interactive context (a user is waiting for the program to start), these delays can become intolerable. As a consequence, just-in-time compilers often don't use the best available algorithms, but instead employ algorithms that execute quickly, at the expense of code quality (e.g., linear-scan register allocation rather than graph-coloring).

We have been exploring alternative mobile-code representations that overcome some of the limitations of the Java Virtual Machine format. In particular, our aim has been to reduce the verification effort, enabling more time to be spent on code generation instead, and to transport code in a format immediately suitable for further optimizing compilation on the target machine, leaving even more time for code generation since less pre-processing is necessary for "lifting" the transport format into a suitable IR for optimization.

Our SafeTSA format (introduced at PLDI 2000 [1]) allows shifting some of the optimizer's workload from the code consumer to the code producer. Based on Static Single Assignment (SSA) format, it enables use of more sophisticated optimizations on the client machine. For example, Budlimic, Cooper, et al.'s recent fast copy copy coalescing and live-range identification method [] depends on certain properties of SSA. The methods can potentially make graph-coloring register allocation fast enough for widespread use in just-in-time compilers; basing a compiler on SafeTSA allows an even faster implementation of their method.

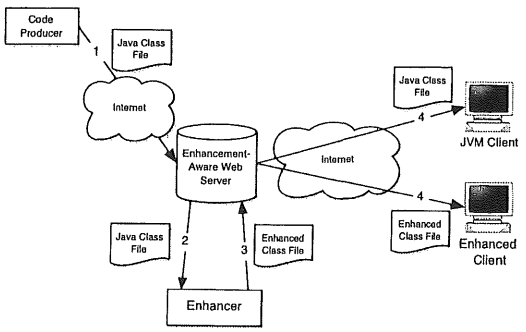

Figure 1: Flow of Class Files Through the System

In the meantime, we have been building a system (Figure 1) in which class files containing ordinary Java bytecode and "enhanced" class files containing our alternative intermediate representation coexist side-by-side. "Enhancement" is offered as an off-line process by certain web servers in this system, the beneficiaries are certain "enhanced" clients.

Several flows of class files are identified in Figure 1: all class files originate in the standard Java bytecode format and are placed on a server for hosting (1). Some of the hosting servers will provide an enhancer that will input an ordinary Java bytecode file (2) and generate an enhanced class file from it (3). Client computers negotiate with every server they connect to; if an enhancement-aware server detects an enhanced client, it will send it an enhanced class file if one is available; otherwise, it will send the standard Java bytecode file (4).

An enhanced client, on the other hand, can process both regular Java bytecode files as well as enhanced class files. This enables it to communicate with all servers on the Internet. If it is communicating with an enhancement-unaware web server, or if no enhanced class file is available on an enhancement-aware server, then it will fall back onto the classic Java bytecode format. If an enhanced class file is available, then it will be used instead, resulting in a higher level of performance.

In our prototype implementation of an enhanced client, the two formats (JVM and SafeTSA) actually share the identical low-level code generator that translates from a low-level intermediate representation (LIR) to the final native instruction stream, resulting in comparable final code quality for the two formats when compilation time is unbounded.

The key point, however, is that compilation time in dynamic compilation environments is hardly ever unbounded. In this situation, the enhanced class file format has a substantial advantage, because fewer steps and less complex operations are needed (Figure 2 (b)) to verify and preprocess it into the LIR (a variant of Static Single Assignment form). The time that has thus been saved (essentially by performing analyses at the code producer's site and transmitting the results within the enhanced mobile-code format in a tamper-proof manner) can then be expended on high-quality code optimization.

We now propose a method to enable more highly optimizing compilation on the client.

2. ENABLING HIGHLY OPTIMIZING COMPILATION ON THE CLIENT

Advances in computer architecture have led to a need for increasingly aggressive program restructuring by optimizing compilers. This is necessary for both single-threaded and multi-threaded processors. We believe it will become increasingly common for just-in-time compilers to perform some of these restructuring optimizations, as we tackle, for example, dynamic compilation for a multi-threaded EPIC processor.

Figure 2 (c) shows a new compilation path for such a just-in-time compiler. Assuming we could quickly build the required data structures (the "Fast Expand" step), we can enable aggressive analysis/optimization and code generation. By building a dedicated code generator around the format, we can avoid the passes that perform representation "lifting" and analysis, and thereby minimize overhead. In this pipeline, aggressive optimizations can be performed at each stage.

In this paper, we provide a solution to the problem of quickly building the required data structures in a (both time and space) efficient manner. Though not addressed in this paper, referential integrity and type safety of the code can be achieved using the same techniques as in SafeTSA.

After defining terminology and notation, we introduce the Augmented Dominator Tree (ADT). The ADT is essentially a serialized version of DJ-graphs[10] with more constraints. It is easy to verify ADT but not DJ-graphs. DJ-graphs were originally designed to enable the design of simple and efficient algorithms for sophisticated program analysis in static (or "ahead-of-time") compilers[11]. Our ADT shares the benefits of the DJ-graph, but is suitable for use as a mobile code format, and as a basis for optimization in a just-in-time compiler.

In section 4.2.2 we present an algorithm to check that the ADT is valid, and reconstruct the control flow graph in a single linear pass. In section 5.1, we modify the algorithm to simultaneously build the dominance frontier. A second pass can quickly build the post-dominator tree; we give a more efficient version of Cooper, Harvey and Kennedy's iterative dominator algorithm in section 5.2 which operates on the ADT.

3. TERMINOLOGY AND NOTATION

In this section, we review all important terminology and notation, which are used in to prove or describe our approach.

Definition 1. A **control flow graph (CFG)** $G(V_G, E_G)$ is a directed graph in which V_G is the set of nodes, and an edge $(x, y) \in E_G$ represents a possible flow of control from x to y. There are two distinguished nodes: $start$ and $end \in V_G$. $start$ has no predecessor and every node is reachable from it. end has no successors and is reachable from every node. The **edge** $(start, end) \in E_G$ **indicates that the surrounding program might not execute** G **at all**. Each node $x \in V_G$ has no more than **2** successors or out edges.

The ADT verification algorithm developed in this paper dependens on this restricted CFG defintion. We will deal with more generic multiple out edges case in section 4.2.4. All $G(V, E_G)$ mentioned in this paper refer to this defintion.

Definition 2. For nodes x and $y \in V_G$, if x appears on every path from $start$ to y, then x **dominates** y. Every node dominates itself. If x appears on every path from y to end, then x **postdominates** y.

Therefore, $start$ dominates all other nodes, and end postdominates all other nodes.

Definition 3. Node x **strictly dominates node** y if x dominates y, and $x \neq y$. We write $x \prec y$ for strictly dominating and $x \preceq y$ for dominating.

Definition 4. Node x is the **immediate dominator** of node y, denoted $x = idom(y)$, if x is the closest strict dominator of y on any path from $start$ to y. Every node of G(V_G, E_G), except $start$, has a unique immediate dominator. Every node has no more than one immediate dominator. We can define **immediate postdominator** in the same way, denoted as $ipdom$.

By the definition 1, $idom(end) = start$ and $ipdom(start) = end$, we say $start$ and end are symmetric.

Definition 5. The edges $\{(idom(x), x) | x \in V_G - \{start\}\}$ form a directed tree rooted at $start$, called the **dominator tree** of G(V_G, E_G), such that x dominates y if and only if x is a proper ancestor of y in the dominator tree. We write the dominator tree as $T(V_T, E_T)$, where $E_T = \{(idom(x), x) | x \in V_G - \{start\}\}$. Since $V_T \equiv V_G$ is always true, the subscript of V can be omitted. We define a function $dtree(G(V, E_G)) = T(V, E_T)$. Similarly, we write T_{post} for the **postdominator tree**.

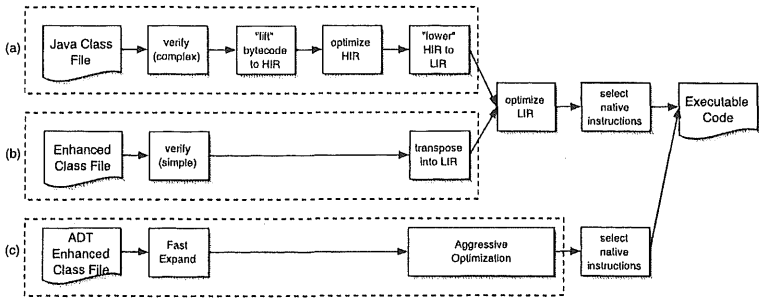

Figure 2: (a) Typical optimizing JVM implementation. (b)Enhanced class files require significantly reduced effort for verification and code generation; when compilation time is constrained, this means that better code can be generated in equal available time. (c) ADT Enhanced class files share the benefits of significantly reduced effort for verification and code generation; they also allow fast reconstruction of the control flow graph, and computation of the dominance frontier. A fast second pass quickly computes the post-dominator tree, thus laying the groundwork for efficient aggressive program analysis and optimization

In this paper, we use **predecessor**, **successor**, **edge** and **path**[1] to refer to nodes in control flow graph, while **parent**, **child**, **ancestor**, **descendant** and **depth** always refer to nodes in a trees (dominator tree here). The equations below define these terms.

$$succ(x)^2 \stackrel{def}{=} \begin{cases} \{y|(x,y) \in E_G\} \wedge 1 \leq |succ(x)| \leq 2 & x \in V - \{end\} \\ \varnothing & x = end \end{cases} \quad (1)$$

$$pred(x)^3 \stackrel{def}{=} \begin{cases} \{y|(y,x) \in E_G\} \wedge |pred(x)| \geq 1 & x \in V - \{start\} \\ \varnothing & x = start \end{cases} \quad (2)$$

$$children(x) \stackrel{def}{=} \begin{cases} \{y|idom(y) = x\} & x \in V - \{end\} \\ \varnothing & x = end \end{cases} \quad (3)$$

$$parent(x) \equiv idom(x) \quad (4)$$

$$ancestor(x)^4 \stackrel{def}{=} \begin{cases} \{y|y \preceq x\} & x \in V - \{start\} \\ \varnothing & x = start \end{cases} \quad (5)$$

$$descendant(x) \stackrel{def}{=} \begin{cases} \{y|x \preceq y\} & x \in V - \{end\} \\ \varnothing & x = end \end{cases} \quad (6)$$

$$depth(x) \stackrel{def}{=} \begin{cases} 0 & x = start \\ 1 + depth(parent(x)) & x \in V - \{start\} \end{cases} \quad (7)$$

Definition 6. Following the terminology from Sreedhar and Gao's DJ-Graphs [9], we call an edge (x, y) in the CFG a **join edge** (or **J-edge**) if $x \not\prec y$. y is termed a **join node**. DJ-graphs are comprised of a program's dominator tree edges (D-edges) and J-edges.

Definition 7. For a node $x \in V_T$, the **dominance frontier** $DF(x)$ is the set of all nodes y of G such that x dominates a predecessor of y but does not strictly dominate y:

$$DF(x) \stackrel{def}{=} \{y|\exists p \in pred(y), x \preceq p \wedge x \not\prec y\} \quad (8)$$

[1]we use **tree path** or **tree edge** to refer to tree.
[2]According to definition 1, $1 \leq |succ(x)| \leq 2$
[3]Based on definition 1, $|pred(x)| \geq 1$
[4]$x \in ancestor(x)$, while $x \notin proper\_ancestor(x)$. The same rule applies to $descendant$.

The $DF(x)$[3] can be obtained by combining the two sets DF_{local} and DF_{up}.

$$DF(x) = DF_{local}(x) \cup \bigcup_{z \in children(x)} DF_{up}(z). \quad (9)$$

Where the DF_{local}(x) is defined by

$$DF_{local}(x) = \{y \in succ(x)|x \not\prec y\}. \quad (10)$$

and DF_{up}(z) is defined by

$$DF_{up}(z) = \{y \in DF(z)|idom(z) \not\prec y\}. \quad (11)$$

Similarly, we write $PDF(x)$ to represent the post dominator frontier of x.

4. THE AUGMENTED DOMINATOR TREE

In this section, we first introduce the relationship between a CFG and its dominator tree and give a definition of "legal" CFGs. Secondly, we give the definition of **augmented dominator tree**(ADT). ADTs are essentially a serialized version of DJ-graph with added constraints on the order of child nodes and $|succ(x)| \leq 2$.

We present an algorithm to verify the validity of an ADT and reconstruct a legal CFG. Finally, we give an improved algorithm to compute the dominator frontier and a simple, fast algorithm to compute postdominance directly from the ADT.

4.1 Graph-Theoretic Characterization of the Dominator Tree

In the following, we explore the relationship between a CFG G and its dominator tree T in order to establish the theoretical support for our algorithm.

LEMMA 1. *If y is a **join node**, then (i) $|pred(y)| \geq 2$ (ii) $\forall x' \in pred(y)$, $depth(x') \geq depth(y)$*

PROOF. Considering assertion (i), by definition 6, there is at least one edge (x, y) in the CFG, and $x \not\prec y$. Suppose y has only one predecessor, then x appears on every path from $start$ to y, contradicting the fact that $x \not\prec y$. Sreedhar and Gao provided a proof of assertion (ii). □

LEMMA 2. *$\forall x \in V$, if $children(x) \neq \varnothing$, then $succ(x) \neq \varnothing$*

PROOF. (By contradiction) Suppose $succ(x) = \varnothing$. Then, there is no path able to go through x, so $children(x) = \varnothing$ contradicting the fact that $children(x) \neq \varnothing$. □

LEMMA 3. *$\forall x \in V$, if $1 \leq |children(x)| \leq 2$, then $children(x) \subseteq succ(x)$. If $y \in succ(x) - children(x)$, then y is a **join node**.*

PROOF. We consider the cases of $|children(x)| = 1$ and $|children(x)| = 2$ separately. By Lemma 2, we have $succ(x) \neq \varnothing$.

1. Let $children(x) = \{y\}$. According to the definition of $succ(x)$ in equation 1, $|succ(x)|$ can have only two values, 1 or 2.

   Let $succ(x) = \{y'\}$. By definition 2, x must appears on every path from $start$ to y, so all paths from x to y must go through y' and y' can not have any predecessor other than x. We can derive that x immediately dominates y'. If $y' \neq y$, then $|children(x)| = 2$, which contradicts the hypothesis that $|children(x)| = 1$. So y' has to be y.

   Let $succ(x) = \{y', y''\}$. Suppose $y \neq y' \wedge y \neq y''$, Then there are at least two paths from x to y, every path must go through either y' or y'' and both y' and y'' can not have any predecessor other than x. We can deduce that x immediately dominates both y' and y''. So $|children(x)| \geq 2$, contradicting the fact that $|children(x)| = 1$. Therefore, either y' or y'' is child of x and the other one is a **join node** by Definition 6.

2. Let $children(x) = \{y, z\}$ and $x = idom(y) = idom(z)$. As with the above proof, we separate the proof into two cases, $|succ(x)| = 1$, and $|succ(x)| = 2$.

   Suppose $succ(x) = \{x'\}$. Since x must appear on every path from $start$ to y or z, if x has only one out $edge(x, x')$, then x' must appear on every path from x to y and z, in other words, x' must appear on every path from $start$ to y or z. We can deduce that x' dominates y and z also, which contradicts the fact $x = idom(y) = idom(z)$. So $succ(x)$ can not have only one node.

   Suppose $succ(x) = \{x', x''\}$. If x has two out edges (x, x') and (x, x''), and x must appear on every path from $start$ to y or z, then both x' and x'' must appear on every path from x to y or z and both x' and x'' can not have predecessors other than x. We can derive that x immediately dominates both x' and x''. If $y \neq x'$ or $z \neq x''$, then $|children(x)| > 2$, contradicting the hypothesis $|children(x)| = 2$. So y, z has to be x', x'' respectively.

□

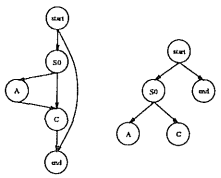

Figure 3: All dominator tree edges are CFG edges.

Therefore, if a dominator tree node x has only one or two children, then those children must appear as successors of x in its CFG, see Figure 3.

LEMMA 4. *$\forall x \in V$, If $|children(x)| \geq 3$, then $|succ(x)|$=2, and $succ(x) \subset children(x)$*

PROOF. By Lemma 2 and definition of succ(x) in equation 1. The only possible cases are $|succ(x)| = 1$ or $|succ(x)| = 2$.

1. Let $|succ(x)| = 1$, and $succ(x) = \{x'\}$. As x must appear on every path from $start$ to $\forall y \in children(x)$, if x has only one out $edge(x, x')$, then x' must appear on every path from x to y. In other words, x' must appear on every path from $start$ to y. We can immediately deduce that x' dominates y, which contradicts the fact that $x = idom(y)$. So $succ(x)$ can not have only one node.

2. Let $|succ(x)| = 2$, and $succ(x) = \{x', x''\}$. Since x must appear on every path from $start$ to $\forall y \in children(x)$, then both x' and x'' must appear on every path from x to y. Also, both x' and x'' can not have predecessors other than x. We can deduce from this that x immediate dominates both x' and x'', so both x' and $x'' \in children(x)$, that is, $succ(x) \subset children(x)$.

□

LEMMA 5. *$\forall x \in V$, if $|children(x)| \geq 3$, $\forall y \in children(x) - succ(x)$ then (i) $|pred(y)| \geq 2$ and (ii) y is a **join node***

PROOF. Considering assertion (i), by Lemma 4, let $succ(x) \cap children(x) = \{x', x''\}$. x has only two out edges (x, x'), (x, x'') and x must appear on every path from $start$ to $y \in children(x) - succ(x)$. Then we can find at least two paths $x' \rightarrow ... \rightarrow y$ and $x'' \rightarrow ... \rightarrow y$. Suppose all paths converge at y' and $y' \neq y$, then y' must appear on every path from x' or x'' to y and y' can not have predecessor other than x' and x''. We can deduce that x dominates y' and y' dominates y, which contradicts that fact that $idom(y) = x$. So, $y' = y$ and $|pred(y)| \geq 2$. Considering assertion (ii), the edge (y', y) is essentially a **J-edge**, so y is a **join node**. □

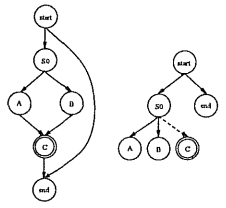

Figure 4: C is a join node.

Lemma 4 and Lemma 5 imply that if a dominator tree node has more than two children, then at least two of its children are its successors in the CFG, and the remaining nodes are **join nodes**, see Figure 4. Intuitively, Lemma 3, Lemma 4 and Lemma 5 imply the mutual dependence between a dominator tree and its original CFG.

Definition 8. The **dominance invariant successors** of node z, denoted by **domis(z)**, is the set of nodes y such that the edge (z, y) can be safely added, if $|succ(z)| \leq 2$, to the CFG without changing the dominance relation.

$$domis(z) \stackrel{def}{=} \{y | dtree(G(V, E_G \cup \{(z, y)\})) \equiv dtree(G(V, E_G))\} \quad (12)$$

By definition, adding any edge from a node to any node in its *domis*, preserves the dominator relation. **We call the resulting CFG "legal" in the sense that it preserves the same dominance relation as before**. All edges from a node z to all nodes y in its *domis* is a superset of z's **J-edges**. Sreedhar and Gao proved

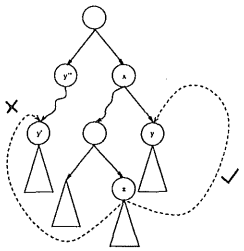

Figure 5: Proving Lemma 6

that the **depth** of the source node of a valid **J-edge** must be larger than or equal to the **depth** of its target node. But the converse is not always true; given $depth(z) \geq depth(y)$, we cannot deduce $y \in domis(z)$. We now show how to find $domis$.

LEMMA 6. $\forall x \in V - \{start\}$

$$domis(z) = \{y | idom(y) \prec z\} \tag{13}$$

PROOF. First we show that if $idom(y)$ strictly dominates z, then $y \in domis(x)$. Let $x = idom(y)$, the edge (z, y) might affect the dominance relation between x and z, y and the dominator subtrees rooted at z, y respectively. We divide the proof in several cases, see Figure 5

1. **Subtree rooted at z will not be changed:** The new edge (z, y) bypasses all $descendant(z)$, which are strictly dominated by z.

2. **Subtree rooted at y will not be changed:** The new edge (z, y) introduces only new incoming paths to y, and y still appears on every path from $start$ to $\forall v \in descendant(y)$.

3. **Subtree rooted at x will not be changed:** Both z and y are strictly dominated by x. As x appears on every path from $start$ to z, so x will also appear on path $start \rightarrow x...z \rightarrow y$. Hence, x still immediate dominates y.

We conclude that the subtree rooted at x is same in all cases and that the dominance relation is preserved.

Secondly, we need to show that no other nodes can be found in $domis(z)$. Suppose $y' \in domis(z)$, $idom(y') \not\prec z$, and let $y'' = idom(y')$. Then we can find a path $start \rightarrow ...z \rightarrow y'$(Figure 5) that bypasses y'', so y'' no longer appears on every path from $start$ to y'. As y'' does not dominate y', this contradicts the assumption that $y' \in domis(z)$. Sreedhar and Gao provide a proof of this in a similar lemma. □

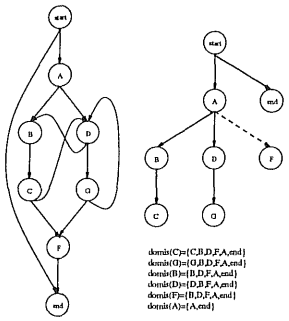

Figure 6: CFG, dominator tree and domis.

Any edge from a node to its $domis$ preserves the dominance relation, Figure 6 shows all $domis$ for a dominator tree.

We define two axillary operations $select2(X)$, which randomly selects 2 elements from a set X, and $remain(X) = X - select2(X)$.

$$select2(X) \cap X \stackrel{def}{=} \begin{cases} select2(X) & |X| > 2 \\ X & |X| \leq 2 \end{cases} \tag{14}$$

$$remain(X) \cap X \stackrel{def}{=} \begin{cases} X - select2(X) & |X| > 2 \\ \varnothing & |X| \leq 2 \end{cases} \tag{15}$$

The purpose of defining these two operations is to show that whatever child nodes are actually successors in the original CFG is not important and will not affect the validity of the ADT and the reconstruction of a legal CFG.

COROLLARY 1. *The dominator tree of any tree is the tree itself, and the tree root node dominates all other tree nodes.*

COROLLARY 2. *Given a CFG $G(V, E_G)$ and its dominator tree $T(V, E_T)$, we construct $G_0(V_0, E_{G_0})$[5] by*

$V_0 \leftarrow \varnothing$
$E_{G_0} \leftarrow \varnothing$
for all $x \in V$ ***do***
  $E_{G_0} \leftarrow E_{G_0} \cup \{(x, y) | y \in select2(children(x))\}$
  $V_0 \leftarrow V_0 \cup \{x\} \cup select2(children(x))$
end for

Let $dtree(G_0(V_0, E_{G_0})) = T(V_0, E_{T_0})$, then $E_{G_0} \subseteq E_T$ and $E_{G_0} \equiv E_{T_0}$

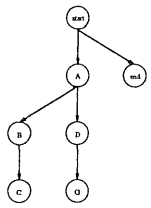

Figure 7: G_0 are computed from the dominator tree in Figure 6

Intuitively, $E_{G_0} \leftarrow E_{G_0} \cup \{(x, y) | y \in select2(children(x))\}$, so $E_{G_0} \subseteq E_T$. Therefore, $G_0(V_0, E_{G_0})$ is a tree. By corollary 1, $E_{G_0} \equiv E_{T_0}$. Figure 7 show a simple example.

THEOREM 1. *Any control flow graph $G_i(V_{G_i}, E_{G_i})$[6], which is created by*

Require: $V_i \leftarrow V_0$
Require: $E_{G_i} \leftarrow E_{G_0}$
for all $z \in V$ ***do***
  $E_{G_i} \leftarrow E_{G_i} \cup \{(z, y) | y \in target(z) \wedge target(z) \subseteq domis(z)\}$
  $V_i \leftarrow V_i \cup target(z)$
end for
Ensure: $\forall z \in V - \{end\}, 1 \leq |succ(z)| \leq 2$
Ensure: $\forall z \in V - \{start\}, |pred(z)| \geq 1$
Ensure: $\forall z \in V, \forall y \in remain(children(z)), |pred(y)| \geq 2$

has the properties:

1. *Let $dtree(G_i(V_i, E_{G_i})) = T_i(V_i, E_{T_i})$, then $E_{T_0} \subseteq E_{T_i}$ and $V_0 \subseteq V_i \subseteq V$.*

2. *$\forall z \in V$, $\exists target(z) \subseteq domis(z)$ such that $dtree(G_i(V_i, E_{G_i})) \equiv T(V, E_T)$.*

[5]Theoretically, this CFG is a tree and does not conform to Definition 1.

[6]This CFG must conform to Definition 1, which is enforced by the **ensure statements**.

PROOF. **(1)**. We show that adding edges $\{(z,y)|y \in domis(z)\}$ will not remove any tree edge from E_{T_0} and might add new tree edges to E_{T_i} or change tree edges of $E_{T_i} - E_{T_0}$. If $y \in V_{G_0}$, then E_{T_0} is preserved by definition 8. If $y \notin V_{G_i}$, then adding (z,y) creates a new dominance relation where z dominates y, hence adding a new edge (z,y) to E_{T_i}. If $y \in V_{G_i} - V_{G_0}$, let $idom(y) = z'$, $(z',y) \in E_{T_i} - E_{T_0}$ and z'' be the lowest common ancestor of z, z', then adding (z,y) will only change the original dominator relation $idom(y) = z'$ to $idom(y) = z''$. Therefore, $E_{T_0} \subseteq E_{T_i}$. Via the constraint $\forall z \in V, \forall y \in remain(children(z)), |pred(y)| \geq 2$, we ensure that $\forall y \notin V_0\ |pred(y)| \geq 2$. **(2)**. Intuitively, If we choose $target(x) = succ(x) - children(x)$ and $target(x) \subseteq domis(x)$ is always true, then we actually get $G_i(V_i, E_{G_i}) \equiv G(V, E_G)$. Hence $dtree(G_i(V_i, E_{G_i})) \equiv T(V, E_T)$. □

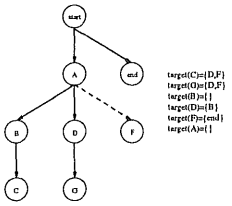

Figure 8: A legal annotation scheme of dominator tree.

Theorem 1 implies that for any dominator tree, there are many CFGs conforming to that dominator tree. In order to get the original CFG, we need annotate each node x with $target(x) = succ(x) - children(x)$. Figure 8 shows an annotation scheme from which we can rebuild the original CFG in Figure 6.

4.2 The Augmented Dominator Tree and Its Properties

In this section, we give the definition of the Augmented Dominator Tree and show how to build and verify the validity of the ADT.

4.2.1 Augmented Dominator Tree

Definition 9. Given a CFG $G(V, E_G)$, its **Augmented Dominator Tree** (ADT) is a tree constructed by the following procedure.

1. Compute the dominator tree $T(V, E_T)$.
2. Build an ordered tree from the dominator tree such that $\forall x \in V - \{end\}$, $\exists y \in succ(x)$ and the post order traversal label of $y \leq$ the post order traversal label of x.[7]
3. Label the tree nodes in post order traversal order.
4. $\forall x \in V$, if $|children(x)| \geq 2$, then label the two nodes which are successors of x, by Lemma 3 and Lemma 4.
5. $\forall x \in V - \{start, end\}$, if $|children(x)| \leq 1$, annotate x with $target(x) = succ(x) - children(x)$.

By applying the above procedure, we can get a safe representation of a CFG as an ADT. Since the valid order of dominator tree is not unique, the ADT is also not unique. Figure 9 shows two valid ADTs. $start$ has only two children, and end is always child of $start$.

[7]This step is not essential, but can speed up the computing of the post-dominator relation during runtime at the client side.

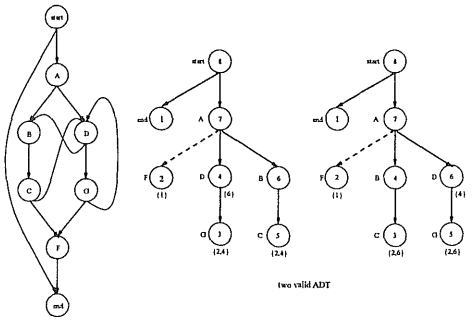

Figure 9: ADTs of the CFG in Figure 6

4.2.2 Algorithm to Verify the Validity of an ADT and to Reconstruct a Legal CFG

We now present an algorithm (Algorithm 1), which is essentially a post-order traversal of the dominator tree, to verify that the $target$ annotation is valid and that the ADT is a valid ordered tree.

We briefly outline the main idea of this algorithm:

Given a ADT, Taking tree node x in post-order traversal order and process x in the following step.

1. If node x has $uncheckedPred(x)$ not-empty, this may happen when some already processed node has an out edge to x, then we need check if this edge conforms to Lemma 6.
2. Add edges from x to all child nodes labeled as successor to E_G, by Lemma 3.
3. Add edges from x to all nodes, which have been found in a post-order traversal, in $target(x)$ and check the validity of target by Lemma 6. If a target y has not been found in a post order traversal then add x to $uncheckedPred(y)$. x will be checked when that y is found.
4. If x is a leaf, then x must have had a target node already identified, this is due to the ordering requirement of the the ADT.

Finally, we check the constraints enforced by Lemma 5, Equation 1 and Equation 2.

Figure 10 is a invalid ADT. This example has three errors. Firstly, $target(B) \nsubseteq domis(B)$, secondly node D is a leaf node, but none of it's target nodes has a post order traversal label less than D's post order traversal label. Lastly F should be a **join node** but it has only one predecessor.

We will now illustrate how our verification algorithm will deal with the illegal ADT in Figure 10. When we traverse to node B, we find target node G has not been found, so we mark B as an unchecked predecessor of G. When we walk to node C, a leaf node, we find that its only target has not yet been found. This contradicts the ordering requirement of the ADT. Assuming we continue the walk through the ADT, the first error will be detected when we travere to node G, as G is not in domis(B). If we were to continue further we will mark F as a **join node** by Lemma 5. When we finish the post order traversal, we next check the constraint enforced by Lemma 5, but we found F has only one predecessor—and this is flagged as an invalid ADT.

Alternatively, we can check the validity of an ADT by building a CFG from the ADT in Figure 10 and computing a dominator tree of the CFG (Figure 11). The rebuilt dominator does not conform to the dominance described in original ADT, so the original ADT is illegal. Hence the reconstructed CFG is also illegal.

Algorithm 1 verifyADT_reconstructCFG:

```
1: for all x ∈ V in post-order traversal order do
2:   for all p ∈ uncheckedPred(x) do
3:     if x ∉ domis(p) then
4:       return false {/*by Lemma 5*/}
5:     end if
6:     E_G ← E_G ∪ {(p, x)}
7:   end for
8:   for all y ∈ children(x) do
9:     if y is labeled as sucessor then
10:      E_G ← E_G ∪ {(x, y)} {/*by Lemma 3, 5*/}
11:    else
12:      mark y as join node {/*by Lemma 5*/}
13:    end if
14:    Boolean atleastOneFound ← false
15:    for all z ∈ target(x) do
16:      mark z as join node
17:      if z hasn't been found in post-order traversal then
18:        uncheckedPred(z) ← uncheckedPred(z) ∪ {x}
19:      else if z ∈ domis(x) then
20:        E_G ← E_G ∪ {(x, z)} {/*by Definition 8*/}
21:        atleastOneFound ← true
22:      else
23:        return false
24:      end if
25:    end for
26:    if !atleastOneFound ∧ y is leaf node then
27:      return false {/*ADT is not proper ordered*/}
28:    end if
29:  end for
30: end for
31: for all u ∈ V do
32:   if u is join node ∧|pred(u)| < 2 then
33:     return false {/*by Lemma 1*/}
34:   end if
35:   if u ≠ end ∧ 1 ≰ |succ(u)| ≰ 2 then
36:     return false {/*by Equation 1*/}
37:   end if
38:   if u ≠ start ∧ |pred(u)| = 0 then
39:     return false {/*by Equation 2*/}
40:   end if
41: end for
42: return true
```

4.2.3 Complexity Analysis

The post-order traversal of the ADT takes $\mathbf{O}(E_T)$. For each node, as we need check the edge annotated by $target$, the total is $E_G - E_T$. The checking(**line2-3** in Algorithm 1) for $x \in domis(p)$ where $p \in uncheckedPred(x)$ can be done in $\mathbf{O}(1)$, because the check is equvilent to check for $p \in descendant(parent(x))$, this is simplely a range test because of the post traversal order labeling all tree node. Figure 12 shows any edge from B or C subtree's child p to x is valid and $min \leq$ postorderlabel of $p \leq max - 1$. We notes that the $doms(parent(x)) \subset domis(x)$, so the check(**line 19** in Algorithm 1) for $z \in domis(x)$ can be done by constructing a bitmap — $domis(x) = \{x' | x'$ is sibling of x and x' has been found in post order traversal $\} \cup domis(parent(x))$. The total time of constructation and deconstruction of $domis(x)$ for all node x is $\mathbf{O}(V)$. After a post-order traversal of all nodes, we also need to check(**line 31-41**) some constraints in $\mathbf{O}(V)$. Thus, the overall time complexity is $\mathbf{O}(E_G + V)$.

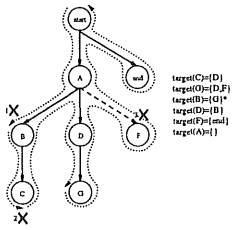

Figure 10: An invalid ADT

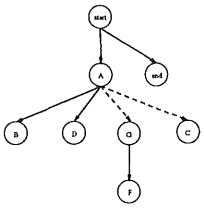

Figure 11: The dominator tree of illegal ADT in Figure 10

4.2.4 Generic CFGs

Irreducible CFGs

Any CFG, reducible or irreducible, has a dominator tree. So the modeling of irreducible graph is a natural result of our ADT approach, no special process is needed to deal with irreducible graph.

Nodes with Multiple Out Edges

Since our definition of the CFG limits the maximum out edges of each node to **2**, an obvious way of modeling nodes with multiple out edges, such as switch statements, is to translate them to cascaded two-way exit nodes (such as an if-else construct). Figure 13 shows one example of an unstructured CFG with multiple out edge nodes. It might appear that our modeling method will lose the information carried by multiple out edge nodes, which would affect code generation on the client. This is not in reality true, as we can easily recover such information.

COROLLARY 3. *If a sequence of cascaded two-way exit nodes is created by decomposing a multiple-way exit node, then these two-way exit nodes immediately dominate each other and the ordering is not essential.*

This is true because if these if-else statements are created by transforming a switch node, then no other edge will jump into this set of if-else nodes. Hence they immediately dominate each other and form a path in the dominator tree. See Figure 13.

We can use a simple algorithm to detect this set of two-way exit nodes and merge them during verification of the ADT. The criteria are as follows:

1. If a group of nodes immediately dominate each other and form a tree path in the dominator tree.
2. If each node has only one instruction <**beq, Rx,** $const_i$, $label_i$ >[8]

These criteria ensure that it is safe to merge this group of nodes to a multiple out edge node which contains only one instruction <**mbeq, Rx,** $const_1$, $label_1$, ..., $const_i$, $label_i$, ...>.

[8]**beq** is branch to $label_i$ when **Rx** == $const_i$. **Rx** is register, $const_i$ is constant, **mbeq** is multiple branch.

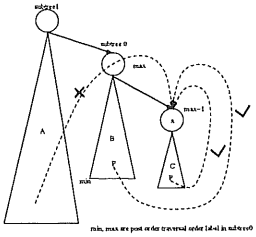

Figure 12: Checking the validity of $uncheckedPred$.

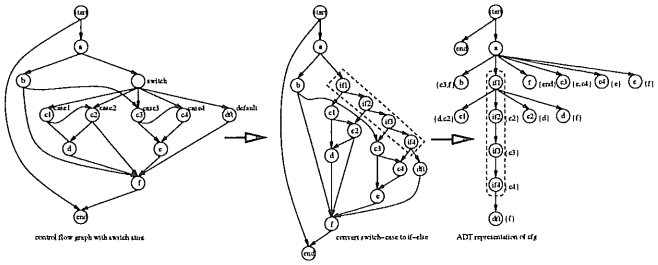

Figure 13: ADT representation of switch statement control flow graph

5. ENABLING HIGHLY OPTIMIZING COMPILATION

Control flow relations, such as dominance and post-dominance frontiers, control dependences etc., have been beneficially used in many statically highly optimizing compilers[5]. An example of a necessary analysis for highly optimizing compilation is loop identification, which allows application of a variety of loop transformations. As we stated earlier, ADTs may be considered a variety of DJ-graph with added constraints. Therefore, it is no surprise that Sreedhar and Gao's bottom-up loop identification algorithm is applicable. We recall that their algorithm has the advantage of being able to identify reducible loops even within irreducible loops of a flowgraph. We also expect that their incremental algorithms, which depend on properties of the DJ-graph, can be adapted for use with the ADT. This will also lead to incremental updating of dominators and dominance frontiers of arbitrary flow graphs and enable use of incremental data flow analysis.

We now present how to compute the dominance frontier and post-dominator trees from the ADT.

5.1 Computing the Dominance Frontier

LEMMA 7. *Given a $ADT(V, E_T)$, $\forall x \in V$*

$$DF_{local}(x) = target(x) \tag{16}$$

PROOF. This is straight forward. Note that $target(x) = succ(x) - children(x) = \{y | y \in succ(x) \wedge x \not\prec y\}$ and the definition of DF_{local} in equation 10. □

LEMMA 8. *Given an $ADT(V, E_T)$, $\forall x \in V$ and $\forall z \in children(x)$*

$$DF_{up}(z) = \{y | y \in DF(z) \wedge depth(y) \leq depth(x)\} \tag{17}$$

PROOF. Sreedhar and Gao provide a proof of this lemma. □

We just need to slightly modify our previous algorithm, without increasing the run time complexity, to compute the dominance frontier while validating the ADT and reconstructing a legal CFG.

5.2 Computing Post-Dominance

Before presenting our algorithm to compute post-dominance, we briefly introduce Cooper, Harvey and Kennedy's improvement[2] on the traditional **iterative dominator algorithm**. Recasting it in our terms, we development an improvement to their algorithm.

Cooper et al. gave a novel representation of the dominators of node x.

$$dom(x) = \{start\} \cup ... \{\} \cup ... \{idom(idom(x))\} \cup \{idom(x)\} \cup \{x\}$$

This sequence actually represents a tree path in the dominator tree. We can reasonably assume the dominator tree exists, before we actually build this tree.

By using the above representation, finding

$$dom(x) = (\bigcap_{p \in pred(x)} dom(p)) \cup \{x\}$$

is equivalent to finding a tree path from the **lowest common ancestor (LCA)** of the predecessors of x to $start$ in the assumed dominator tree.

The essence of the **iterative dominator algorithm** is to build a forest of dominator subtrees, merge those forests to a single dominator tree and find a **stable tree path** from each node to $start$. We first define **stable tree paths** and **stable nodes**.

Definition 10. A **stable node** is a node which has a **stable tree path** to its parent in the assumed dominator tree.

Definition 11. $\forall x \in V$,

1. If $|pred(y)| = 1$, and $pred(y) = \{x\}$, then $idom(y) = x$, and the tree path $y \rightarrow x$[9] is **stable**. We term this a **stable minimum tree path** and y, x forms a **stable minimum dominator subtree**. y is **stable node**, but x might not be **stable node**.

2. If $|pred(y)| > 1$ and $\forall x \in pred(y)$, $x \stackrel{+}{\rightarrow} start$ is **stable**, then $y \stackrel{+}{\rightarrow} start$ is **stable**. Both x and y are **stable nodes**.

We note that the **stable tree path** is iteration invariant. Based on this observation, we can improve Cooper's iterative dominator algorithm by only iterating over **unstable tree paths**. This may not change the complexity of the **iterative dominator algorithm**, but it reduces the iteration over all graph nodes to only possibly affected (**unstable**) graph nodes. We do not repeat Cooper's algorithm here, but this improvement will be used in our modified version which is used to compute the post-dominator tree directly from the ADT.

LEMMA 9. *$\forall x \in V - \{end\}$. if $end \in target(x)$, then end immediately post-dom x.*

PROOF. end is reachable from every node, therefore it post-dominates every node. If there is an edge from $x \in V - \{end\}$ to end, then end immediately post-dominates x. □

LEMMA 10. *$\forall x \in V$ if $|children(x)| + |target(x)| = 1$, $y \in children(x)$ or $y \in target(x)$, then y immediately post-dominates x, $ipdom(x) = y$.*

PROOF. Intuitively, end is reachable from every node including x, and x has only one successor y. Therefore, every path from x to end must go through y, so y immediately post-dominates x. □

[9] $\rightarrow$ represents only one directly connected tree edge, while $\stackrel{+}{\rightarrow}$ means at least one directly connected tree edge.

Lemma 9, and Lemma 10 help us to find all **stable minimum post dominator subtrees** for the post-dominator tree.

COROLLARY 4. *The post order traversal of the ADT is a partial order of the post-order of the RCFG.*

As each parent has a path to its children in the CFG, its children are the predecessors of its parent in the RCFG. Hence, we only need order the child nodes such that every tree leaf node (always treating *end* as processed) has at least one successor that is processed, which is exactly the ordering of the ADT. This is checked by $atleastOneFound$ in Algorithm 1. We can thus guarantee each node has at least one successor that is processed. We have essentially the same guarantee as the reverse post-order of RCFG and do not need to compute the reverse post-order of the RCFG again during client-side processing.

Our **iterative dominator algorithm**(Algorithm 2) has two steps:

1. First, we compute a post-order traversal of the ADT to build an array which essentially contains a forest of **stable minimum post dominator subtrees**, the forests are linked by **unstable tree paths** to form a unstable post dominator tree.

2. Next, iterate over all **unstable nodes** and update **ipdom** to the **LCA** of its predecessors in current post dominator tree until all paths are stable or no change, that is, we obtain a tree which contains only stable tree paths.

Figure 14 show a example how to compute a post-dominator tree directly from the ADT. The left side of the figure shows the input ADT, the middle of figure shows the contents of **node, ipdom** and **stable** array after each stage of the algorithm. All array are indexed by node's post order traversal label and **node** array contains the symbolic name of each node. The right side of the figure shows the contents of **ipdom** and **stable** array in graphic. Dashed lines represents **unstable tree paths** or **unstable nodes**, while solid lines represent **stable tree paths** or **stable nodes**. The *end* node is always a **stable node**.

Post order traversal of ADT: load information to **node, ipdom** and **stable** array respectively as well as finds all **stable minimum post dominator subtrees**. $\{c, b\}$ and $\{end, F, start\}$ are the only two **stable minimum post dominator subtrees**, they are linked by **unstable tree path** with other **unstable node** to form **unstable post dominator tree**.

Iteration 1: iterate over all **unstable nodes**{G,D,C,A} and update their **ipdom** by the **LCA** of its successors in current unstable post dominator tree. For example, at node D in this iteration, D has sucessors {G, B}, the $LCA(B, G) = F$ so $ipdom(D)$ is changed to F.

Iteration 2: iterate over all **unstable nodes**{G,D,C,A} and update their **ipdom**, but find no one changed. Then done.

The first step—a post order traversal of ADT—can be merged with the verification of the ADT. Since we limit each iteration to unstable nodes and propagate stable paths, each iteration costs less time than Cooper et al.'s algorithm. The complexity of the iterative dominator algorithm[2] is $O(N + E \cdot D)$ where D is the size of the largest Dom set. We can save the time to do the DFS, $O(E)$, and one iteration that is merged with the verification phase. Such savings are important for practical implementations of dynamic compilers. Cooper et al.'s experimental results show that the simple iterative dominator algorithm is 2.5 times faster than the classic Lengauer-Tarjan algorithm[6] on real programs.

After obtaining the post-dominator tree, we can use Cooper et al.'s simple dominance frontier algorithm to quickly compute the

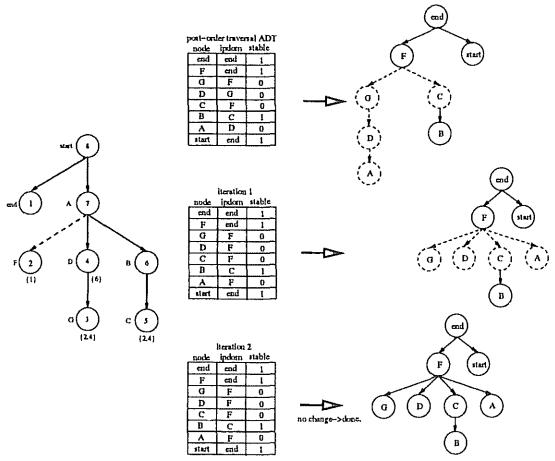

Figure 14: Computing the post-dominator tree

post-dominator frontier; an important data structure for automatic parallelization.

We also note that the post-dominator tree and post-dominator frontier (Figure 15) is a "no-caching" Augmented Postdominator Tree (APT)[7, 8].

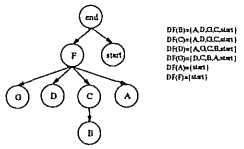

Figure 15: No-Caching APT

Pingali and Bilardi introduced the APT to the allow efficient querying of control dependence relations during compilation. In particular, they concentrate on the following dependence sets:

1. $cd(e)$: which statements are control dependent on the control flow edge e

2. $conds(w)$: which edges statement w is control dependent on

3. $cdequiv(w)$: which statements have the same control dependences as statement w

Analysis and optimizations such as loop transformations and scheduling can viewed as requiring the computation of these sets[4]. As we are aiming to enable advanced optimizations for recent architectures, such as EPIC, in a just-in-time setting, these sets are invaluable. For example, using $cdequiv(w)$ we can identify groups of basic blocks suitable for region scheduling.

In this case "no-caching" means that no information is stored at internal nodes to speed processing of control dependence relation queries. Caching information at internal nodes can speed processing, but at the added cost of extra storage. Although Sreedhar and Gao did provide procedures to query DJ-Graphs, they did not reduce the complexity of the queries when compared to previous approaches. It may, therefore, be more effective to exploit Pingali and Bilardi's procedures for an APT style data structure. However, this choice will require an experimental investigation in a realistic compiler.

6. SUMMARY AND CONCLUSION

We have introduced the Augmented Dominator Tree (ADT) as a backbone for a new mobile code format. We intend the ADT to enable efficient program analysis for highly optimizing just-in-time compilation for modern architectures. Our algorithm to validate the ADT runs in $O(E + V)$, and not only validates the ADT but reconstructs a legal CFG and the dominance frontier. We also gave an improvement to Cooper, Harvey, and Kennedy's iterative dominator algorithm.

The primary related work to the ADT is Sreedhar and Gao's DJ-Graph. While the DJ-Graph was invented to enable efficient program analysis in a static compiler, our ADT is also a suitable mobile code format with an efficient verification algorithm. We also have presented how to efficiently compute the post-dominator tree for an ADT.

Although we have not stressed the connection between SafeTSA and ADTs, it should be noted that the techniques can be combined to gain the advantages of both. In particular, such a combination (which is the focus of our current implementation effort) enables an end-to-end mobile code pipeline that allows effective program optimization at each stage.

7. REFERENCES

[1] W. Amme, 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'01 Conference on Programming Language Design and Implementation*, pages 137–147. ACM Press, 2001.

[2] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. In *Software-Practice And Experience*, pages 4:1–10. John Wiley and Sons, Ltd., 2001.

[3] R. Cytron, 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(4):451–490, Oct. 1990.

[4] R. Cytron, J. Ferrante, and V. Sarkar. Compact representations for control dependence. *SIGPLAN Notices*, 25(6):337–351, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[5] R. Cytron, J. Ferrante, and V. Sarkar. *Experiences using control dependence in PTRAN*, pages 186–212. The MIT Press, 1990.

[6] T. Lengauer and R.E.Tarjan. A fast algorithm for finding dominators in a flowgraph. In *ACM Trans. Program. Lang. Syst.*, volume 1, pages 115–120, July 1979.

[7] K. Pingali and G. Bilardi. APT: A data structure for optimal control dependence computation. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 32–46, La Jolla, California, 18–21 June 1995.

[8] K. Pingali and G. Bilardi. Optimal control dependence computation and the Roman chariots problem. *ACM Transactions on Programming Languages and Systems*, 19(3):462–491, May 1997.

[9] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 62–73. ACM Press, 1995.

[10] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, Nov. 1996.

[11] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. A new framework for elimination-based data flow analysis using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 20(2):388–435, Mar. 1998.

APPENDIX

A. ALGORITHM TO COMPUTE THE POST-DOMINATOR TREE FROM AN ADT

Algorithm 2 compute post dominator tree

postOrderTraversalADT(root, stable, ipdom)
iterate (ipdom, stable)

postOrderTraversalADT Node x × ArrayOfBoolean *stable* × ArrayOfNode *ipdom*
if $|succ(x)| = 1$ **then**
  $ipdom[x] \leftarrow$ a sucessor of x.
  $stable[x] \leftarrow true$
else if *end* is one of sucessors of x **then**
  $ipdom[x] \leftarrow end$
  $stable[x] \leftarrow true$
else
  $ipdom[x] \leftarrow undefined$
end if
for all $y \in succ(x)$ **do**
  $postOrderTraversalADT(y, stable, ipdom)$
  if $ipdom[x] = undefined \wedge y$ is first processed succ of x **then**
    $ipdom[x] \leftarrow y$
    $stable[x] \leftarrow stable[y]$
  else
    $(ipdom[x], stable[x]) \leftarrow$ **LCA**$(y, ipdom[x])$
  end if
end for
return
iterate ArrayOfNode *ipdom* × Boolean *stable* → PostDominatorTree
while changed **do**
  $changed \leftarrow false$
  for all $x \in$ unstable nodes **do**
    $newidom \leftarrow$ first sucessor of x
    for all other sucessor p of x **do**
      if $ipdom[p] \neq undefined$ **then**
        $(newidom, stable[x]) \leftarrow$ **LCA**$(p, newidom)$
      end if
    end for
    if $ipdom[x] \neq newidom$ **then**
      $ipdom[x] \leftarrow newidom$
      $change \leftarrow true$
    end if
  end for
end while
LCA Node $b1$ × Node $b2 \rightarrow Node \times Boolean$
Node $x1 \leftarrow b1$
Node $x2 \leftarrow b2$
Boolean $s \leftarrow stable[x1] \wedge stable[x2]$
while $x1 \neq x2$ **do**
  while $x1 > x2$ **do**
    $x1 \leftarrow ipdom[x1]$
    $s \leftarrow s \wedge stable[x1]$
  end while
  while $x2 > x1$ **do**
    $x2 \leftarrow ipdom[x2]$
    $s \leftarrow s \wedge stable[x2]$
  end while
end while
return $(x1, s)$