Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 190 (2007) 121–132
www.elsevier.com/locate/entcs

# Computing SSA Form with Matrices

Quan Hoang Nguyen[1]

*The School of Computer Science and Engineering*
*The University of New South Wales*
*Sydney, Australia*

Bernhard Scholz[2]

*The School of Information and Technologies*
*The University of Sydney*
*Sydney, Australia*

**Abstract**

Static Single-Assignment (SSA) form is an efficient intermediate representation used in virtual machines and modern compilers. It provides data flow information that simplifies the implementation of standard program optimisations such as constant propagation, dead code elimination, and partial redundancy elimination. Constructing SSA form involves the computation of graph relations such as dominance, and non-iterated and iterated dominance frontier. Although there exist efficient graph algorithms for these relations, the algorithms are elaborate to implement. In this paper we introduce a new approach to compute the dominance relation, the dominance frontiers, and the iterated dominance frontiers based on Boolean matrix calculus. We implemented our approach in an optimising backend for LCC bytecode and compared its performance with the state-of-the-art approaches. We use the Spec95 benchmark suite for our experimental evaluation.

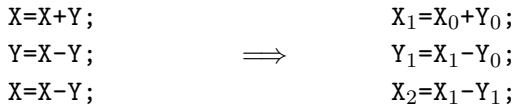
*Keywords:* SSA form, dominance relation, dominance frontier, Boolean matrices

## 1 Introduction

*Static single-assignment* (SSA) form [11] is a sparse intermediate representation that encodes data-flow information. It has been successfully employed as an intermediate representation in several commercial and open source projects such as LLVM [22], IBM's research Java VM called Jikes RVM [17], Sun's HotSpot Server VM [31], and many other projects. For each variable in SSA form there exists only a single assignment. Figure 1 illustrates an example of SSA form for some straight-line code.

[1] Email:quanh@cse.unsw.edu.au
[2] Email: scholz@it.usyd.edu.au


doi:10.1016/j.entcs.2007.02.064




(a) Input Program (b) SSA Form

Fig. 1. SSA Example

Multiple definitions of a variable may reach a confluence point and a special assignment (aka. $\phi$-function node) is inserted to make the variable use after a confluence point unique, i.e., $v_x = \phi(v_{i_1}, v_{i_2}, \ldots, v_{i_k})$ where $v_x$ is the merged value of variable definitions $v_{i_1}, \ldots, v_{i_k}$. The problem of computing the minimum number of insertion points for $\phi$-function nodes was solved by Cytron et al. [10]. However, the insertion of $\phi$-function nodes requires elaborate algorithms to compute

- the dominance relation [27,23,6,2,14],
- the dominance frontier, and
- the iterated dominance frontier [30,25,15,8].

These algorithms are complex, time-consuming and error-prone to implement, and they require sophisticated data structures. To overcome this problem, several other approaches were introduced in the literature. Aycock and Horspool [4] proposed an algorithm for finding placements of $\phi$-function nodes in two phases. The first phase is a crude placement strategy, placing $\phi$-function nodes for all variables at confluence points as proposed in Appel's textbook [3]. In the second phase unnecessary $\phi$-function nodes are eliminated. In [4] it was proved that the approach places a minimal number of $\phi$-function nodes for reducible flowgraphs[3] but not for irreducible graphs. Brandis and Mössenböck [5] introduced an approach that generates SSA form for structured languages whose programs form a subclass of reducible flowgraphs. Sreedhar and Gao introduced a linear-time algorithm for $\phi$-function node insertions [30] based on $DJ$-graphs. This algorithm transforms the input program to SSA form in time $O(E \times V)$ where $E$ is the number of edges, and $V$ is the number of variables.

In this paper we propose a new approach for the generation of SSA form that is based on Boolean matrix calculus. This new approach places a minimal number of $\phi$-function nodes for arbitrary flowgraphs. In contrast to previous work our approach computes the graph relations in terms of simple matrix equations. Although solving the matrix equations have a higher worst-case complexity class than the state-of-the-art approach, our approach is useful for (1) rapid prototyping of a compiler and (2) validating the result of the elaborated algorithms [27,23,6,2,14,30,25,15,8]. The contribution of this paper is as follows:

- We compute the graph relations used to construct SSA form by solving simple matrix equations

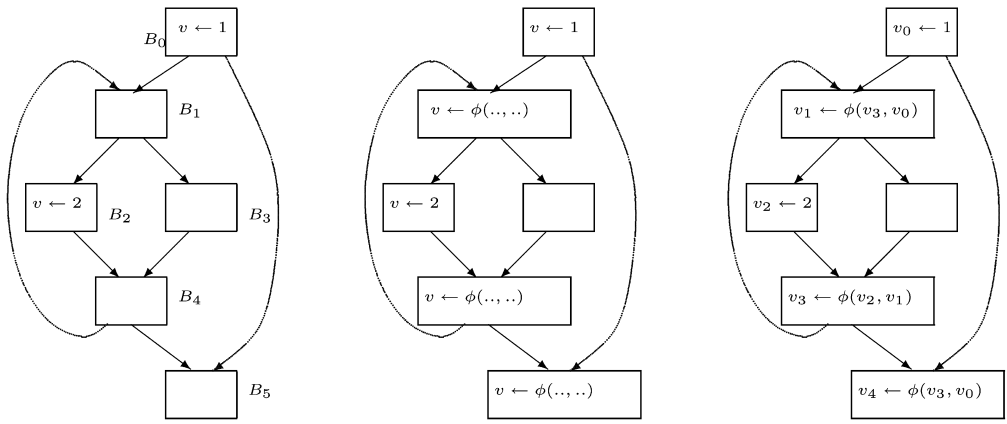
[3] A reducible flowgraph is a control flow graph whose loops have a single entry point.

- We implemented the new approach and compared its performance to the state-of-the-art approaches.

The paper is organised as follows: In Section 2 we motivate our approach. In Section 3 we give the necessary background for our approach. In Section 4 we discuss the proofs of the matrix equation for dominance relation, dominance frontier, and iterated dominance frontier. In Section 5 we present the results of our experiments and in Section 6 we draw our conclusions.

## 2 Motivation

Transforming an input program to SSA form is performed in two steps: In the first step $\phi$-function nodes are inserted at confluence points and in the second step variables are renamed, i.e., subscripts are added to the definitions and uses of variables. Figure 2 shows an example for generating SSA form. The input program is depicted in Figure 2(a) as a flowgraph. In the program there are assignments for variable $v$ in basic block $B_0$ and in basic block $B_2$. The confluence points of the program are basic blocks $B_1$, $B_4$, and $B_5$. In all confluence points a $\phi$-function node is required as shown in Figure 2(b), e.g., in basic block $B_1$ the definition of basic block $B_0$ and the definition of basic block $B_2$ need to be merged. In the second step the definitions and the uses of variable $v$ are renamed as depicted in Figure 2(c).




(a) Input Program (b) Inserting $\phi$-function nodes (c) Renaming Variables

Fig. 2. An example of SSA form transformation

For inserting a minimal number of $\phi$-function nodes Cytron et al. [10] introduced iterated dominance frontiers that compute the insertion points of $\phi$-function nodes for a given set of definitions of a variable. More formally, they map a subset of nodes, i.e., the set of definitions of a variable, to a subset of nodes, i.e., the set of confluence points. In our approach, we use Boolean matrices to express this mapping. A Boolean matrix consists of 0 and 1 values and the underlying algebra is the "and" operation for the multiplication and the "or" operation for the addition.

A subset of nodes $S$ is expressed as Boolean vector $\boldsymbol{s} \in \{0,1\}^n$ of size $n$ where $n$ is the number of nodes in the control flowgraph. Each vector element corresponds to a node and is set to 1 if the node is in the set; otherwise the vector element is set to zero. To compute the iterated dominance frontier for the definitions of a variable $v$ we determine the result of following vector-matrix multiplication:

$$J^+(S_v) = [\boldsymbol{s}_v.\mathbf{J}] \tag{1}$$

where set $S_v$ represents the nodes that contain definitions of variable $v$ and $\boldsymbol{s}_v$ the corresponding vector of $S$. Matrix $\mathbf{J}$ is determined by the topological structure of the control flow graph.

For example, to describe the set of definitions of the example in Figure 2 we use the vector (101000) that corresponds to the definition set $S_v = \{B_0, B_2\}$. To compute the insertion points for the $\phi$-function nodes we compute the vector-matrix multiplication

$$\boldsymbol{s}_v.\mathbf{J} = (101000) \times \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} = (010011)$$

resulting in the node set $\{B_1, B_4, B_5\}$ for placing $\phi$-function nodes. The computation of matrix $\mathbf{J}$ requires several simple matrix equations:

$$\mathbf{J} = \mathbf{D}^+ \tag{2}$$

$$\mathbf{D} = (\mathbf{A}.\mathbf{M} - \mathbf{M})^T \tag{3}$$

$$\mathbf{M} = \neg f^*(\neg \mathbf{M}_0, \mathbf{A}, \neg \mathbf{I}), \tag{4}$$

where matrix $\mathbf{A}$ is the transposed adjacency matrix, matrix $\mathbf{M}_0$ is an initialisation matrix, and $\mathbf{I}$ is the identity matrix. The operation $\mathbf{A}^+$ denotes the transitive closure of a matrix, i.e., $\mathbf{A}^+ = \sum_{k=1}^{\infty} (\mathbf{I} + \mathbf{A})^k$. The *extended transitive closure function* $f^*(S, A, C)$ is defined by the recurrence relation

$$\begin{aligned} X_0 &= S \\ X_{i+1} &= A.X_i \cap C, \ \forall i \geq 0. \end{aligned} \tag{5}$$

The result of the extended transitive closure function is $X_k$ for a $k \geq 0$ such that $X_k$ is equal to $X_{k+1}$, i.e. a fix-point is reached.

Matrix $\mathbf{D}$ is the bit representation of the dominance frontier [10]. I.e., the vector-matrix multiplication $\boldsymbol{s}_v\mathbf{D}$ computes the dominance frontier for subset $S_v$. Matrix $\mathbf{M}$ is the dominance relation represented as a matrix. If element $m_{ij}$ in $\mathbf{M}$ is set to one, node $j$ dominates node $i$; otherwise it is set to zero.

We employ the extended transitive closure function to compute matrix $\mathbf{M}$ for our example in Figure 2. The parameters for the extended transitive closure operator are the matrices $\mathbf{A}$ (that is the transposed adjacency matrix), the negated identity matrix, and matrix $\mathbf{M}_0$ that is a matrix whose elements are set to one except for

the first row. In the first row only the first element in the row is set to one. All other elements in the first row are set to zero. The matrices of the examples in Figure 2 are given below:

$$\mathbf{A} = \begin{pmatrix} 0&0&0&0&0&0\\ 1&0&0&0&1&0\\ 0&1&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&1&1&0&0\\ 1&0&0&0&1&0 \end{pmatrix} \mathbf{M}_0 = \begin{pmatrix} 1&0&0&0&0&0\\ 1&1&1&1&1&1\\ 1&1&1&1&1&1\\ 1&1&1&1&1&1\\ 1&1&1&1&1&1\\ 1&1&1&1&1&1 \end{pmatrix} \mathbf{M} = \begin{pmatrix} 1&0&0&0&0&0\\ 1&1&0&0&0&0\\ 1&1&1&0&0&0\\ 1&1&0&1&0&0\\ 1&1&0&0&1&0\\ 1&0&0&0&0&1 \end{pmatrix}$$

For computing the dominance frontier we use the equation $(\mathbf{A}.\mathbf{M} - \mathbf{M})^T$ resulting in the following matrix:

$$\mathbf{D} = \left[ \begin{pmatrix} 0&0&0&0&0&0\\ 1&0&0&0&1&0\\ 0&1&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&1&1&0&0\\ 1&0&0&0&1&0 \end{pmatrix} \begin{pmatrix} 1&0&0&0&0&0\\ 1&1&0&0&0&0\\ 1&1&1&0&0&0\\ 1&1&0&1&0&0\\ 1&1&0&0&1&0\\ 1&0&0&0&0&1 \end{pmatrix} - \begin{pmatrix} 1&0&0&0&0&0\\ 1&1&0&0&0&0\\ 1&1&1&0&0&0\\ 1&1&0&1&0&0\\ 1&1&0&0&1&0\\ 1&0&0&0&0&1 \end{pmatrix} \right]^T = \begin{pmatrix} 0&0&0&0&0&0\\ 0&0&0&0&0&1\\ 0&0&0&0&1&0\\ 0&0&0&0&1&0\\ 0&1&0&0&0&1\\ 0&0&0&0&0&0 \end{pmatrix}$$

In our new approach we compute iterated dominance frontiers based on matrix calculus. For an implementation a simple binary matrix calculator is needed that is able to compute transitive closures of binary matrices and extended transitive closures. The transitive closure operations can be implemented as simple recurrences until the result stabilises. However, more advanced techniques exist in the literature.

## 3 Background

A *flowgraph* is a directed graph $G = \langle V, E, s\rangle$ where $V$ is the set of nodes representing basic blocks and $E$ is the set of edges. node $s$ is a distinguished *start node*. A path is a sequence of nodes $\langle v_1, \ldots, v_k\rangle$ such that $v_i \rightarrow v_{i+1} \in E$ for all $1 \leq i < k$. In a flowgraph all nodes are reachable, i.e. there is a path from $s$ to every other node in $V$. For each flowgraph node $x$, the set of immediate predecessors and successors of $x$ are defined as $preds(x) = \{n|(n, x) \in E\}$ and $succs(x) = \{n|(x, n) \in E\}$, respectively.

A node $x$ *dominates* a node $y$ (written as $x\, dom\, y$) if every path from $s$ to $y$ includes $x$. The set of dominators $dom(x)$ of a node $x$ is the set of nodes which dominate $x$, i.e. $dom(x) = \{y|y\, dom\, x\}$. For the start node $dom(s) = \{s\}$. For remaining nodes in the flowgraph, the set of dominators is the solution to the following equation system:

$$dom(u) = \{u\} \cup \bigcap_{p \in preds(u)} dom(p), \quad \forall u \in V \setminus \{s\} \tag{6}$$

A node $x$ strictly dominates $y$ (written as $x\, sdom\, y$) if $x$ is not equal to $y$ and $x$ dominates $y$, and the strict dominators of a node are given as $sdom(u) =$

$dom(u) \setminus \{u\}$. The dominance relation has been extensively studied in the past [27,23,6,2,15,14]. There exist some linear-time algorithms in the literature [6,2,15,14]. Although they are asymptotically linear, some of the approaches have high linear constants [2], and are not practical to implement.

The *dominance frontier* [10] of node $x$ (written as $DF(x)$), is the set of all nodes $y$ in the flowgraph such that $x$ dominates an immediate predecessor of $y$ but does not strictly dominate $y$. That is,

$$DF(x) = \{y \mid \exists p \in preds(y) : x\ dom\ p \wedge \neg(x\ sdom\ y)\} \tag{7}$$

Given a set of flowgraph nodes $S \subseteq V$, the *dominance frontier of a set* $S$ is defined as the union of the dominance frontiers of all nodes in $S$, i.e.,

$$DF(S) = \bigcup_{x \in S} DF(x) \tag{8}$$

The *iterated dominance frontier* of $S$ is defined as:

$$J^{+}(S) = \lim_{i \to \infty} DF^{i}(S) \tag{9}$$

where $DF^{1}(S) = DF(S)$ and $DF^{i+1}(S) = DF(S \cup DF^{i}(S))$.

Our approach requires some basic concepts of Boolean matrix calculus. The algebraic properties of Boolean algebras, Boolean vector and Boolean matrices are well-studied [20,21]. The principal idea is that sets are represented as Boolean vectors. Let $U$ be the universal set with $n$ elements and let's assume that there is a fixed order among the elements. Then each set $A \subseteq U$ has a corresponding n-bit vector $\boldsymbol{a}$ in which $a_i$ is set to one if $a_i \in A$, otherwise $a_i$ is zero. Function $[\boldsymbol{a}]$ maps vector $\boldsymbol{a}$ to set $A$.

A *Boolean matrix* of size $m \times n$ is an $m \times n$ matrix over $\mathbb{B}$. Let $\mathbb{B}_{mn}$ denote the set of all $m \times n$ Boolean matrices and let $\mathbb{B}_n$ denote the set of all $n \times n$ Boolean matrices. The $n \times n$ *identity matrix* $\mathbf{I}$ is matrix $\delta_{ij}$ such that $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ if $i \neq j$. The $m \times n$ *universal matrix* $\mathbf{1}$ is the matrix all of whose entries are 1. Each row of an $m \times n$ Boolean matrix is a Boolean vector $v_i \in V_n$. Componentwise matrix operations $+, \cap, \neg, -$ are similar to those of Boolean vectors. Concepts such as transpose, symmetry, and idempotency are the same as in the case of scalar matrices. The transpose of $\mathbf{A} \in \mathbb{B}_n$ (denoted as $\mathbf{A}^T$) is $\mathbf{B} \in \mathbb{B}_n$ such that $b_{ij} = a_{ji}, \forall i, j \in \{1 \ldots n\}$. Let $\mathbf{A}.\mathbf{B} = \mathbf{C}$ where $c_{ij} = \sum_{k=1}^{n} a_{ik}.b_{kj}, \forall i, j \in \{1 \ldots n\}$. $\mathbf{A}^k = \prod_{i=1}^{k} \mathbf{A}$ if $k > 0$ and the identity matrix if $k = 0$. $\mathbf{A}(u)$ is row $u$ of matrix $\mathbf{A}$.

**Lemma 3.1** *Let* $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{B}_n$,

$$\boldsymbol{A}(u).\boldsymbol{B} = \sum_{p \in [\boldsymbol{A}(u)]} \boldsymbol{B}(p). \tag{10}$$

The transitive closure of a Boolean matrix can be seen as a reachability problem

in a graph. The transitive closure of a graph $G = (V, E)$ is a graph $G^* = (V, E^*)$ such that $E^*$ contains an edge $(u, v)$ if there exists a path from $u$ to $v$ in $G$. In the literature, there are lots of studies on transitive closure of a directed graph [32,19,29,16,24,1,18,12,26]. Warshall's algorithm [16] is simple to implement, but exhibits a worst-case runtime of $O(n^3)$. More sophisticated algorithms [26] and fastest matrix multiplication techniques [9] with a squaring technique reduces the asymptotic worst-complexity to $O(n^{2.376} \log n)$.

# 4 Unified Approach

## *4.1 Dominance Relation*

In [27] the computation of $dom(u)$ applies Equation 6 for following recurrences relation

$$\begin{aligned} dom_0(s) &= \{s\} \\ \forall u \in V \setminus \{s\} : dom_0(u) &= V \\ \forall u \in V : dom_{i+1}(u) &= \{u\} \cup \bigcap_{p \in preds(u)} dom_i(p) \end{aligned} \tag{11}$$

where $dom(u) = \lim_{k \to \infty} dom_k(u)$ (for all $u \in V$). Note that after a finite number of steps the recurrence relation stabilises.

**Lemma 4.1** $\boldsymbol{M} = \neg f^*(\neg \boldsymbol{M}_0, \boldsymbol{A}, \neg \boldsymbol{I})$ *where* $\boldsymbol{M}_0 = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 1 & \dots & \dots & 1 \\ \vdots & & & \vdots \\ 1 & \dots & \dots & 1 \end{pmatrix}$.

**Proof.** Let $\boldsymbol{\delta_u} = [\{u\}]$. Then, $dom_{i+1}(u) = [\mathbf{M}_{i+1}(u)]$ where

$$\begin{aligned} \mathbf{M}_{i+1}(u) &= \boldsymbol{\delta_u} + \textstyle\prod_{p \in [\mathbf{A}(u)]} \mathbf{M}_i(p) && \text{(cf. Eqn 11)} \\ \neg\mathbf{M}_{i+1}(u) &= \neg\boldsymbol{\delta_u} . \textstyle\sum_{p \in [\mathbf{A}(u)]} \neg\mathbf{M}_i(p) && \text{(De'Morgan's law)} \\ &= (\mathbf{A}(u).\neg\mathbf{M}_i).\neg\boldsymbol{\delta_u} && \text{(Distributive Law and Equation 10)} \end{aligned}$$

Since $\boldsymbol{\delta_u}$ is row $u$ of identity matrix $\mathbf{I}$, we obtain $\neg\mathbf{M}_{i+1} = (\mathbf{A}.\neg\mathbf{M}_i) \cap \neg\mathbf{I}$ which is the recurrence relation. By translating the boundary condition of fixed-point Equation 11 into matrix form, we represent the dominance relation by the extended fixed-point operation $f^*$ as $\mathbf{M} = \neg f^*(\neg\mathbf{M}_0, \mathbf{A}, \neg\mathbf{I})$. □

## *4.2 Dominance Frontier*

Let $\mathbf{D}$ and $\mathbf{J}$ be the matrices of the non-iterated and iterated dominance frontier relation, i.e. $DF(u) = [\mathbf{D}(u)]$ and $J^+(u) = [\mathbf{J}(u)]$.

**Lemma 4.2** $\boldsymbol{D} = (\boldsymbol{A}.\boldsymbol{M} - \boldsymbol{M})^T$.

**Proof.** By definition, the dominance frontier of a flowgraph node $x$ is $DF(x)$= { $y$ | ($\exists$ $p \in preds(y)$) s.t. ($x$ *dom* $p$ and $x$ ! *sdom* $y$) }, that is equivalent to $S_1 \cap S_2$ where $S_1$= { $y$ | ($\exists$ $p \in preds(y)$) s.t. $x$ *dom* $p$} and $S_2$= {$y$ | $x$ ! *sdom* $y$ }. $x \in dom(p)$ iff $p \in \mathbf{M}^T(x)$. Therefore, $S_1$ is rewritten in vector form as $\boldsymbol{s_1}$= { $y$ | $\mathbf{A}(y).\mathbf{M}^T(x) \neq \mathbf{0}$ } = { $y$ | $\mathbf{M}^T(x).\mathbf{A}(y) \neq \mathbf{0}$ } and $\boldsymbol{s_1} = \mathbf{M}^T(x).\mathbf{A}^T$ and $\boldsymbol{s_2} = \neg\mathbf{M}^T(x)$. Therefore, $\mathbf{D}(x) = (\mathbf{M}^T(x).\mathbf{A}^T).\neg\mathbf{M}^T(x)$ that results in

$$\mathbf{D} = (\mathbf{M}^T.\mathbf{A}^T) \cap \neg\mathbf{M}^T = (\mathbf{A}.\mathbf{M} - \mathbf{M})^T.$$

□

**Lemma 4.3** $DF(S) = \boldsymbol{S}.\boldsymbol{D}$

**Proof.** Let $\boldsymbol{S}$ be a Boolean vector representation of set $S$. Then, $\bigcup_{x \in S} DF(x)$ is rewritten as $\sum_{x \in \boldsymbol{S}} \mathbf{D}(x) = \boldsymbol{S}.\mathbf{D}$ (cf. Equation 10). □

**Lemma 4.4** $J^+(S) = \boldsymbol{S}.\boldsymbol{D}^+$

**Proof.** The iterated dominance frontier of $S$ is $J^+(S) = \lim_{i \to \infty} DF^i(S)$ where $DF^1(S) = DF(S)$ and $DF^{i+1}$(S)= $DF(S \cup DF^i(S))$, as given in Equation 9.

$$DF^1(S) = DF(S) \tag{12}$$

$$\forall i \; DF^{i+1}(S) = DF(S \cup DF^i(S)) \tag{13}$$

$$J^+(S) = \lim_{i \to \infty} DF^i(S) \tag{14}$$

Equation 13 is rewritten as follows:

$$DF^{i+1}(S) = DF(S \cup DF^i(S)) = (\boldsymbol{S} + DF^i(S)).\mathbf{D} \tag{15}$$

The closed form of the recurrence relation is $DF^i(S) = (\boldsymbol{S} + DF^{i-1}(S)).\mathbf{D} = \boldsymbol{S}.\sum_{k=1}^{i} \mathbf{D}^k$ and Equation 14 is transformed to $J^+(S) = \lim_{i \to \infty} DF^i(S) = \lim_{i \to \infty} \boldsymbol{S}.\sum_{k=1}^{i} \mathbf{D}^k = [\boldsymbol{S}.\mathbf{D}^+]$. □

### *4.3 Extended Transitive Closure*

In our unified approach for constructing SSA form we use the extended transitive closure function $f^*(A, B, C)$ (see Definition 5) which is an extension of a transitive closure of a graph. For the simple case, i.e. $C$ is the one matrix $\mathbf{1}$, simple and fast algorithms are suitable such as Warshall's algorithm [16], which exhibits a worst-case runtime of $O(n^3)$. More sophisticated algorithms such as Munro [26] and fastest matrix multiplication techniques [9] with a squaring technique reduces the asymptotic worst-complexity to $O(n^{2.376} \log n)$. For the more general case (i.e. $C$ is not the one matrix) techniques introduced for computing data-flow analysis [28] can be used. This approach uses dynamic programming techniques to efficiently compute the extended transitive closure of a relation.

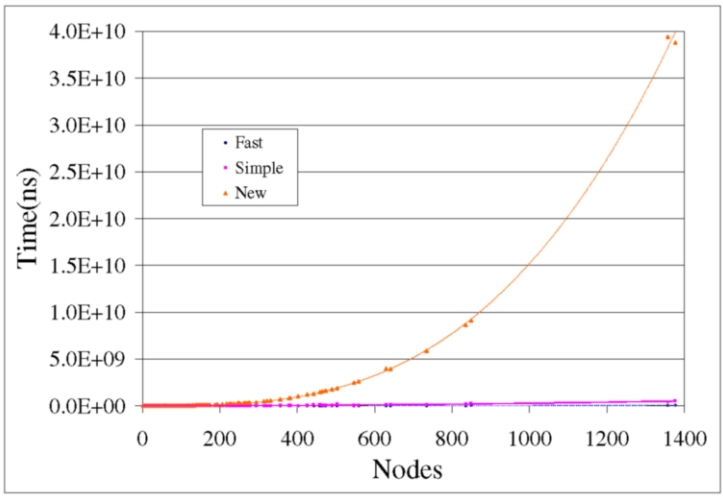



Fig. 3. Experiments:Dominance Relation

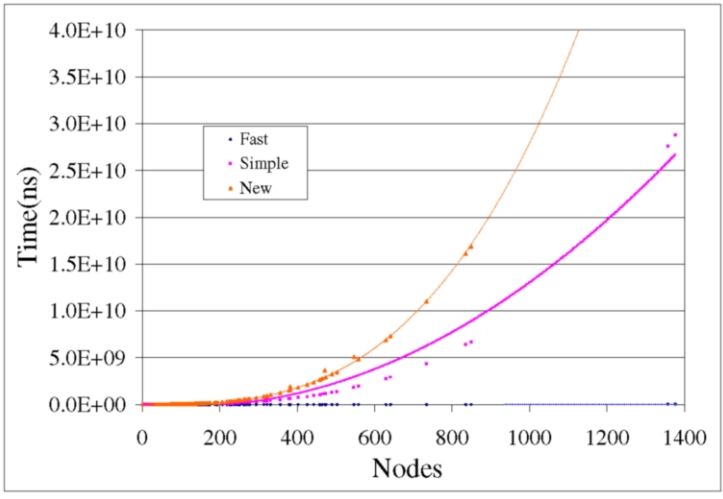



Fig. 4. Experiments: Dominance Frontier

## 5 Experiments

We implemented our new approach as part of an optimiser written in Java for a virtual machine [7]. The framework reads bytecode generated by LCC [13] and constructs SSA form. After performing machine independent optimisations the optimised bytecode is emitted.

The experiments for the new approach and the state-of-the-art approaches were conducted on a Linux platform with 1GB RAM and a 2.0GHz CPU, running Fedora Core 2. The experiments were compiled and run with Java SDK 1.5.0. As a benchmark suite we used the Spec95 integer benchmark programs.

As a yardstick for our new approach we implemented Purdom and Moore's algorithm [27], and Lengauer and Tarjan (LT79) [23] to compute the dominance relation. For non-iterated and iterated dominance frontier we implemented the

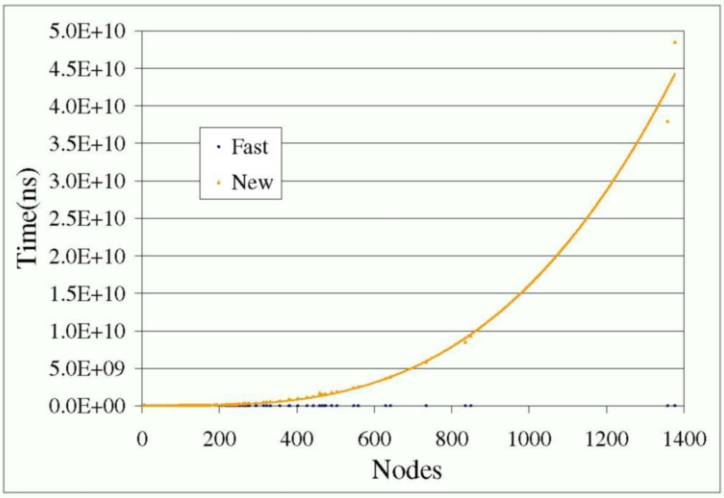



Fig. 5. Experiments: Iterated Dominance Frontier

algorithms described in [25,8].

In Fig. 3 the performance values of the fast [23], simple [27], and of our approach are depicted. The time measurements are in nano seconds. The best-fit line of our new approach ("new") shows cubic-time complexity whereas the other approaches have a linear or slight quadratic complexity to cubic time behaviour.

The performance of the dominance frontier is shown in Fig. 4. We compared the performance of the "fast" dominance frontier algorithm [8] with the "simple" dominance frontier algorithm [25] and our "new" approach. For each generated control-flow graph, we computed the dominance frontiers of all nodes and measured the execution time. The "fast" algorithm is almost linear. The "simple" algorithms shows a quadratic-time complexity; while our new approach is the slowest with cubic-time.

For the iterated dominance frontier, we compared our new approach with the standard recursive algorithm described in [25,8]. Fig. 5 shows the performance of the "fast" algorithm (recursive) and of our "new" approach. In our experiments, we first computed the dominance frontier $DF$ for all nodes and generated a random vertex set $S$. $J^+(S)$ is then computed once for each approach and we measured the execution time. As shown in the figure, the "fast" algorithm runs. In our approach, the first computation of $J^+(S)$ requires a computation of transitive closure $\mathbf{D}^+$ and hence it has a cubic time complexity as depicted.

The experimental results indicate that our algorithms have a cubic time-complexity. However, this result goes in line with our theoretical considerations of our unified approach. A better algorithm for finding transitive closures (instead of performing a simple recursion) would improve the performance significantly. Note the implementation effort of our approach is minimal because only a simple binary matrix calculator needs to be implemented. Our approach has the smallest LOC( lines of code) in all three cases.

# 6 Conclusions

In this paper, we demonstrated a unified approach for computing graph relations used to compute SSA form. The state-of-the-art algorithms are elaborate to implement. To overcome this problem we advised a unified approach using Boolean matrix operations that expresses dominance, and iterated and non-iterated dominance frontier as simple Boolean Matrix equations. A simple binary matrix calculator is sufficient to compute the relations. Our approach is useful for (1) rapid prototyping of compilers and virtual machines and (2) as a mean to validate more sophisticated algorithms.

# Acknowledgement

We would like to thank Bernd Burgstaller for his useful comments and for proofreading the manuscript. This work has been supported by the ARC Discovery Project Grant "Compilation Techniques for Embedded Systems" under Contract DP 0560190.

# References

[1] Agrawal, R., S. Dar and H. V. Jagadish, *Direct transitive closure algorithms: design and performance evaluation*, ACM Trans. Database Syst. **15** (1990), pp. 427–458.

[2] Alstrup, S., D. Harel, P. W. Lauridsen and M. Thorup, *Dominators in linear time*, SIAM Journal on Computing **28** (1999), pp. 2117–2132.

[3] Appel, A. W., "Modern compiler implementation in Java," Cambridge University Press, 1998.

[4] Aycock, J. and R. N. Horspool, *Simple generation of static single-assignment form*, in: *Proceedings of the 9th Internation Conference on Compiler Construction*, Lecture Notes in Computer Science **1781** (2000), pp. 110–124.

[5] Brandis, M. M. and H. Moessenboeck, *Single-pass generation of static single-assignment form for structured languages*, ACM Transactions on Programming Languages and Systems **16** (1994), pp. 1684–1698.

[6] 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.

[7] Burgstaller, B., B. Scholz and M. A. Ertl, *An embedded systems programming environment for c.*, in: W. E. Nagel, W. V. Walter and W. Lehner, editors, *Euro-Par*, Lecture Notes in Computer Science **4128** (2006), pp. 1204–1216.

[8] Cooper, K. D. and L. Torczon, "Engineering a Compiler," Morgan Kaufmann, 2004.

[9] Coppersmith, D. and S. Winograd, *Matrix multiplication via arithmetic progressions*, in: *Proc. of Symposium on Theory of Computing*, 1987, pp. 1–6.

[10] 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.

[11] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and K. Zadeck, *An efficient method of computing static single assignment form*, in: *POPL89*, 1989, pp. 25–35.

[12] Fischer, M. J. and A. R. Meyer, *Boolean matrix multiplication and transitive closure*, in: *Proceedings of the Twelfth Annual Symposium on Switching and Automata Theory* (1971), pp. 129–131.

[13] Fraser, C. W. and D. R. Hanson, *A retargetable compiler for ANSI C*, Technical Report CS–TR–303–91, Princeton University, Department of Computer Science, Princeton, N.J. (1991).

[14] Georgiadis, L. and R. E. Tarjan, *Finding dominators revisited: extended abstract*, in: *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms* (2004), pp. 869–878.

[15] G.Ramalingam, *On loops, dominators, and dominance frontiers*, ACM Trans. Program. Lang. Syst. **24** (2002), pp. 455–490.

[16] Henry S. Warren, J., *A modification of warshall's algorithm for the transitive closure of binary relations*, Commun. ACM **18** (1975), pp. 218–220.

[17] IBM Research, *Jikes RVM Project*. URL http://jikesrvm.sourceforge.net/

[18] Ioannidis, Y. E., *On the computation of the transitive closure of relational operators*, in: *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases* (1986), pp. 403–411.

[19] Ioannidis, Y. E. and R. Ramakrishnan, *Efficient transitive closure algorithms*, in: *VLDB '88: Proceedings of the 14th International Conference on Very Large Data Bases* (1988), pp. 382–394.

[20] Kim, K. H., "Boolean Matrix Theory and Applications," Dekker, New York, 1982, 288 pp.

[21] Koppelberg, S., "Handbook of Boolean Algebras," Elsevier Science Ltd, 1989.

[22] Lattner, C. and V. Adve, *Llvm: A compilation framework for lifelong program analysis & transformation*, in: *CGO '04: Proceedings of the international symposium on Code generation and optimization* (2004), p. 75.

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

[24] Matsunaga, Y., P. C. McGeer and R. K. Brayton, *On computing the transitive closure of a state transition relation*, in: *DAC '93: Proceedings of the 30th international conference on Design automation* (1993), pp. 260–265.

[25] Muchnick, S. S., "Advanced Compiler Design and Implementation," Morgan Kaufmann Publishers, 1997.

[26] Munro, J. I., *Efficient determination of the transitive closure of a directed graph.*, Inf. Process. Lett. **1** (1971), pp. 56–58.

[27] Purdom, P. W. and E. F. Moore, *Immediate predominators in a direct graph*, Communications of the ACM **15** (1972), pp. 777–778.

[28] Reps, T., S. Horwitz and M. Sagiv, *Precise interprocedural dataflow analysis via graph reachability*, in: *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, 1995, pp. 49–61.

[29] Shyamasundar, R. K., *A note on the transitive closure of a boolean matrix*, SIGACT News **9** (1978), pp. 18–21.

[30] Sreedhar, V. C. and G. R. Gao, *A linear time algorithm for placing &phgr;-nodes*, in: *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1995), pp. 62–73.

[31] SUN Microsystems, *The Java HotSpot Virtual Machine, v1.4.1, White Paper, Sun Microsystems* (2006).

[32] Ullman, J. D. and M. Yannakakis, *The input/output complexity of transitive closure*, in: *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data* (1990), pp. 44–53.