SSI Properties Revisited

BENOIT BOISSINOT, ENS-Lyon
PHILIP BRISK, University of California, Riverside
ALAIN DARTE, CNRS
FABRICE RASTELLO, Inria

The static single information (SSI) form is an extension of the static single assignment (SSA) form, a well-established compiler intermediate representation that has been successfully used for numerous compiler analysis and optimizations. Several interesting results have also been shown for SSI form concerning liveness analysis and the representation of live-ranges of variables, which could make SSI form appealing for just-in-time compilation. Unfortunately, we have uncovered several mistakes in the previous literature on SSI form, which, admittedly, is already quite sparse. This article corrects the mistakes that are most germane to SSI form. We first explain why the two definitions of SSI form proposed in past literature, first by C. S. Ananian, then by J. Singer, are not equivalent. Our main result is then to prove that basic blocks, and thus program points, can be totally ordered so that live-ranges of variables correspond to intervals on a line, a result that holds for both variants of SSI form. In other words, in SSI form, the intersection graph defined by live-ranges is an interval graph, a stronger structural property than for SSA form for which the intersection graph of live-ranges is chordal. Finally, we show how this structure of live-ranges can be used to simplify liveness analysis.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Control-flow graph, interval graph, liveness analysis, loop nesting forest, static single assignment (SSA), static single information (SSI), intersection/interference graph, program structure tree (PST)

ACM Reference Format:
Boissinot, B., Brisk, P., Darte, A., and Rastello, F. 2012. SSI properties revisited. ACM Trans. Embed. Comput. Syst. 11S, 1, Article 21 (June 2012), 23 pages.
DOI = 10.1145/2180887.2180898 http://doi.acm.org/10.1145/2180887.2180898

1. INTRODUCTION

Since its introduction [Alpern et al. 1988; Cytron et al. 1991; Rosen et al. 1988], the static single assignment (SSA) form has been a very successful program representation that has been used for numerous compiler analyses and optimizations. In recent years, the possibility of performing register allocation using SSA form has emerged as an active research topic. The first key result is that the interference graph for a procedure

This work was supported by the Nano2012 Mediacom contract with the Compiler group of STMicroelectronics, Grenoble.
Authors' addresses: B. Boissinot, A. Darte, and F. Rastello, Compsys team, LIP, UMR CNRS, ENS Lyon, UCB Lyon 1, Inria 5668, France; email: benoit.boissinot@ens-lyon.org, {alain.darte, fabrice.rastello}@inria.fr; P. Brisk, University of California, Riverside, CA; email: philip@cs.ucr.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1539-9087/2012/06-ART21 $10.00
DOI 10.1145/2180887.2180898 http://doi.acm.org/10.1145/2180887.2180898

in SSA form is a chordal graph [Bouchez et al. 2005; Brisk et al. 2006; Hack et al. 2006; Pereira and Palsberg 2005]. Many problems, including graph coloring, that are NP-complete for general graphs have efficient polynomial-time solutions for chordal graphs and their subclasses. Nevertheless, the most important problems of register allocation in compilers, spilling and coalescing, remain NP-complete even in SSA form [Bouchez et al. 2006].

The static single information (SSI) form is an extension of the SSA form that has remained relatively obscure since its introduction [Ananian 1999]. The SSA form renames variables so that each variable is textually assigned exactly once, which may require to introduce new variables, with an operator called a ϕ-function, at some joint points of the control-flow graph. The SSI form goes one step further: it also introduces variables, with an operator called a σ-function, at some split points of the control-flow graph so that each variable can be renamed in each conditional context where it is used. The initial motivation for SSI form was to get a more natural representation for array bounds checking, bitwidth analysis, and bidirectional data flow analyses in general.

More recently, an investigation into the possibility of using SSI form for register allocation yielded a proof that the interference graph for a program represented in SSI form belongs to the class of interval graphs [Brisk and Sarrafzadeh 2007]. The structure of interval graphs is interesting because the k-colorable subgraph problem, which is NP-complete for chordal graphs, has an efficient polynomial-time solution for interval graphs [Yannakakis and Gavril 1987]. Some register allocators, based on graph coloring, model the spilling problem, that is, how to partition the set of variables between registers and memory, as a variant of the k-colorable subgraph problem.

Unfortunately, we discovered an error in the aforementioned proof regarding SSI form and interval graphs; at the time, we believed that the statement of the theorem was in fact correct, and that the proof itself could be corrected with minimal effort. Upon closer inspection, however, we discovered that this mistake had been made in an earlier paper as well, and that other mistakes had been made in several papers that built on SSI form. The goal of this article, therefore, is to clear up the mistakes to the greatest possible extent.

The key mistake was the assumption that the basic blocks of a control flow graph could be ordered according to the orders induced by the dominance and postdominance relations, at the same time. In other words, if basic block b_1 dominates basic block b_2, then b_1 should occur before b_2 in the order; similarly, if basic block b_4 postdominates basic block b_3, then b_3 should occur before b_4 in the order. In fact, such an order of basic blocks does not exist in general: for example, the header node of a natural loop (e.g., a for or a while loop) may simultaneously dominate and postdominate all of the basic blocks contained in the loop. This error was made by Johnson et al. [1994], who introduced a data structure called the program structure tree (PST), which attempted to represent the structure of a control flow graph hierarchically. Brisk and Sarrafzadeh [2007] later based their proof that the interference graph for a program in SSI form is an interval graph on the same erroneous assumption.

Ananian introduced the SSI form in his master's thesis and gave a construction algorithm based on the PST definition [Ananian 1999]. As this definition is unsound, it follows that Ananian's SSI construction algorithm is incorrect. Nevertheless, his definition of SSI form remains sound. It may be possible to fix the underlying errors that plague the PST, but we do not attempt to do so here.

Singer [2006] introduced an alternative definition of SSI form and assumed that it was equivalent to Ananian's. Singer also presented an alternative construction algorithm for SSI form. We show that these two definitions of SSI form are, in fact,

distinct from one another. However, it appears that Singer's construction algorithm produces an SSI form that corresponds to Ananian's definition. At present, there is no construction algorithm for SSI form that corresponds to Singer's definition, and we do not attempt to present such an algorithm here. Henceforth, we will refer to Ananian's definition of SSI as the *strong* SSI form and Singer's definition as the *weak* SSI form, as Ananian's definition is more constrained.

Finally, we correct the errors in the paper by Brisk and Sarrafzadeh [2007]. We prove that the intersection graph defined by the live-ranges of variables in SSI form is indeed an interval graph using an adequate order of basic blocks, a property that holds both for strong and weak SSI form. In both cases, our proof is constructive, we are even able to exhibit an order that depends only on the structure of the control-flow graph, that is, not on the live-ranges of the variables. For strong SSI form, the order we build is defined from the dominance and postdominance relations. For weak SSI form, our construction is more involved and relies on the concept of loop nesting forest [Ramalingam 2002]. Brisk and Sarrafzadeh also stated that liveness analysis for a procedure in SSI form can be accomplished in a single iteration of a standard iterative dataflow solver, essentially eliminating the "iterative" aspect of the analysis. This result was based on the same incorrect assumption that the basic blocks can be ordered by the dominance and postdominance relations. We correct their proof, showing that it holds only for strong SSI form. We also show how it can be done for weak SSI form.

2. FOUNDATIONS

This section recalls or specifies all the needed definitions and concepts on control-flow graphs, liveness analysis, and SSA & SSI forms.

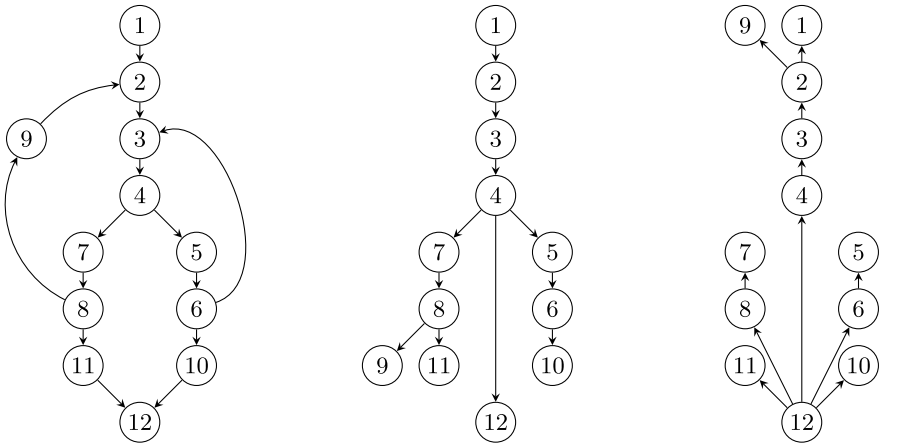
2.1 Preliminaries

2.1.1 Control-Flow Graph. A procedure is represented as a *control-flow graph* (CFG), which is a directed graph $G = (V, E, r, t)$, with set of nodes V, set of edges E, and two specific nodes r and t: r is the *entry* node, with no incoming edge, and t is the *exit* node, with no outgoing edge.

A *path* P of length $k \geq 0$ from a node u to a node v in V is a nonempty sequence $(v_0, v_1, \ldots, v_k)$ of nodes such that $u = v_0$, $v = v_k$, and $(v_{i-1}, v_i) \in E$ for $i \in [1..k]$. With this definition, a path of length 0 is a path with one node and no edge. If the CFG contains a self-edge, that is, an edge of the form (u, u), then there is also a path of length 1 from u to itself. Node v is *reachable* from u if there is a path from u to v in the CFG. In this article, we assume that every node is reachable from r and that t is reachable from every node as well.[1]

Each node in the CFG represents a *basic block*, that is, a sequence of successive instructions in the program with no branches or branch targets interleaved. A *program point* denotes the entry/exit of each basic block, or the implied space between two instructions within a basic block. In the usual definition of basic block, the sequence of instructions that define it should be of maximal length. In this case, a program point generally does not refer to a CFG node itself, unless it corresponds to an empty basic block. In our case, we sometimes need to discuss at the granularity of program

[1]These two properties are needed to define the dominance and postdominance relations, and thus the SSI form. If they are not fulfilled, some artificial edges must be added to the CFG. The SSI form then depends on this completion, but all results of this article apply.



(a) Control-flow graph (b) Dominator tree (c) Postdominator tree

Fig. 1. Example of control-flow graph with its associated dominator and postdominator trees.

points and instructions, for example for defining live-ranges of variables. At the same time, we do some reasoning at the granularity of basic blocks of maximal length as well, for example for ordering them. Therefore, there are appropriate times where we must talk about program points, and others at which we must talk about CFG nodes. As instructions and program points are totally ordered in a basic block, the notion of paths, of reachability, of dominance, of postdominance, of total order, etc., can all be defined similarly for program points and instructions. This corresponds to manipulate a CFG at a different granularity.

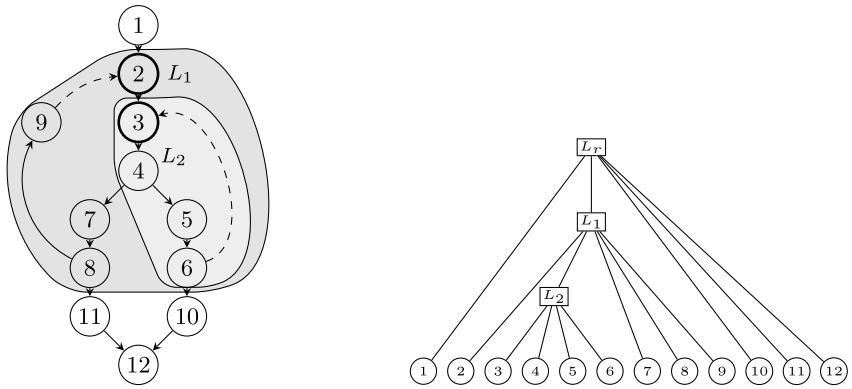
2.1.2 Dominance/Postdominance. A node u in a CFG *dominates* a node v, denoted u dom v, if every path from the entry node r to v contains u. If u dom v and $u \neq v$, then u *strictly dominates* v, denoted u sdom v. The node u is the immediate dominator of v, denoted idom(v), if $u$sdom$v$ and there exists no node w such that $u$sdom$w$ and $w$sdom$v$. Every CFG node other than r has a unique immediate dominator. The directed graph whose nodes are the nodes of the CFG and in which each node other than r is pointed to by its immediate dominator is a tree rooted at r, called the *dominator tree*.

The postdominance relationship is defined similarly in a CFG. A node v *postdominates* a node u, denoted v pdom u, if every path from u to the exit node t contains v. The *strict postdominance* ($v$spdom$u$), the *immediate postdominator* (ipdom(u)), and the *postdominator tree* rooted at t are defined analogously. Notice that the postdominance information is equivalent to the dominance information if the direction of every edge of the CFG is reversed.

Figure 1 gives an example of CFG (with maximal-length basic blocks), its dominator tree, and its postdominator tree.

2.1.3 Minimal Loop Nesting Forest. The *minimal loop nesting forest* [Ramalingam 2002] plays an important role in the theory outlined in this article. This section describes this data structure in detail.

A *strongly connected component* (SCC) in a directed graph is a maximal subgraph such that every node is reachable from any other node. A *trivial* SCC contains a single node with no edge (i.e., no self-loop); all other SCCs are *nontrivial*. If each SCC in a directed graph is collapsed into a single node, then the resulting graph is a directed acyclic graph (DAG).



(a) CFG with loops (shaded regions), loop-edges (dotted lines), and loop headers (bold nodes)

(b) Loop forest with an additional root node

Fig. 2. Example of control-flow graph with an associated minimal loop forest (tree).

A minimal loop nesting forest can be defined recursively as follows.

— Partition the CFG into its SCCs. Each nontrivial SCC is called a *loop*.
— Within each loop, consider the set of nodes that are not strictly dominated by any other node of the same SCC. Among these nodes, choose a nonempty subset, called the *loop-header*. The nodes it contains are the loop headers.
— Remove all edges, inside the SCC, that enter a node of the loop-header. Call these edges *loop-edges*.
— Once loop-edges are removed, each SCC can again be partitioned into its different SCCs. The process is repeated until there remain only trivial SCCs.

This decomposition can be represented by a forest as follows. For each node of the CFG, there is a corresponding leaf node in the forest. For each nontrivial SCC (i.e., each loop) encountered in the decomposition, there is a corresponding internal node in the forest, labeled by its loop-header. The successors of an internal node (i.e., of a loop) are the different SCCs obtained after removing its loop-edges, thus all inner loops as well as all singletons without self-edge (this includes all loop-headers in particular). When the CFG is not strongly connected, the forest contains more than one tree. Then, a root, corresponding to the CFG as a whole, can be added to convert the forest representation to a tree, called the *loop-tree*.

This process is illustrated in Figure 2. The CFG has one strongly connected component (in dark gray in Figure 2(a)), with a unique loop header, the node 2. The loop-edge from 9 to 2 is removed and the process continues. Again, one SCC is identified (in light gray in Figure 2(b)), with a unique loop header, the node 3. The resulting loop-tree is depicted in Figure 2(b): the node L_1 corresponds to the first identified loop, the node L_2 to the second one, and the node L_r corresponds to the artificial node added to transform the forest into a tree.

Different loop nesting forests have been proposed in past literature [Havlak 1997; Sreedhar et al. 1996; Steensgaard 1993]. Ramalingam formally proved that all of them are minimal loop nesting forests as defined previously; they differ only in their definition of loop headers. As for us, we will need an additional property, that all loop headers are *entry nodes* of the corresponding loop, that is, nodes with an incoming edge from outside of the loop. In this case, we say that the loop forest is *connected*, for reasons that will be explained later (see Lemma 4.5).

2.1.4 Liveness and Intersection/Interference Graph. Consider a variable x in a CFG. An instruction that assigns a value to x is called a *definition* of x. An instruction that uses x is called a *use* of x. The variable x is said to be *live* at some program point p in the CFG if there is a path from a definition of x to p and a path from p to a use of x that does contain any other definition of x.[2] The *live-range* of x is the set of program points in the CFG where x is live. In the rest of the article, when we discuss about liveness, we talk about definitions and uses as if they were program points. Actually, we mean the program points just after the definitions and just before the uses, respectively.

As mentioned in Section 1, a code in SSA form may contain ϕ-functions and a code in SSI form may contain both ϕ-functions and σ-functions. These instructions define and use variables. We will explain later how the liveness of the corresponding variables is defined.

The concept of *interference graph* was popularized in two papers published by Chaitin et al., in the early 1980s [Chaitin et al. 1981; Chaitin 1982]. Two variables interfere if they cannot be assigned to the same storage location. The interference graph is an undirected graph with one node per variable and an edge between any two nodes that correspond to interfering variables. The "ultimate" definition of interference given by Chaitin et al. states that two variables interfere if and only if there is at least one program point where they are both live and they do not hold the same value. Unfortunately, this definition of interference is not of practical use as it is not possible, in general, to determine statically if two variables hold the same value. The vast majority of literature on register allocation states that two variables interfere if and only if the intersection of their live-ranges is nonempty. In this case, the interference graph is just the *intersection graph* of the live-ranges, that is, there is an edge between two nodes if and only if the two corresponding live-ranges overlap. We make the same assumption: when we use the term interference graph, we actually mean the intersection graph of the live-ranges. The most important theoretical result presented in this article is that, when variables are in SSI form, this intersection graph is an *interval graph*, that is, live-ranges can be represented as intervals on a linearly ordered set, for instance, the real line [Golumbic 2004].

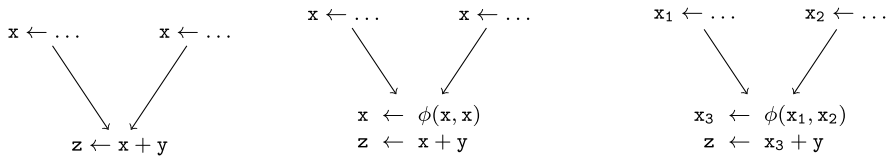
We point out that, in practice, a slightly less conservative definition of interference can be used that takes the *move* instructions into account [Boissinot et al. 2009; Chaitin 1982]. But we will not consider it here as we are only interested in the structure of live-ranges, not their content.

2.2 Static Single Assignment (SSA) Form

The *static single assignment* (SSA) *form* was introduced in a pair of papers, published at POPL'88, that focused on the identification of redundant computations [Alpern et al. 1988; Rosen et al. 1988]. Three years later, a more comprehensive journal paper described SSA form's theoretical underpinnings and presented an efficient construction algorithm [Cytron et al. 1991]. This journal paper has been widely cited, and is generally credited with effectively popularizing SSA form among compiler researchers and practitioners.

2.2.1 Unique Definitions. A code is in SSA form if it satisfies *the unique definition property*, in other words each variable defined locally in a procedure is defined (textually) only once [Appel and Palsberg 2002; Cooper and Torczon 2004]. Of course, to transform a non-SSA code into SSA form, it is not enough to replace the n definitions of a

[2]It is often assumed that every path from the entry node r to a use of a variable contains a definition of it (strict program, see Section 2.2.3). In this case, it is sufficient to state that x is live at p if there is a path from p to a use of x.



(a) Non-SSA program (b) With single reaching definitions (c) SSA program after renaming

Fig. 3. Placement of ϕ-functions.

variable by n definitions of n different variables. Each use of the original variable must also be renamed using the adequate new variable whose definition reaches it. When several definitions can reach a given use, renaming the use to one of them changes the semantics of the program; similarly, not renaming it is also an error. To address this issue, the SSA form uses the concept of ϕ-functions.

A ϕ-function is placed at a control-flow join point, it selects the adequate variable depending on which path is executed, and it copies the selected value to a variable whose value is now statically disambiguated from the control flow. This definition can now be used to rename the subsequent uses that it reaches, therefore ensuring correct program execution semantics. Several ϕ-functions placed at the entry of the same block act as parallel statements. To define liveness and dominance, each use of a ϕ-function is considered to take place, not at the entry of the basic block where the flow joins, but at the exit of each corresponding predecessor block.

For example, in Figure 3, two definitions of x reach the use of x, which requires the introduction of a ϕ-function to disambiguate them. The three definitions of x can be renamed into x_1, x_2, and x_3, and the initial use of x now refers to x_3 defined by the ϕ-function. The use of x_1 (resp. x_2) in the ϕ-function is considered to take place at the end of the upper-left (resp. upper-right) basic block.

2.2.2 Minimal SSA. Notice that, so far, we did not specify how to transform a code into SSA form: we simply state a property to be satisfied (the unique definition property) and the semantics of the ϕ-functions that can be used to fulfill it.

A program point p is *reached by a definition* of a variable x if there is a path from this definition to p that does not contain any other definition of x. A code has the *single reaching-definition property* if no program point can be reached by two definitions of the same variable. The standard construction algorithm for SSA form is decomposed into two phases. First, for each variable x, ϕ-functions of the form $x = \phi(x, \ldots, x)$ are inserted, so that the code fulfills the single reaching-definition property. Then, in a second phase, variables can be renamed so that the code contains textually only one definition per variable and each use then refers to its unique corresponding reaching definition. Figure 3 illustrates this mechanism.

For ϕ-functions insertion, Cytron et al. used the notion of join nodes [Cytron et al. 1991]. A node w is a *join* node for nodes $u \neq v$, denoted $w \in \mathcal{J}(\{u, v\})$, if there exist two paths, one from u to w and one from v to w, with only w in common. To enforce the single reaching-definition property for a variable x, ϕ-functions are inserted at the (iterated) join set $\mathcal{J}^+(D_x)$ where D_x is the set of definitions of x.[3]

Actually, Cytron at al. do not manipulate join sets directly. Instead, they place the ϕ-functions at the *iterated dominance frontier* $\mathcal{DF}^+(D_x)$ (see, for example, Appel and

[3]For a set of nodes S, $J^+(S)$ is the limit of the increasing sequence of sets $J_1 = J(S)$, $J_{i+1} = J(S \cup J_i)$. The iteration is motivated by the fact that the set of definition points grows as we insert ϕ-functions. In fact $J^+(S) = J(S)$ but this is out of the scope of this article.

Palsberg [2002] for a definition), which is an over-approximation of the iterated join sets. If the set of definitions of a variable is enlarged with a *pseudodefinition* at the CFG entry r, the two insertion techniques are equivalent as $\mathcal{DF}^+(D_x) = \mathcal{J}^+(D_x \cup \{r\})$. This is the case of the SSA form, called *minimal SSA form*, constructed by Cytron et al.'s algorithm.

2.2.3 SSA Form with Dominance Property. Considering that every variable has a pseudo-definition at the CFG entry node provides an interesting equivalent characterization, which we recall here.

A procedure is *strict* if every path from r, the CFG entry node, to a use of a variable passes through a definition of this variable. Implicitly, a code with pseudo-definitions at r is strict. If, in addition, each use of a variable is reached by only one definition of this variable (single reaching-definition property), then the use is dominated by the definition. We call this property the *dominance property*.

Conversely, if, before renaming (and a fortiori after renaming), for each variable, each use is dominated by any definition that reaches it, then the definition is unique. Thus, enforcing the dominance property or enforcing the single reaching-definition property, with a pseudodefinition, are two equivalent mechanisms to insert ϕ-functions. This second equivalent definition of SSA form has been reported in a recent textbook [Cooper and Torczon 2004]: a program is in SSA form if, before renaming, it satisfies the dominance property. Notice that, in this case, after renaming, the (unique) definition of each variable dominates its entire live-range.

2.2.4 Liveness and Variants of SSA. The main drawback of minimal SSA form is that it may insert ϕ-functions for a variable x at join points where x is not live in the original non-SSA code. Two SSA variants have been introduced, which reduce the number of required ϕ-functions, the semipruned and the *pruned* SSA forms.

The semipruned SSA form is based on the observation that many variables are never used outside of the basic block where they are defined [Briggs et al. 1998]. Specifically, if every use of a variable is preceded by a definition of it in the same basic block, then semipruned SSA form will not instantiate any ϕ-functions for this variable. Minimal SSA form is then constructed for the remaining variables that are not filtered in this manner.

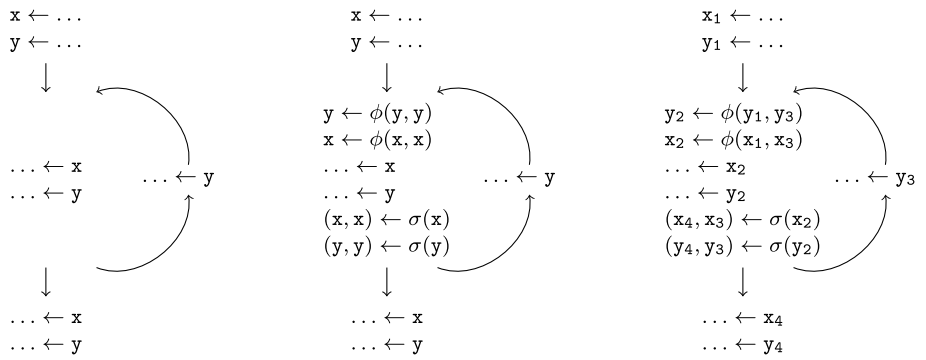
The pruned SSA form is based on the observation that there is no need to instantiate a ϕ-function for a variable at program points where it is not live. One method to construct pruned SSA form is to run liveness analysis in advance, and then suppress the instantiation of ϕ-functions for variables that are not live. A second method is to build minimal or semipruned SSA form and then to eliminate the extraneous (dead) ϕ-functions using dead code elimination.

2.3 Static Single Information (SSI) Form

The *static single information* (SSI) form is an extension of SSA form that treats uses and definitions symmetrically with respect to one another.

2.3.1 Ananian's Definition of SSI. Ananian [1999] introduced SSI form in a similar way to what Cytron et al. did for SSA, that is, using properties on paths. To this end, the notions of split set (similar to join set for SSA) and of upward-exposed use (similar to reaching definition for SSA) are used.

The *split* set of two CFG nodes $u \neq v$, denoted $\mathcal{S}(\{u, v\})$, is the set of nodes w such that there exist two paths, one from w to u and one from w to v, with only w in common. A use of a variable x is *upward-exposed* at a program point p if there is a path from p



(a) Non-SSI program (b) With single reaching definitions and single upward-exposed uses (c) Strong SSI after renaming

Fig. 4. Placement of ϕ-functions and σ-functionsfor strong SSI.

to the use that does not go through any other use of x. [4] A procedure satisfies the *single upward-exposed-use property* if at most one use of each variable is upward-exposed at each program point p.

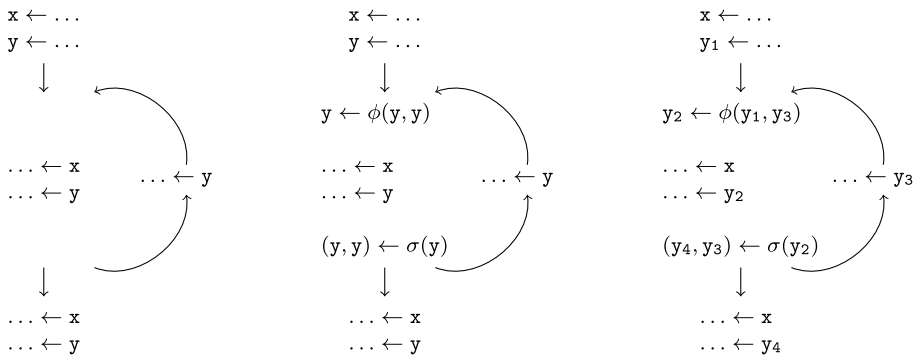
The SSA form construction inserts ϕ-functions at join points to merge multiple variable definitions, thereby satisfying the single reaching-definition property. Similarly, the SSI form construction inserts σ-functions at split points that reach multiple upward-exposed uses, thereby satisfying the single upward-exposed-use property. A σ-function has one argument (a variable use) and it defines as many variables as successors of the split point. Several σ-functions placed at the beginning of the same block act as parallel statements. To define liveness and dominance, each definition in a σ-function is considered to take place, not at the exit of the block where the flow splits, but at the entry of each corresponding successor block.

Ananian provided a definition of SSI in the spirit of the definition of SSA. Each variable has a *pseudo-use* at the CFG exit node t. A code is in SSI form if it is in minimal SSA form and if it satisfies the single upward-exposed-use property. To satisfy this property for each variable x, σ-functions of the form $(\mathtt{x}, \ldots, \mathtt{x}) = \sigma(\mathtt{x})$ can be inserted at the *iterated postdominance frontier* of the set of uses U_x, denoted $p\mathcal{DF}^+(U_x) = \mathcal{S}(U_x \cup \{t\})$. However, as σ-functions create new definitions, more ϕ-functions may be inserted. Then, in a later phase, variables can be renamed so that each variable is defined only once and all uses can be renamed accordingly.

Figure 4 illustrates Ananian's SSI definition. The situation is similar for x and y despite the use of y on the back-edge. Two uses are upward-exposed at the end of the central basic block, thus a σ-function is inserted. Now, the central use is reached by two definitions and a ϕ-function is added at block entry (see Figure 4(b)). Then, definitions and uses of x and y are renamed, so that each variable has a unique definition and each use refers to the right definition, see the code in Figure 4(c).

Ananian's algorithm for SSI construction, however, is not based directly on the iterated dominance and postdominance frontiers: instead, it uses a data structure called the program structure tree (PST). As the soundness of the PST is questionable

[4]Notice that this definition may differ from the one used by liveness analysis. Indeed, for liveness analysis the corresponding path should not contain any definition of the variable. In our context, both definitions of upward-exposed use can be considered, as soon as the code is in SSA form. Any SSI construction algorithm on a non-SSA code may have to cope with subtleties related to this notion of upward-exposed use. This is out of the scope of this article.



(a) Non-SSI program (b) With dominance and post-dominance properties (c) Weak SSI after renaming

Fig. 5. Placement of ϕ-functions and σ-functionsfor weak SSI.

(see Section 5.1), the correctness of his construction algorithm is arguable. This article does not, however, question the soundness of Ananian's definition of SSI form. To summarize, a code is in SSI form according to Ananian's definition if it satisfies, before final renaming, the *single reaching-definition* and *single upward-exposed-use* properties, assuming that the CFG has a pseudodefinition at the entry node r and a pseudo-use at the exit node t for every variable.

2.3.2 Singer's Definition of SSI. Singer, in his Ph.D. thesis [Singer 2006], proposed an alternative definition of SSI form: a program is in SSI form if, before renaming (and a fortiori after renaming), it satisfies both the dominance property and the postdominance property. The postdominance property is the symmetric of the dominance property, inverting uses and definitions. It means that, for each variable, any use that is upward-exposed at a definition of the variable postdominates this definition (such a use is thus unique). After renaming, as there is a single definition per variable, the (unique) definition of each variable is postdominated by all its uses, and only one is upward-exposed at the definition.

Singer assumed (incorrectly, as we will show) that this definition is equivalent to Ananian's. It is true that Ananian's definition does satisfy the postdominance property; however, Singer's definition does *not* imply the single upward-exposed use property for all program points, only for definition points as we explain in Section 3.1. Therefore, situations exist where Ananian's definition of SSI form leads to the instantiation of more σ-functions than with Singer's definition. An example is given in Figure 5. Now, with Singer's definition, the situation is different for x and y. The two uses of x postdominate its unique definition, so no σ-function is needed and, consequently, no ϕ-function either. For y, the use on the back-edge does not postdominate the definition of y, thus a σ-function is added at the end of the central block, then a ϕ-function at the entry of the basic block (see Figure 5(b)). Then, definitions and uses of y are renamed, leading to the code of Figure 5(c).

2.3.3 Semipruned and Pruned SSI Form. Similar in principle to SSA form, semipruned, and pruned variants of SSI form can be defined. It is worth saying that the pseudo-use considered by Ananian's definition is only used to guide the placement of σ-functions during SSI construction. It does not alter the live-range of a variable, as doing so would cause all variables defined locally in the procedure to interfere at the exit point, which is not, in fact, the case.

For the sake of simplicity, we assume pruned SSI form, unless stated otherwise.

3. WEAK AND STRONG SSI FORMS

We use the term strong SSI to denote the SSI form defined by Ananian, that is, with dominance property, a pseudo-use for each variable, and the single upward-exposed use property. We use the term weak SSI to denote the SSI form defined by Singer, that is, with dominance property and postdominance property. In this section, we discuss differences and common properties of these two SSI definitions.

3.1 Weak and Strong SSI Forms Are Not Equivalent

Figures 4 and 5 illustrate the difference between the two definitions of SSI form. Both definitions insert one ϕ-function and one σ-function for variable y; the difference is observed with respect to variable x. As we just discussed, the different key properties for inserting ϕ-functions and σ-functions are linked as follows:

(1) the single reaching-definition property, with pseudodefinitions, is equivalent to the dominance property;
(2) the single upward-exposed-use property, with pseudo-uses, implies the postdominance property; this is not an equivalence as illustrated in Figures 4 and 5.

The subtle difference between the single upward-exposed use property and the postdominance property is the following. If the postdominance property holds, then, before renaming, every use of a variable that is upward-exposed at a definition of the variable is required to postdominate this definition. As the postdominance relation forms a tree, this implies that, for each definition, there is a single upward-exposed use (possibly the pseudo-use, unless pruned SSI is considered). On the contrary, if the single upward-exposed use property is satisfied, then for *any* program point p, not necessarily a definition, there is a single upward-exposed use u. This implies that p is postdominated by u, otherwise there is a path from p to u and a path from p to the pseudo-use at the CFG exit node t, not containing u, and the single upward-exposed use property is not satisfied for p.

Thus, a procedure that has been converted to strong SSI form also satisfies the criteria for weak SSI form. Strong SSI enforces the single upward-exposed use property for all program points, pruned SSI for the entire live-range, and weak SSI only for all definitions. As our example illustrates, these conditions are not equivalent. For SSA, the situation is different, as enforcing the single reaching-definition property for all uses is equivalent to enforcing it for the entire live-range. To summarize, after renaming, for both weak and strong SSI forms, the unique definition of a variable dominates all its uses and each use postdominates its definition. In (pruned) strong SSI, an additional property is true: considering that each variable has a pseudo-use at the CFG exit node, then, for each program point of the live-range, there is a unique upward-exposed use.

3.2 Properties of Variables in Weak and Strong SSI Forms

Variables in SSI form satisfy certain properties, common to weak and strong SSI forms, which we will exploit in our proofs regarding the structure of live-ranges.

Let x be a variable in (strong or weak) SSI form and let d be its (unique) definition point. In (strong or weak) SSI form, each use of x postdominates d and the postdominance relation forms a tree. Therefore, all uses belong to a path, in the postdominator tree, from t, the CFG exit node, to d in the postdominator tree (note that paths in the postdominator tree take the reverse direction from the CFG, e.g., each CFG node is reachable from t). Therefore, one of these uses postdominates all other uses. We call this use the *last use* of x, and denote it by u.

Let LIVE denote the live-range of x, that is, the set of program points where x is live. By definition, for any strict program, $p \in$ LIVE if there is a path, in the CFG, from p to some use of x that does not go through d. Because all uses of x postdominate d, and u postdominates all other uses of x, the previous path can be extended to form a path from p to u that does not contain d. In other words, under both strong and weak SSI, LIVE is the set of points p such that there is a path from p to the last use u that does not contain d.

The next three lemmas hold for any variable x, in either weak or strong SSI. Lemma 3.1 states that if x is live at a node p of the postdominance tree, then it is live at all descendants of p that are not descendants of d, that is, in the whole subtree rooted at p minus the subtree rooted at d.

LEMMA 3.1. *If* $p \in$ LIVE, p pdom q, *and* $\neg d$ pdom q, *then* $q \in$ LIVE.

PROOF. Since $\neg d$pdom$q$, there exists a path, in the CFG, from q to t not containing d. Because p pdom q, this path must contain p. Thus, there must exist a subpath from q to p that does not contain d. Since $p \in$ LIVE, there exists a path from p to u that does not contain d. By concatenating these paths, we can construct a path from q to u that does not contain d; it follows that $q \in$ LIVE. □

Lemma 3.2 states that if x is live at a node p of the postdominance tree, then it is live at all ancestors of p that are not ancestors of u, that is, all nodes in the path of the postdominance tree leading to p and starting from, but excluding, the least common ancestor of p and u.

LEMMA 3.2. *If* $p \in LIVE$, q pdom p, *and* $\neg q$ pdom u, *then* $q \in$ LIVE.

PROOF. Since $p \in$ LIVE, there is a path, in the CFG, from p to u that does not contain d. Because $\neg q$ pdom u, there exists a path from u to t that does not contain q. Concatenating the two paths yields a path from p to t via u, such that the subpath from u to t does not contain q, and the subpath from p to u does not contain d. Since q pdom p, this path from p to t must contain q, thus q must be in the subpath from p to u that does not contain d. This establishes the existence of a path from q to u that does not contain d, from which follows $q \in$ LIVE. □

Lemma 3.3 states that if x is live at a node p of the postdominance tree, then the ancestors of d that are not ancestors of p dominate p, that is, p is dominated by all nodes in the path of the postdominance tree leading to d and starting from, but excluding, the least common ancestor of p and d.

LEMMA 3.3. *If* $p \in$ LIVE, q pdom d, *and* $\neg q$ pdom p, *then* q *dom* p.

PROOF. Consider a path, in the CFG, from r to p. As $p \in$ LIVE, d dom p, thus this path contains d. Since $\neg q$ pdom p, there exists a path from p to t without q. Since q pdom d, q must be included in the path from d to p, otherwise we get a path from d to t without q. It follows that q dom p. □

4. THE STRUCTURE OF LIVE-RANGES IN SSI

This section proves that, for a procedure in either strong or weak SSI form, the interference graph (more precisely, the intersection graph of the live-ranges; see Section 2.1.4) is an interval graph. More precisely, we prove that the nodes of a CFG can be totally ordered so that the live-range of each SSI variable corresponds to an interval in this linearized representation. Under both strong and weak SSI form, the start point of the interval is the definition of the variable. Under strong SSI form, the end point of the interval is a use of the variable, while this is not always true in weak SSI.

Our linearization order is based on the postdominator tree, the dominance relation, and, for weak SSI, a connected minimal loop nesting forest, as described in Section 2.1.3.

4.1 Strong SSI Form

The total order of blocks we provide in Section 4.2 for weak SSI form is of course also suitable for strong SSI form. Nevertheless, as the situation is simpler for strong SSI form, we prefer to define in this section a suitable order of blocks that is more intuitive and easier to build.

First, we compute a *dominance-based order*, that is, a partial order of blocks that respects the dominance relation. In such an order, if a node u strictly dominates a node v, then u appears before v. Such an order can be computed by a topological traversal of the dominator tree. It can also be obtained by a preorder or a reverse postorder of any *depth-first search* (DFS) of the CFG.

Second, a preorder DFS traversal of the postdominator tree is performed with one additional constraint on the order in which the child nodes are visited. Let u and v be two children of a given node in the postdominator tree, then the following must hold.

(1) *Constraint on dominance.* If u dom v, then the preorder DFS traversal of the postdominator tree should visit the subtree rooted at v before it visits the subtree rooted at u.

For that, it suffices to visit the children in the reverse order of the dominance-based order computed in the preceding step. The resulting linearization of the CFG nodes is called a *reverse strong interval order*. It starts at the CFG exit node and ends at the CFG entry node. The third and final step is to reverse the reverse strong interval order, yielding a *strong interval order*.

Note that an order of nodes (i.e., of basic blocks) extends directly to an order of program points by traversing all program points within a block from entry to exit.

Consider again the CFG of Figure 1(a), whose corresponding dominator and postdominator trees are shown in Figures 1(b) and 1(c) respectively. To find a dominance-based order, we can compute a DFS of the CFG considering the nodes in preorder:

$$[1, 2, 3, 4, 7, 8, 9, 11, 12, 5, 6, 10].$$

Next, we perform a preorder DFS traversal of the postdominator tree, shown in Figure 1(c). The children are visited with a priority given by the reverse dominance-based order, yielding a reverse strong interval ordering. For example, nodes 8 and 11 are both children of node 12 in the postdominator tree. Since node 8 dominates node 11, as shown in Figure 1(b), we must visit node 11 before node 8 during the preorder traversal of the postdominator tree. Similarly, we must also visit node 8 before node 4. One possible reverse strong interval order that results from this traversal is the following:

$$[12, 10, 6, 5, 11, 8, 7, 4, 3, 2, 9, 1].$$

Reversing this order yields the following strong interval order:

$$[1, 9, 2, 3, 4, 7, 8, 11, 5, 6, 10, 12].$$

Now, we move on to the proof itself. We prove that, if blocks (and thus program points) are (totally) ordered according to a strong interval order, then the live-range of each variable under strong SSI form is an interval whose start point is the variable definition and whose end point is its last use (as defined in Section 3.2).

First, we prove that, for any variable, any strong interval order visits the node (and thus the program point) that contains its definition before all other nodes where it is live. This result holds for both strong and weak SSI form and will be used in the proofs for both of these representations. In other words, the definition point will be the start point of the interval.

THEOREM 4.1. *Given a reverse strong interval order of the nodes in a CFG in either strong or weak SSI form, the program point that contains the definition of a given variable is always visited after all other program points where it is live.*

PROOF. We use the same notations as in Section 3.2: x denotes the variable of interest, d its definition point, u its last use, and LIVE its live-range. Let $p \in$ LIVE.

We first exclude the trivial case where d is postdominated by p in which case, in a DFS of the postdominator tree, d is always considered after p, in particular in a reverse strong interval order. Now suppose that d is not postdominated by p. We prove that p is not postdominated by d either. $p \in$ LIVE implies the existence of a d-free path from p to u. As u postdominates d, there is a d-free path from u to t. These two paths can be concatenated to provide a d-free path from p to t, which proves that p is not postdominated by d.

Since d and p are not comparable for the postdominance, we can define $q = \text{LCA}(d, p)$, $q \neq d$, $q \neq p$, to be the least common ancestor of d and p in the postdominance tree. Let p' and d' be the corresponding children of q. More formally, $\text{ipdom}(p') = \text{ipdom}(d') = q$, d' pdom d, and p' pdom p. Now, $\neg p'$ pdom d, by construction of p', and u pdom d, by definition of SSI. It follows that $\neg p'$ pdom u. Since p' pdom p, by construction of p', and $p \in$ LIVE, Lemma 3.2 ensures that $p' \in$ LIVE. Finally, because d' pdom d, by construction of d', and $\neg d'$ pdom p', by construction of d' and p', Lemma 3.3 proves that d' dom p'. It follows that d' and its descendant d is visited after p' and its descendant p in a reverse strong interval order. This is due to the criterion for choosing children during the preorder DFS of the postdominator tree: the reverse of the dominance order. □

The next theorem characterizes the live-range of a variable in strong SSI as the descendants, in the postdominator tree, of its last use, minus the descendants of its definition.

THEOREM 4.2. *In strong SSI form, the live-range of a variable is the set of program points postdominated by its last use but not postdominated by its definition.*

PROOF. Again, let x denote the variable of interest, d its definition point, u its last use, and LIVE its live-range. Recall that d dom u and u pdom d.

First, we show that u pdom p for every point $p \in$ LIVE. Assume to the contrary that there exists a path from p to t that does not go through u. Since $p \in$ LIVE, there is also a path from p to u. The existence of these two different paths contradicts the single upward-exposed-use property of strong SSI form, for at least one point, for example the point where they split. Therefore, the complete live-range is included in the set of descendants of u in the postdominator tree. Conversely, Lemma 3.1 shows that any descendant of u that is not a descendant of d belongs to LIVE. Finally, if p is a descendant of d, all paths in the CFG from p to u contain d, thus p cannot belong to LIVE. □

This yields our main result, in the case of strong SSI.

THEOREM 4.3. *In a strong interval order, the live-range of a variable in strong SSI form corresponds to an interval. Moreover, this interval starts at the definition point of the variable and ends at its last use.*

PROOF. This follows from Theorems 4.1 and 4.2. In a reverse strong interval order, which is a preorder DFS of the postdominator tree, the last use of a variable is encountered before all the points it postdominates. Then, the subtrees rooted at its children are completely traversed, one after the other, the last one being the subtree that contains the definition of the variable, as Theorem 4.1 shows. Furthermore, according to Theorem 4.2, all points traversed this way before reaching the variable definition correspond exactly to the live-range of the variable. □

COROLLARY 4.4. *For a procedure in strong SSI form, the interference graph, that is, the intersection graph of the live-ranges, is an interval graph.*

PROOF. This follows immediately from Theorem 4.3. □

4.2 Weak SSI Form

For weak SSI form, the general scheme of block linearization requires both a postdominator tree and a connected minimal loop nesting forest. The construction of the order is similar to the strong interval order: first build an auxiliary order, then do a preorder DFS traversal of the postdominator tree using the previous order, and finally reverse it. The difference is that, during the preorder DFS traversal of the postdominator tree in the second step, one additional constraint is imposed on the order in which children of a given node are processed. Let u, v, and w be children of a given node in the postdominator tree. The two constraints that the order must now satisfy are as follows.

(1) *Constraint on dominance*. If u dom v, then the preorder DFS traversal of the postdominator tree should visit the subtree rooted at v before it visits the subtree rooted at u.
(2) *Constraint on the loop nesting forest*. If u and v belong to the same loop, and w does not, then the subtree rooted at w should not be visited between the subtrees rooted at u and v.

A total order resulting from such a traversal is called a *reverse weak interval order* and its reversal is called a *weak interval order*. Next, we prove that such an order satisfying both constraints exists.

In a minimal loop nesting forest denoted by $\mathcal{L}$, let $\mathcal{F}_{\mathcal{L}}(G)$ be the DAG obtained after removing all loop-edges from G. It has a topological order that respects the nesting of loops, which means that all nodes of a given loop can be visited before visiting any other loop [Ramalingam 2002, Theorem 4]. To see this, we can order the nodes of the loop-tree during its construction: at each level of the decomposition, the children of a loop are sorted according to a topological order of the DAG obtained by removing all loop-edges and considering each resulting strongly connected component (SCC) as a single node.

A topological order of $\mathcal{F}_{\mathcal{L}}(G)$ respects the dominance relation if it is connected, that is, if there is a path from the root r to any other node u. Indeed, in this case, such a path contains any node v that dominates u, thus v is processed before u in any topological order. Lemma 4.5 shows that this occurs when loop-headers are entry nodes, that is, when the minimal loop forest is connected as defined in Section 2.1.3.

LEMMA 4.5. *Consider a CFG with root r from which there is a path to any other node. Then, for each SCC (i.e., loop) L obtained during the construction of a minimal loop nesting forest, the set of entry nodes for L is not empty. This ensures that a connected minimal loop forest can be built. Furthermore, after loop-edges are removed, there is still a path from r to any other node.*

PROOF. Let us recall the construction of a loop nesting forest: each step of the decomposition computes a set of SCCs, which form new loops in the loop nesting forest. Then, for each such loop L, the loop-edges, that is, the edges from a node in L to a loop-header of L, are removed. By induction, we prove that there still exists a path from r, the CFG entry node, to any other node in the CFG after the removal of these edges. For the basis, observe that each CFG node is reachable from r in the initial CFG, prior to the identification of the first set of loops.

Then let L be a loop in the nesting forest. Let G' and G'' denote the CFG before and after the removal of the loop edges of L. By the induction hypothesis, all nodes are reachable, in G', from r. As $r \notin L$, there exists at least one entry node of L, that is, a node in L with an incoming edge from outside of L. Therefore, it is always possible to select a set of loop-headers that are also entry nodes of L, and the decomposition can continue.

Note that any path ending at an entry node u for L and whose previous node v in the path is not in L cannot contain a node in L, except u, otherwise v would also belong to the SCC L. Thus, in G', there is a path from r to any entry node u of L that does not contain any node in L except u. None of the edges along this path are loop-edges of L, so this path remains present in G''. On the other hand, if u is not an entry node of L, consider a path, in G', from r to u. Let v be the last (if any) entry node of L in P. The subpath from v to u does not contain any loop-edges for L and v remains reachable from r in G''. Therefore, concatenating these two paths ensures that a path from r to u exists in G''. □

Lemma 4.5 shows that any topological order of $\mathcal{F}_{\mathcal{L}}(G)$ respects the dominance if $\mathcal{L}$ is a connected minimal loop forest. In other words, when defining a reverse weak interval order, the two constraints used to decide which subtree to traverse next are not contradictory. If the second constraint (the constraint on the loop nesting forest) is satisfied, the first one (the constraint on dominance) will be automatically satisfied. In particular, we call the order given by Ramalingam [2002, Theorem 4] a *loop-dominance-based order*.

Consider the example of Figure 2. As explained in Section 2.1.3, the CFG depicted in Figure 2(a) has two nested loops: $L_1 = \{2, 3, 4, 5, 6, 7, 8, 9\}$ and $L_2 = \{3, 4, 5, 6\}$. Due to the edges between the SCCs encountered during the construction of the loop forest and the nesting of loops, this CFG has only two possible loop-dominance-based orders:

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] \qquad \text{(or 10 and 11 inverted)}.$$

Now, we perform a preorder DFS traversal of the postdominator tree (shown in Figure 1(c)) where the children of each node are visited according to the reverse loop-dominance-based order; in particular, node 8 must be visited before node 6. This provides us with the following reverse weak interval order:

$$[12, 11, 10, 8, 7, 6, 5, 4, 3, 2, 9, 1].$$

Reversing this order yields the following weak interval order:

$$[1, 9, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12],$$

for which, as we will show, the live-range of any variable in weak SSI is an interval.

THEOREM 4.6. *In a weak interval order, the live-range of a variable in weak SSI form corresponds to an interval.*

PROOF. Again, let x denote the variable of interest, d its definition point, u its last use, and LIVE its live-range. Let $\leq$ denote a weak interval order. By Theorem 4.1, we already know that $d \leq p$ for any program point $p \in$ LIVE. To establish that the

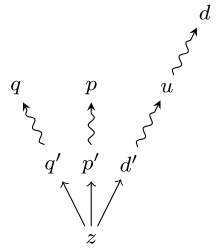


Fig. 6. Postdominance relation between the program points used in the proof.

live-range corresponds to an interval, it remains to prove that $q \in \text{LIVE}$ and $d < p < q$ imply $p \in \text{LIVE}$, for any two points p and q.

The order $\leq$ respects the postdominance, as it is based on a preorder DFS traversal of the postdominator tree; thus $\neg d \,\text{pdom}\, p$ and $\neg p \,\text{pdom}\, q$. If $q \,\text{pdom}\, d$, then $q \,\text{pdom}\, p$ due to the preorder DFS traversal of the postdominator tree, and $p \in \text{LIVE}$ by Lemma 3.1. Similarly, u pdom p implies $p \in \text{LIVE}$ by Lemma 3.1 (recall that $u \,\text{pdom}\, d$, $u \in \text{LIVE}$, and $\neg d$ pdom p). Therefore, we can assume that $\neg u$ pdom p and $\neg q$ pdom d.

Recall that LCA(u, v) denotes the least-common ancestor of u and v in the postdominator tree. Since $p < q$, the subtree rooted at q in the postdominator tree is visited before the subtree rooted at p, from which it follows that $\text{LCA}(d, q) \,\text{pdom}\, \text{LCA}(d, p)$. Let $z = \text{LCA}(d, p)$. Let us define d', p', and q' such that $d' \,\text{pdom}\, d$, $p' \,\text{pdom}\, p$, $q' \,\text{pdom}\, q$, and $\text{ipdom}(d') = \text{ipdom}(p') = \text{ipdom}(q') = z$. Since $\neg u \,\text{pdom}\, p$, it follows that $\neg u \,\text{pdom}\, z$. Since $z \,\text{spdom}\, d$, by definition, and $u \,\text{pdom}\, d$, by the definition of weak SSI form, it follows that u must be on a path in the postdominator tree from z to d, and therefore $d' \,\text{pdom}\, u$. Figure 6 illustrates the postdominance relationship between the various nodes (or program points) that we just introduced. In this figure, paths in the postdominator tree are traversed from left to right when computing a weak interval order.

Since q' and u are in separate branches of the postdominator tree, it follows that $\neg q' \,\text{pdom}\, u$. Since $q \in \text{LIVE}$, Lemma 3.2 shows that $q' \in \text{LIVE}$. Then, using Lemma 3.3, we deduce that $d' \,\text{dom}\, q'$ and $u \,\text{dom}\, q'$. As the dominance relation forms a tree, either $u \,\text{dom}\, d'$ or $d' \,\text{dom}\, u$. If $d' \,\text{dom}\, u$, there is an elementary path from r to q' that goes through d' and then through u, before reaching q'. This establishes the existence of a path from u to q' that does not contain d'. Moreover, since $\neg d' \,\text{pdom}\, q'$, there exists a path from q' to t that does not contain d', which contradicts the fact that $d' \,\text{pdom}\, u$. Therefore, $u \,\text{dom}\, d'$.

Since $q' \in \text{LIVE}$ and $q' \neq d$, there is a path from q' to u such that d dominates each node on this path. The facts that $d \,\text{sdom}\, u$, $u \,\text{dom}\, d'$, and $d' \,\text{dom}\, q'$ imply that there exists a path from u to q' that includes d', and d dominates all points on this path. This, in turn, implies that d', q', and u all belong to a cycle containing nodes that are dominated by d. Let L denote the loop in the loop nesting forest such that the removal of the loop-edges of L breaks this circuit. At least one node on this circuit must be a loop-header for L. Therefore, $d \notin L$, since d dominates each node on the circuit, and a loop-header cannot be dominated by any other node within the loop when it is a loop-entry node.

Now, recall that d', p', and q' are all children of z in the postdominator tree, and that the preorder DFS traversal of the dominator tree, which constructs the reverse weak interval ordering, processes the subtrees rooted at the children of z in reverse loop-dominance-based order. Specifically, p' must occur between q' and d' in the reverse loop-dominance-based order. The fact that the reverse loop-dominance-based order respects the loop nesting property implies that $p' \in L$. Therefore, there exists a path

from p' to u in L, thus it does not contain d, which proves that $p' \in$ LIVE. Finally, Lemma 3.1 shows that $p \in$ LIVE too. □

Theorem 4.6, therefore, proves that the live-range of each variable under weak SSI corresponds to an interval in a total order of CFG nodes. Moreover, this order depends only on the structure of the CFG and not on the relative positions of definitions and uses of variables within the basic blocks of the CFG. This order can thus be precomputed to help with liveness analysis, as described in the following section, and it remains unchanged even if various transformations move or delete instructions, as long as the CFG structure is not modified. Note however that some forms of code motion may require the insertion of additional ϕ-functions and σ-functions in order to preserve weak SSI form.

COROLLARY 4.7. *For a procedure in weak SSI form, the interference graph, that is, the intersection graph of the live-ranges is an interval graph.*

PROOF. This follows immediately from Theorem 4.6. □

The primary difference between strong and weak SSI is the "end" of the interval corresponding to the live-range of each variable. Under strong SSI, the end of the interval is the last use of the variable and it postdominates all points of the live-range. Under weak SSI, the end of the interval may not be a use and it does not necessarily postdominate all points of the live-range. Figure 1 illustrates these points: if a variable is defined in node 1 and used only in node 3, weak SSI does not require the insertion of any ϕ-functions or σ-functions: the initial live-range is not split and the variable is live everywhere in the two loops, that is, its live-range spans all CFG nodes 1 through 9. In particular, there is no point in the live-range that postdominates the entire live-range. The strong interval order proposed in Section 4.1 would be incorrect here because 11 is between 1 and 6, but the variable is not live in 11. With the weak interval order proposed in Section 4.2 for weak SSI, the live-range is an interval, its end point is node 8, and 8 does not postdominate 6.

4.3 Liveness Analysis

Liveness analysis usually computes for each basic block of the CFG its set of *live-out* and *live-in* variables. This information is generally required to perform register allocation, and to efficiently translate a program out of SSA or SSI form while minimizing the number of copies between variables [Budimlić et al. 2002; Sreedhar et al. 1999]. Historically, liveness analysis has been formulated as a dataflow analysis problem. Dataflow analysis can be quite expensive in terms of compilation time; this is especially true when manipulating bit-vectors with a large number of variables. As both SSA and SSI forms increase the number of uniquely defined variables in a procedure, the costs of performing dataflow analysis using these representations must be taken into account when crafting a compiler back-end; this is especially true for just-in-time (JIT) compilers.

In principle, it should be possible to convert minimal or semipruned SSI form to pruned SSI form using dead code elimination. This ensures that all σ-functions and ϕ-functions corresponding to a variable x in the pre-SSI program are placed at points where x was originally live, and liveness analysis is not required to ensure this property. We may still require liveness analysis in SSI form, either to perform register allocation or to eliminate as many copies as possible during SSI elimination. One possibility is to compute liveness analysis prior to building SSI form, and then updating the results to account for the fact that the conversion to SSI partitions each pre-SSI

variable into a set of smaller variables; the other possibility is to perform liveness analysis once the procedure has been converted to SSI form.

This section points out a consequence of Theorem 4.3: under strong SSI form, iterative dataflow analysis is not required. Instead, it suffices to traverse the nodes of the CFG in reverse strong interval order and, within each node, to traverse the instructions in reverse order. During this traversal, a variable is live exactly from the first time one of its uses is encountered (which is guaranteed to be the last use) until its definition is processed. The consequences of this approach are twofold.

(1) If live sets are explicitly required, neither dataflow equations nor union of sets are needed throughout the traversal: the live-out set of a basic block is exactly the live-in set of the previously processed block. Dataflow analysis is reduced to its simplest form: a single pass over the control-flow graph is enough.
(2) Register allocators based on *linear scan* [Poletto and Sarkar 1999] approximate the lifetime of each variable as an interval, rather than an exact live-range. An SSI-based implementation of a linear scan register allocator can compute live intervals exactly without the need to compute live sets explicitly.

This fast approach to liveness analysis does *not* directly work for weak SSI form because there may be no last use that postdominates each point where a variable is live. If the basic algorithm we have outlined is applied, a point where a variable becomes live may be encountered using the reverse weak interval order before the last use is encountered. Instead, we need to identify, for each variable, the last point (in the order) of the largest loop in the loop nesting forest that contains its last use but not its definition. This will be the end of the interval, as the proof of Theorem 4.6 shows. Be careful, this property is not due to the fact that the order of blocks respects the nesting of the loops as this is not true in general for a weak or strong interval order. If this last point is precomputed, the liveness analysis can be performed in a unique pass, as for strong SSI, following the reverse weak interval order. Another solution is to first compute, for each variable, an incomplete interval from its definition to its last use, then to extend it in a second pass, following the weak interval order and exploiting the structure of the loop forest.

Once liveness analysis has been performed, the interval property implies a simple $O(1)$-time query to determine whether a variable x is live at a given point p in a procedure. Under strong SSI form, the endpoints of the interval are the definition point d and the final use u of x; therefore, x is live at p if and only if $d \leq p \leq u$ using a strong interval order. Under weak SSI, one endpoint of the live interval is d, and liveness analysis must be run in advance to determine the other endpoint, which we denote as q. Therefore, x is live at a point p if and only if $d \leq p \leq q$ using a weak interval order. Similarly, computing live-in and live-out sets can be done in one pass, once these intervals are identified.

5. CONCLUDING REMARKS

We first make a few remarks on some of the errors made in previous papers related to SSI form and on some questions that are left open.

5.1 SSI Construction Algorithms

Ananian's algorithm for (strong) SSI construction [Ananian 1999] is based on an auxiliary data structure called the *program structure tree* (PST) [Johnson et al. 1994]. The PST decomposes the control flow graph into a hierarchy of canonical *single-entry single-exit* (SESE) regions. Extending the notion of dominance and postdominance to edges instead of just nodes, a SESE region is formed by a pair of CFG edges (e_i, e_j) if

e_i dom e_j, e_j pdom e_i, and e_i and e_j are *loop-cycle equivalent*, meaning that every cycle in the CFG that contains e_i also contains e_j, and vice-versa. A SESE region (e_i, e_j) is defined to be *canonical* if e_j dom e_k for any SESE region (e_i, e_k) and e_i pdom e_k for any SESE region (e_k, e_j). The hierarchy of canonical SESE regions is based on the following definition of containment: SESE region (e_i, e_j) contains a basic block b if e_i dom b and e_j pdom b.

The paper that introduced the PST stated the following as a theorem: If R_1 and R_2 are two canonical SESE regions of a CFG, then either R_1 and R_2 are node-disjoint, or R_1 is contained within R_2 (or vice-versa). Unfortunately, the proof is incorrect, as shown by the counterexample in Figure 7. In this example, (A, B) and (B, C) are both canonical SESE regions. However, both contain the basic block b, as per the definition of containment provided previously. Therefore, it is impossible to construct a hierarchy of SESE regions, given this definition of containment. As a consequence, the PST appears to be unsound, at least in its current form. It may be possible to fix the definition of the PST using a new definition of containment; however, it is not our goal or intention to do so here.

As a consequence, Ananian's algorithm for SSI construction is incorrect, because it relies on the PST to place ϕ-functions and σ-functions. As it is impossible to construct a PST, as it is defined, for the CFG shown in Figure 7, it is unclear how Ananian's algorithm would behave. Of course, this statement may be revised in the future if a corrected PST emerges.

This leaves us with one correct algorithm to construct SSI form, the algorithm proposed by Singer [2006], and two similar yet distinct versions of SSI form: strong and weak. Although Singer used the weak definition of SSI form, his construction algorithm actually builds the strong version. Singer's algorithm repeatedly performs two steps: insertion of ϕ-functions using iterated dominance frontiers (e.g., SSA construction [Cytron et al. 1991]), and insertion of σ-functions using iterated postdominance frontiers. The algorithm continues to iterate until neither step inserts any further ϕ-functions or σ-functions.

For example, consider the non-SSI CFG in Figure 4(a). The definitions of the variables x and y dominate all of the uses, so no ϕ-functions are inserted. Now, let us consider the basic block within the loop that contains uses of both x and y. By definition, this basic block belongs to its own iterated postdominance frontier, so ϕ-functions for both x and y are inserted at the bottom of the loop; as a consequence of the variables that are defined by these σ-functions the next iteration will instantiate ϕ-functions for x and y at the top of the loop. Looking back at Figure 4(c), it is clear that this corresponds to strong, rather than weak, SSI form, as it satisfies the single upward-exposed-use property.

Thus, we are left without a construction algorithm for weak SSI form. It is not our intention to present such an algorithm in the context of this article. Moreover, it is unclear, immediately, whether there are compiler analyses and optimizations that favor strong SSI over weak, or vice-versa (with the exception of liveness analysis, as discussed in the preceding section). All things being equal, weak SSI is probably preferable, as it requires fewer ϕ-functions and σ-functions than strong SSI. That being said, further differentiation between these two flavors of SSI, and the development of an efficient construction algorithm for weak SSI, are left open for future work.

5.2 Order That Respects Both Dominance and Postdominance

In the preceding section, we mentioned an error in Theorem 1 of the paper that introduced the PST [Johnson et al. 1994]. Although we provided a counterexample that debunked the statement of the theorem, we did not explain precisely what mistakes

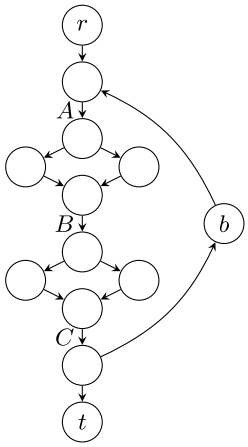


Fig. 7. (A, B) and (B, C) are canonical SESE that have a partial overlap: both contain block b.

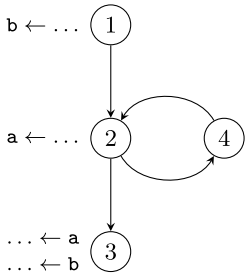


Fig. 8. Node 2 simultaneously dominates and postdominates node 4.

were made by the authors in the proof itself. The theorem itself contains three parts, and parts 1 and 3 wrongly assume that "an edge cannot both dominate and postdominate a node." In the example of Figure 7, the three edges A, B, and C both dominate and postdominate the node (i.e., basic block) b. Actually, reasoning with nodes (and not edges), it is not possible to construct an order of the nodes of a CFG that respects both the dominance and postdominance relations, that is, an order denoted by $\leq$ such that $u \leq v$ if u dom v and $u \leq v$ if v pdom u. Such an order cannot exist because the header node of a for-loop or a while-loop may simultaneously dominates and postdominates every other node in the loop, as shown in Figure 8. Thus, it is impossible to construct an order under which the header occurs both before and after each node in the loop.

This mistake was repeated in a previous paper that incorrectly proved that the interference graph for a procedure in SSI form is an interval graph [Brisk and Sarrafzadeh 2007]. In this case, the theorem remained correct (with the caveat that there are both strong and weak variants of SSI, a fact that has not been observed until now), despite the fact that the proof itself was wrong. That paper included an incorrect algorithm (the procedure PD-DFS) that attempts to build such an ordering; this algorithm is wholly incorrect, and would terminate on the CFG shown in Figure 8 without processing all of the nodes in the CFG.

5.3 Summary

This article has corrected and clarified a number of mistakes in prior literature regarding SSI form and the program structure tree (PST). In particular, we have distinguished between two similar, yet distinct, variants of SSI form, which we denote as strong [Ananian 1999] and weak [Singer 2006]. We have shown that the PST, as it is understood today, is unsound, due to an incorrect definition of containment. This debunks Ananian's algorithm for (strong) SSI construction, which relies on the PST to insert ϕ-functions and σ-functions and construct SSI form. We have also shown that Singer's algorithm for SSI construction actually builds strong SSI form, thus, to date, there is no known algorithm that constructs weak SSI form.

A prior paper had attempted to prove that the interference graph for a program in SSI form (without differentiating between strong and weak variants) is an interval

graph [Brisk and Sarrafzadeh 2007]. The proof in that paper was incorrect, and, in fact, contained the same mistake that led to the incorrect assertion about containment in the paper that introduced the PST [Johnson et al. 1994]. To correct it, we have given a constructive proof that a linearization of basic blocks (and therefore, program points) exists such that all live-ranges of all variables under SSI form corresponds to intervals. The ordering we build for weak SSI is slightly more constrained than for strong SSI, as weak SSI is a generalization of strong SSI. This ensures that, for a procedure in SSI form, the interference graph (more precisely the intersection graph of the live-ranges) is an interval graph. We have also shown that an efficient algorithm exists for liveness analysis under strong SSI that does not rely on iterative dataflow analysis. For weak SSI, an additional computation is required, on the loop forest used to build the order of blocks, so as to identify the end of the interval for each variable (it is the last point, in the order, of the largest loop that contains its last use but not its definition).

A number of questions, both theoretical and empirical, remain open. First, the construction of SSI form, both weak and strong, remains an issue. It is not clear either if weak SSI form is preferrable to strong SSI form, despite the fact that it requires fewer ϕ-functions and σ-functions. Second, it is unclear whether SSI form offers any meaningful benefits over SSA. Although SSA form was widely adopted following its introduction in academic literature, to the best of our knowledge, SSI form has not yet been adopted. The first algorithm for SSA-based sparse conditional constant propagation [Wegman and Zadeck 1991] was far superior to anything that had come before it, and there is a considerable ongoing research effort that is attempting to exploit the chordal interference graph property of SSA form in order to develop better register allocators. In contrast with SSA form, SSI form, to date, has lacked an application where it shows significant advantages and has attracted scant research attention. Given the theoretical results of this article, the next step is clearly to develop an SSI-based register allocator that exploits the interval graph property and possible fast liveness analysis. The existence of a good SSI-based register allocator could increase the profile of SSI form.

ACKNOWLEDGMENTS

We would like to thank Sebastian Hack for the fruitful initial discussions, Pierre Clairambault for his help on the main proof, and Florian Brandner for his careful proofreading. Thanks are also due to the anonymous reviewers for their insightful comments.

REFERENCES

ALPERN, B., WEGMAN, M. N., AND ZADECK, F. K. 1988. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*. ACM, 1–11.

ANANIAN, C. S. 1999. The static single information form. Tech. rep. MIT-LCS-TR-801, Laboratory for Computer Science, Massachusetts Institute of Technology.

APPEL, A. W. AND PALSBERG, J. 2002. *Modern Compiler Implementation in Java* 2nd Ed. Cambridge University Press.

BOISSINOT, B., DARTE, A., DUPONT DE DINECHIN, B., GUILLON, C., AND RASTELLO, F. 2009. Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'09)*. IEEE Computer Society Press, 114–125.

BOUCHEZ, F., DARTE, A., GUILLON, C., AND RASTELLO, F. 2005. Register allocation and spill complexity under SSA. Tech. rep. RR2005-33, LIP, ENS Lyon, France.

BOUCHEZ, F., DARTE, A., GUILLON, C., AND RASTELLO, F. 2006. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? In *Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC'06)*. Lecture Notes in Computer Science, vol. 4382, Springer, 283–298.

BRIGGS, P., D.COOPER, K., HARVEY, T. J., AND SIMPSON, L. T. 1998. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper. 28*, 8, 859–881.

BRISK, P. AND SARRAFZADEH, M. 2007. Interference graphs for procedures in static single information form are interval graphs. In *Proceedings of the 10th International Workshop on Software & Compilers for Embedded Systems (SCOPES'07)*. ACM Press, 101–110.

BRISK, P., DABIRI, F., JAFARI, R., AND SARRAFZADEH, M. 2006. Optimal register sharing for high-level synthesis of SSA form programs. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 25*, 5, 772–779.

BUDIMLIĆ, Z., COOPER, K. D., HARVEY, T. J., KENNEDY, K., OBERG, T. S., AND REEVES, S. W. 2002. Fast copy coalescing and live-range identification. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*. ACM Press, 25–32.

CHAITIN, G. J. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*. ACM, 98–105.

CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. W. 1981. Register allocation via coloring. *Comput. Lang. 6*, 1, 47–57.

COOPER, K. D. AND TORCZON, L. 2004. *Engineering a Compiler*. Morgan Kaufmann.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4, 451–490.

GOLUMBIC, M. C. 2004. Algorithmic graph theory and perfect graphs. In *Annals of Discrete Mathematics*, vol. 57, North-Holland.

HACK, S., GRUND, D., AND GOOS, G. 2006. Register allocation for programs in SSA form. In *Proceedings of the 15th International Conference on Compiler Construction*. A. Zeller and A. Mycroft Eds., Lecture Notes in Computer Science, vol. 3923, Springer, 247–262.

HAVLAK, P. 1997. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst. 19*, 4, 557–567.

JOHNSON, R., PEARSON, D., AND PINGALI, K. 1994. The program structure tree: Computing control regions in linear time. *ACM SIGPLAN Not. 29*, 6, 171–185.

PEREIRA, F. M. Q. AND PALSBERG, J. 2005. Register allocation via coloring of chordal graphs. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*. Lecture Notes in Computer Science, vol. 3780, Springer, 315–329.

POLETTO, M. AND SARKAR, V. 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst. 21*, 5, 895–913.

RAMALINGAM, G. 2002. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst. 24*, 5, 455–490.

ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1988. Global value numbers and redundant computations. In *Proceedings of the 5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*. ACM Press, 12–27.

SINGER, J. 2006. Static program analysis based on virtual register renaming. Tech. rep. UCAM-CL-TR-660, Computer Laboratory, University of Cambridge.

SREEDHAR, V. C., GAO, G. R., AND LEE, Y.-F. 1996. Identifying loops using DJ graphs. *ACM Trans. Program. Lang. Syst. 18*, 6, 649–658.

SREEDHAR, V. C., JU, R. D.-C., GILLIES, D. M., AND SANTHANAM, V. 1999. Translating out of static single assignment form. In *Proceedings of the 6th International Symposium on Static Analysis (SAS'99)*. Springer, 194–210.

STEENSGAARD, B. 1993. Sequentializing program dependence graphs for irreducible programs. Tech. rep. MSR-TR-93-14, Microsoft Research, Redmond, WA.

WEGMAN, M. N. AND ZADECK, F. K. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13*, 2, 181–210.

YANNAKAKIS, M. AND GAVRIL, F. 1987. The maximum k-colorable subgraph problem for chordal graphs. *Inf. Process. Lett. 24*, 2, 133–137.

Received June 2009; revised February 2010; accepted April 2010