SSI Revisited

Benoit Boissinot, Philip Brisk, Alain Darte, Fabrice Rastello

SSI Properties Revisited

LIP Research Report 2009-24

Benoit Boissinot
Compsys team, LIP, UMR CNRS–ENS Lyon–UCB Lyon 1–Inria 5668, France
and
Philip Brisk EPFL, Lausanne, Switzerland
and
Alain Darte
CNRS, Compsys team, LIP, UMR CNRS–ENS Lyon–UCB Lyon 1–Inria 5668, France
and
Fabrice Rastello
Inria, Compsys team, LIP, UMR CNRS–ENS Lyon–UCB Lyon 1–Inria 5668, France

---

The static single information (SSI) form, proposed by Ananian, then in a more general form by Singer, is an extension of the static single assignment (SSA) form. The latter is 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 concerning liveness analysis and representation of live-ranges of variables, which could make SSI appealing for just-in-time compilation. Unfortunately, previous literature on the SSI form is sparse and appears to be partly incorrect. Our paper corrects some of the mistakes that have been made. Our main result is a complete proof that, even for the most general definition of SSI, basic blocks, and thus program points, can be totally ordered so that live-ranges of variables correspond to intervals. This corrects the erroneous proof of Brisk and Sarrafzadeh.

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)

---

1. INTRODUCTION

The static single information (SSI) form is an extension of the static single assignment (SSA) form; the latter is a well-established compiler intermediate representation that has been successfully used for numerous compiler analysis and optimizations. SSI form extends SSA form with several additional properties, for example, making it effective for bi-directional dataflow analysis.

Unfortunately, previous literature on the SSI form is sparse and appears to be partly incorrect; this paper corrects some of the mistakes that have been made. Two different definitions of SSI have been proposed, which we call *weak* and *strong* SSI forms: these definitions are not equivalent, and, in fact, the weak SSI form is a generalization of the strong SSI form, which is more constrained.

The strong SSI form was first proposed by Ananian [Ananian 1999]. Ananian's construction algorithm uses a data structure called the program structure tree

(PST) [Johnson et al. 1994]. Unfortunately, we show that, although the algorithm in [Johnson et al. 1994] to build a particular PST is maybe correct, its general definition is not. Specifically, the PST is defined in terms of single entry single exit (SESE) regions and the authors incorrectly proved that each basic block in the control-flow graph (CFG) belongs to precisely one canonical SESE region. Consequently, the PST, as it is understood today, cannot be built for CFGs that contain basic blocks that belong to multiple canonical SESE regions; it follows that Ananian's strong SSI construction algorithm is incorrect for these CFGs as well.

The weak SSI form was proposed by Singer [Singer 2006], who incorrectly assumed that this definition was equivalent to Ananian's. It is in fact a generalization. Interestingly enough, Singer's SSI construction algorithm actually builds a strong SSI form; thus, the resulting program representation satisfies all of the requirements for the weak SSI form, as well as other additional properties that hold for the strong, but not the weak, SSI form. To date, no algorithm that computes the weak SSI form, but not the strong SSI form, has been proposed. This paper does not attempt to propose such an algorithm, but focuses on understanding the similarities and differences between the two in terms of both their definition and sound theoretical properties.

In recent years, there has been a growing interest in performing register allocation for procedures represented in SSA form. Several research groups independently exploited the fact that the interference graph for a procedure in SSA form is a chordal graph [Pereira and Palsberg 2005; Hack et al. 2006; Brisk et al. 2006; Bouchez et al. 2005]; this result is significant because chordal graphs can be colored optimally in polynomial time. However, the key problems of register allocation in compilers, spilling and coalescing, remain NP-complete, even in SSA form [Bouchez et al. 2006]. As a follow-up to this work, Brisk and Sarrafzadeh [Brisk and Sarrafzadeh 2007] proved that the interference graph for a procedure in SSI form is an interval graph, a subclass of chordal graphs. Interval graphs are significant because the k-colorable subgraph problem, which has some principle similarities to spilling, can be solved in polynomial time for interval graphs, but remains NP-complete for chordal graphs [Yannakakis and Gavril 1987].

Unfortunately, Brisk and Sarrafzadeh's work is based on several incorrect assumptions and some proofs and algorithms presented in their paper are incorrect. One of their mistakes was that they used the weak definition of SSI form, but assumed the existence of properties that hold for strong SSI form. The paper contained other technically incorrect points that are clarified and corrected here. Nonetheless, the primary claim made in their paper remain true: there is a total order of basic blocks for which live-ranges form intervals. As a consequence, if variables are assumed to have different values, the interference graph of a procedure in SSI form is the intersection graph of the live-ranges, thus an interval graph. An additional interesting property is that the suitable order of basic blocks depends only on the structure of the CFG, i.e., not on the live-ranges. Its construction is a bit simpler for strong SSI form than for weak SSI form for which we need to use the concept of loop nesting forest to build a suitable order of basic blocks.

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 anal-

ysis, essentially eliminating the "iterative" aspect of the analysis. This analysis was based on the same incorrect assumption that plagued their proof of the interference graph property. Here, we correct their proof and provide the materials that explain why it holds only for strong SSI form.

2. FOUNDATIONS

2.1 Graphs

2.1.1 *Control-flow graph.* Each procedure is represented as a control-flow graph (CFG). A CFG $G = (V, E, r, t)$ is a directed graph with two distinguished nodes r and t: r is the entry node, with no incoming edge, and t is the exit node, with no outgoing edge. Typically, each node of the CFG represents a basic block, which contains a (possibly empty) totally-ordered list of program points. Thus, by node, we sometimes mean a program point and not just the basic block that contains it.

A *path* P of length k from a vertex u to a vertex v in G is a non-empty 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 $i \in [1..k]$. Implicitly, a single node constitutes a path (a trivial path of length 0) even if it does not have a self-edge.

In the rest of the paper, we assume that the CFG is connected: for each node u, there is a path from r to u, and a path from u to t.

2.1.2 *Dominance/post-dominance.* A node x in a CFG *dominates* a node y if every path from r to y contains x. Furthermore, if $x \neq y$, x strictly dominates y. If x dominates y, we write $x \operatorname{dom} y$ and $x \operatorname{sdom} y$ if the dominance is strict. Node y is the immediate dominator of x, denoted $\operatorname{idom}(x)$, if $y \operatorname{sdom} x$ and there exists no node z such that $y \operatorname{sdom} z \operatorname{sdom} x$. Every vertex other than r has a unique immediate dominator. The immediate dominance relation form a *dominance tree* rooted at r.

The post-dominance relationship is defined similarly. A node x post-dominates a node y if every path from y to t contains x. The concepts of strict post-dominance, of immediate post-dominator, and of post-dominator tree rooted at t are analogous. One way to compute the post-dominance information is to compute the dominance relation on the reverse CFG, i.e., the CFG where all edges are reversed.

2.1.3 *Minimal loop nesting forest.* As mentioned, to build a suitable order of basic blocks, we use the concept of *minimal loop nesting forest* introduced by Ramalingam [Ramalingam 2002]. It can be defined recursively as follows.

—Partition the CFG into its strongly connected components (SCCs). Each non-trivial SCC (i.e., not reduced to a single-node without self-edge) is called a *loop*.

—Within each non-trivial SCC, consider the set of nodes not dominated by any other node of the same SCC. Among these nodes, choose a non-empty subset and call it the *loop-header*.

—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. At each level of the decomposition, create a node for each non-trivial SCC (i.e., each loop). Its

successors in the forest 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). If the computed forest contains more than one tree, a root (corresponding to the entire CFG) is added so that the forest can be represented as a tree. We call this the *loop-tree*. The leaves of this tree are exactly the nodes of the CFG. Internal nodes, labeled by loop-headers, correspond to loops. Note also that a node of a loop-header cannot belong to any deeper loop.

Different loop nesting forests have been proposed by Steensgard [Steensgaard 1993], Sreedhar [Sreedhar et al. 1996], and Havlak [Havlak 1997]. Ramalingam formally proved that all of them are minimal loop nesting forests as defined above; they differ only in their definition of loop headers. As for us, we will use a minimal loop nesting forest where all loop-headers are *entry nodes* of the corresponding loop, i.e., nodes with an incoming edge from outside of the loop. We say that such a loop forest is *connected* for reasons that will be clear later (see Lemma 3.1).

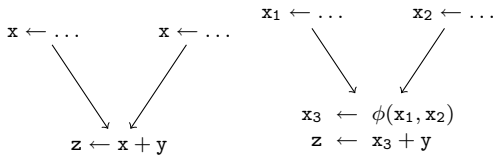
2.1.4 *Liveness and intersection/interference graph.* Consider a variable `x` in a CFG. An instruction that assigns a value to `x` is a *definition* of `x`. An instruction that uses the variable `x` is a *use* of `x`. `x` is *live* at some point p in a CFG if there is a path from a definition of `x` to p and a path, without any definition of `x`, from p to a use of `x`. The *live-range* of `x` is the set of points in the CFG at which `x` is live.

Register allocation uses the notion of interference graph to correctly assign variables to storage locations. Two variables `x` and `y` *interfere* if, at some program point, they are simultaneously live and do not hold the same value. In this case, they cannot be stored at the same location. The *interference graph* has one node per variable and an edge between any two nodes that correspond to interfering variables. A more pessimistic and conservative approach is to assume that simultaneously-live variables always interfere. [1] In this case, the interference graph is just the *intersection graph* of the live-ranges, i.e., there is an edge between two nodes iff (if and only if) the two corresponding live-ranges overlap. This is the assumption we make here: when we speak about the interference graph, we actually mean the intersection graph of the live-ranges. One of our results is that, when variables are in SSI form, this intersection graph is an *interval graph*, i.e., live-ranges can be identified as intervals on a linearly ordered set, e.g., the real line [Golumbic 2004].

2.2 Static single assignment

Static single assignment (SSA) form [Cytron et al. 1991] is a popular program-representation property used in virtually all modern optimizing compilers. An informal definition of SSA form is that each scalar variable is defined only once in the program text (see [Appel and Palsberg 2002; Cooper and Torczon 2004] for details). To construct SSA form, the n definitions of a variable are first replaced by n definitions of n different variables. At control-flow join points, one may have to disambiguate which of the new variables to use. For that, the SSA form uses the abstract concept of ϕ-functions, which select the correct variable depending on the incoming control flow. A ϕ-function defines a new variable that holds the control-flow-disambiguated value. See Figure 1 for an example.

[1] This is the case when the program is strict (see later) and values of variables are not analyzed.



(a) non-SSA program (b) SSA program

Fig. 1: Placement of ϕ-functions

2.2.1 *Single reaching-definition property.* Let $x \neq y$ be two CFG nodes. If there exist two paths from x to z and from y to z, with only z in common, then z is said to be a *join* node for x and y. We write $z \in \mathcal{J}(\{x, y\})$.

Let v be a variable. A program point p is reached by a definition of v if there is a path from this definition to p that does not contain any other definition of v. We say that a code has the *single reaching-definition property* iff no program point can be reached by two definitions of the same variable.

Cytron et al. [Cytron et al. 1991] provide a constructive definition of SSA form that splits live-ranges (by inserting ϕ-functions) at join nodes whenever the single reaching-definition property is not verified. They also make the assumption that each variable has a *pseudo definition* (pseudo-definition property) at the root r of the CFG. With this assumption, they characterize the minimum set of ϕ-functions insertion points by using the *iterated dominance frontier* of a control-flow node x, $\mathcal{DF}^+(\{x\}) = \mathcal{J}(\{x, r\})$. See details in compiler textbooks.

From now on, we say that a code is in SSA according to Cytron et al.'s definition if it has both the pseudo-definition and the single reaching-definition properties.

2.2.2 *Dominance property.* A procedure with a CFG $G = (V, E, r, t)$ is *strict* if every path from r to a use of a variable contains a definition of this variable. This property is always satisfied, implicitly, if the pseudo-definition property is applied. Under SSA, because there is only one (static) definition per variable, strictness is equivalent to the *dominance property*: each use of a variable is dominated by its definition. Furthermore, any program point of the live-range of a variable is dominated by its definition. With this terminology, Cooper and Torczon [Cooper and Torczon 2004] proposed an equivalent definition of SSA, stated in a slightly different way than Cytron et al.'s initial definition: (1) each variable has a unique definition (unique-definition property); (2) the code has the dominance property.

2.2.3 *Minimal and pruned SSA.* Cytron et al. defined *minimal SSA* as SSA with the minimum number of ϕ-functions that satisfies both pseudo-definition and single reaching-definition properties. The drawback of this definition is that it permits the instantiation of *dead* ϕ-functions, i.e., ϕ-functions inserted for a variable x at points in the procedure where x was not originally live in the pre-SSA CFG.

Choi et al. [Choi et al. 1991] suggested that a ϕ-function for a variable x should only be inserted at program points where x was live in the original procedure. The resulting program representation is called *pruned SSA* form and contains significantly fewer ϕ-functions than minimal SSA. It is possible to convert minimal SSA

to pruned SSA through dead code elimination without performing liveness analysis. The existence of such a method to build, without liveness analysis, pruned SSA (this holds for SSI too) is important to not lose the benefit of having an efficient liveness analysis for strong SSI that does not require iterative dataflow analysis.

2.3 Static single information

Static single information (SSI) form is an extension of SSA form that treats uses and definitions symmetrically (however, not all properties are symmetrical).

2.3.1 *Single upward-exposed-use property.* Let $x \neq y$ be two CFG nodes. If there exist two paths from z to x and from z to y, with only z in common, then z is said to be a *split* node for x and y. We write $z \in \mathcal{S}(\{x, y\})$.

Let v be a variable. A use of v is upward-exposed at program point p if there is a path from p to this use that does not contain any other use of v. We say that a code has the *single upward-exposed-use property* iff no program point has two different upward-exposed-uses of the same variable.

Ananian [Ananian 1999] provided a constructive definition of SSI which, in addition to SSA properties, satisfies: (1) each variable has a *pseudo use* at the exit t of the CFG (pseudo-use property); (2) live-ranges are split (by inserting new operators called σ-functions) at split nodes whenever the single upward-exposed-use property is not verified. The σ-function is similar in principle to a ϕ-function. Just as the insertion of ϕ-functions is based on the concept of join sets and dominance frontiers, the insertion of σ-functions when building SSI is based on split sets and, with the pseudo-use property, σ-functions are inserted at the *iterated post-dominance frontier* of a control-flow node x, $p\mathcal{DF}^+(\{x\}) = \mathcal{S}(\{x, t\})$.

From now on, we say that a code is in SSI according to Ananian's definition [2] if it has simultaneously the pseudo-definition, the single reaching-definition, the pseudo-use, and the single upward-exposed-use properties.

2.3.2 *Post-dominance property.* Singer [Singer 2006] proposed a slightly different – and more general – definition of SSI: a procedure is in SSI with *post-dominance property* if it is in SSA form and every use of a variable post-dominates its definition. It is thus characterized by only three properties: the unique-definition property, the dominance property, and the post-dominance property.

2.3.3 *Minimal and pruned SSI.* Similar in principle to SSA form, minimal and pruned variants of SSI exist. The next section discusses the two main SSI definitions, by Ananian and Singer, their similarities and differences.

2.4 Weak and strong SSI

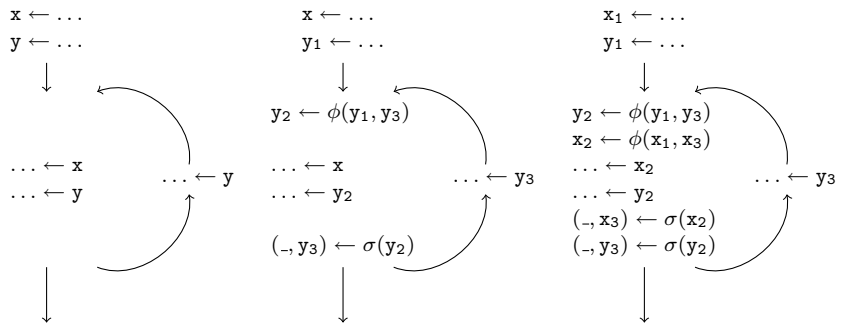
In this paper, we consider only pruned SSI. We call *strong* SSI the SSI form as defined by Ananian, i.e., which satisfies the pseudo-definition and pseudo-use properties, as well as the single reaching-definition and upward-exposed-use properties. We call *weak* SSI the SSI form as defined by Singer, i.e., which satisfies the unique-

[2] Only Ananian's definition is considered here, not Ananian's algorithm, whose correctness is arguable. See the discussion on SESE regions, in the debunking part, Section 4.

definition property, as well as the dominance and post-dominance properties. [3] These two SSI flavors differ as illustrated by the example of Figure 2. For the variable y, both definitions lead to the same code, but hey differ for the variable x. Indeed, in the weak SSI flavor, the definition of x dominates its use, which itself post-dominates its definition. So, there is no need for any ϕ-function or σ-function. However, there are two paths from the end of the loop to a use of x: one along the back-edge to the use of x in the loop, and the other to the exit of the procedure, which contains the pseudo-use. Therefore, the single upward-exposed-use property is not satisfied and, for strong SSI, a σ-function is inserted at the end of the loop body (the post-dominance frontier of the use). This insertion itself implies the insertion of a ϕ-function at the beginning of the loop body (the dominance frontier of the back-edge which contains the definition of the σ-function). Actually, we can prove the following (probably folklore) properties. For defining pruned SSI form,

(1) the single reaching-definition property with the pseudo-definition property is equivalent to the dominance property;
(2) the single upward-exposed-use and pseudo-use properties imply the post-dominance property, but this is not an equivalence as the example illustrates.

Thus, a procedure in strong SSI form is also in weak SSI form. In other words, Singer's definition is a generalization of Ananian's definition. The subtle difference between the single upward-exposed-use property and the post-dominance property is that, in the post-dominance property, only the definition should be post-dominated by the uses while, with the single upward-exposed-use property, *all* program points where a use is upward-exposed should be post-dominated by this use. Indeed, if such a point is not post-dominated, there are two paths from this point, one to the upward-exposed-use, another to the exit node t and its pseudo-use.



(a) non-SSI program (b) weak-SSI program (c) strong-SSI program

Fig. 2: Placement of ϕ-functions and σ-functions

[3] Singer's definition differs from his algorithm, which actually builds strong SSI. Indeed, the algorithm uses the dominance and post-dominance frontiers for ϕ-functions and σ-functions insertion.

3. RESULTS

We now show that there exists a total order, i.e., a linearization, of the CFG nodes such that the live-range of every variable under SSI form is an interval. Furthermore, if the variable is in strong SSI, the last point of the interval is a use. Our linearization order is based on the post-dominance tree, the dominance relation, and, for weak SSI, a connected minimal loop-nesting forest (see Section 2).

3.1 Basic-blocks order for strong SSI

The linearization for strong SSI follows three steps:

(1) Build an order, that we call *dominance-based order*, which respects dominance: if a dominates b then a should appear before b. In other words, a topological order of the dominance tree is required. A pre-order or a reverse post-order of any depth-first search (DFS) of the CFG provide such an order.
(2) Perform a DFS of the post-dominance tree with the additional constraint: visit the set of children of a given node in such a way a child that dominates another child is visited last. In practice, simply visit the children using the reverse of the previously-computed dominance-based order.
(3) Reverse the pre-order obtained by this particular DFS.

Consider the CFG of Figure 3. The dominance tree is shown in Figure 3b. To find a dominance-based order, we can take the pre-order from a DFS of the CFG:

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

Now, perform a pre-order of a DFS of the post-dominance tree (shown in Figure 3c) where the children are visited with a priority given by the reversed dominance-based order (e.g., visit 11 before 8, and 8 before 4). A possible order is the following:

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

Once reversed, we obtain the order:

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

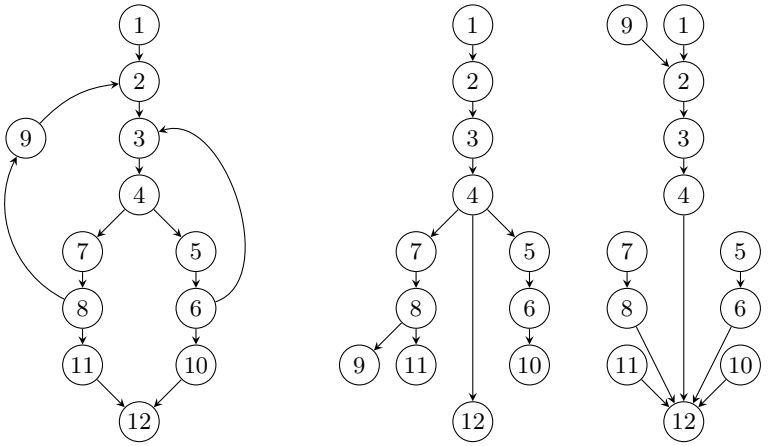
for which, as we show later, the live-range of any variable in strong SSI is an interval. Additionally, this interval will start at the definition and end with one of the uses.

3.2 Order for weak SSI

For weak SSI, the general scheme of linearization is similar to the one for strong SSI. It differs by the fact that the DFS of the post-dominance tree is guided not only by dominance but also thanks to a connected minimal loop nesting forest. In other words, the following constraints are required:

(1) *Constraint on dominance:* if the child a dominates the child b then a should be visited after b.
(2) *Constraint on the loop nesting forest*: if two children a and b belong to the same loop and the child c does not, then c should not be visited between a and b.

We call *dominance-loop-based order* an order of the CFG nodes that follows both the loops nesting and the dominance. We first prove that such an order exists.



(a) Control-flow graph (b) Dominance tree (c) Post-dominance tree

Fig. 3: Example of control-flow graph with its associated dominance and post-dominance trees.

Ramalingam [Ramalingam 2002, Theorem 4] proved that, for every minimal loop nesting forest $\mathcal{L}$, the graph $\mathcal{F}_{\mathcal{L}}(G)$ obtained after removing all loop-edges is a DAG and it has a topological order that respects the nesting of loops, which means that all nodes of a given loop are visited before visiting any other loop. (To see this, order the nodes of the loop-tree during its construction: at each level of the decomposition, sort the children of a loop following a topological order of the DAG obtained after removing loop-edges and considering each resulting SCC as a single node.) Then, a topological order of the DAG $\mathcal{F}_{\mathcal{L}}(G)$ will respect the dominance if it is connected, i.e., if there is a path from the root r to any node x. Indeed, such a path contains any node y that dominates x, thus y is processed before x in any topological order. Lemma 3.1 shows that this is true when loop-headers are entry-nodes, i.e., when the minimal loop forest is *connected* as defined in Section 2.1.3.

Lemma 3.1. *Consider a CFG with root r from which there is a path to any other node. Then, for each strongly connected component (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: at each step of the decomposition, the strongly connected components are computed, which form new loops of the loop nesting forest. For each such loop L, the loop-edges, i.e., the edges from a node in L to a loop-header of L, are removed. We show, by induction, that after each loop-edges removal, there is still a path from r to any other node x.

So, consider a loop L of the loop nesting forest. Denote by G' (resp. G'') the current graph before (resp. after) the removal of the loop-edges of L. By induction hypothesis, in G', all nodes are reachable from r. As $r \notin L$, there exist entry nodes for L, i.e., nodes with an incoming edge from outside L. Thus, a set of loop-headers that are also entry nodes for L can be selected and the construction can continue.

Note that any path ending at an entry node x for L and whose previous node y is not in L cannot contain a node in L, except x, otherwise y would also belong to the SCC L. Thus, in G', there is a path from r to any entry node x of L that does not contain any node in L except x. Such a path still exists in G'' as it contains no loop-edges for L. Finally, if y is not an entry node for L, consider a path, in G', from r to y. Let x be the last (if any) entry node of L in P. The path from x to y does not contain any loop-edges for L and x is still reachable from r in G''. Thus, concatenating the two paths, there is still a path from r to y in G''. □

Lemma 3.1 shows that any topological order of $\mathcal{F}_{\mathcal{L}}(G)$ respects the dominance if $\mathcal{L}$ is a connected minimal loop forest. In particular, the order given by [Ramalingam 2002, Theorem 4] is a dominance-loop-based order.

Consider the CFG of Figure 3. It has two loops: $L_1 = \{2, 3, 4, 5, 6, 7, 8, 9\}$ and $L_2 = \{3, 4, 5, 6\}$. Here, there are only 2 possible dominance-loop-based orders:

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

Now, perform a pre-order of a DFS of the post-dominance tree (shown in Figure 3c) where the children are visited following the reversed loop-dominance-based order (in particular, 8 must be visited before 6). We get the following order:

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

Once reversed, we obtain the order:

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

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

3.3 Proof for strong SSI form

The goal of this section is to prove that, in strong SSI, with an order as defined in Section 3.1, the live-range of each variable forms an interval.

Consider a variable `v` in (strong or weak) SSI and let d be its (unique) definition point. Recall that all its uses post-dominate the definition d and the post-dominance form a tree. So, all uses belong to a path from t to d in the post-dominance tree (be careful, compared to the CFG, paths go in the opposite direction in the post-dominance tree). Thus, one of the uses post-dominates all the other uses: let u denote this "last" use. Finally, let LIVE be the set of program points where `v` is live. By definition, $x \in$ LIVE if, in the CFG, there are both a path from d to x and a path from x to some use of `v` not containing d. Because all uses post-dominate d and u post-dominates all the uses, there is also a path from x to u not containing d. Conversely, if this holds, there is also a path from d to x, otherwise u is not dominated by d. In other words, in (strong or weak) SSI, LIVE is simply the set of points x such that there is a path in the CFG from x to u not containing d.

In the rest of the paper, we write $\neg\mathcal{P}$ to mean that the property $\mathcal{P}$ is not true. With these notations, the main result of this section can be stated as follows:

THEOREM 3.2. *In the reverse pre-order of a post-dominance tree DFS, guided by a reverse dominance-based order, the live-range* LIVE *of a variable v in strong SSI form corresponds to an interval. Moreover,* LIVE *starts at its definition point d and ends at its last-use u.*

The next three lemmas hold even for any variable v, in either weak or strong SSI. Lemma 3.3 states that if v is live at a node x of the post-dominance tree, then it is live at all descendants of x that are not descendants of d, i.e., in the whole subtree rooted at x minus the subtree rooted at d.

LEMMA 3.3. *If $x \in$ LIVE, x pdom y, and $\neg d$ pdom y, then $y \in$ LIVE.*

PROOF. Since $\neg d$ pdom y, there exists a path, in the CFG, from y to t not containing d. Because x pdom y, this path must contain x. So, there is a sub-path which goes from y to x and does not contain d. Since $x \in$ LIVE, there exists a path from x to u not containing d. By concatenating these paths, we can construct a path from y to u that does not contain d, hence $y \in$ LIVE. □

Lemma 3.4 states that if v is live at a node x of the post-dominance tree, then it is live at all ancestors of x that are not ancestors of u, i.e., all nodes in the path of the post-dominance tree leading to x and starting from (this node excluded) the least common ancestor of x and u.

LEMMA 3.4. *If $x \in LIVE$, y pdom x, and $\neg y$ pdom u, then $y \in$ LIVE.*

PROOF. Since $x \in$ LIVE, there is a path, in the CFG, from x to u not containing d. Because $\neg y$ pdom u, there exists a path from u to t not containing y. Concatenating the two paths, we get a two-parts path from x to t via u, where the second part does not contain y, and the first part does not include d. Since y pdom x, this path from x to t must contain y, so y is in its first part, which is thus a path from x to u that contains y but not d. From that, we deduce the existence of a path from y to u not containing d, hence $y \in$ LIVE. □

Lemma 3.5 states that if v is live at a node x of the post-dominance tree, then the ancestors of d that are not ancestors of x dominate x, i.e., x is dominated by all nodes in the path of the post-dominance tree leading to d and starting from (this node excluded) the least common ancestor of x and d.

LEMMA 3.5. *If $x \in$ LIVE, y pdom d, and $\neg y$ pdom x, then y dom x.*

PROOF. Consider a path, in the CFG, from r to x. This path contains d as $x \in$ LIVE, thus d dom x. Since $\neg y$ pdom x, there exists a path from x to t without y. Since y pdom d, y must be included in the path from d to x, otherwise we get a path from d to t without y. This proves y dom x. □

Now we can prove that a depth-first search of the post-dominance tree, guided by reverse dominance to select children, will visit d after all program points where v is live. Thus, the first (in this total order) live point will always be the definition.

THEOREM 3.6. *In a depth-first search of the post-dominance tree, guided by a reverse dominance-based order, the program point of the definition of a variable in weak SSI form is always visited after all the other points where the variable is live.*

PROOF. Consider $x \in$ LIVE. We first exclude the trivial case where d is post-dominated by x in which case, in a DFS, d is always considered after x. So suppose that d is not post-dominated by x. Let us also show that x is not post-dominated by d either. $x \in$ LIVE implies the existence of a d-free path from x to a use. All

uses, and in particular this one, post-dominate the definition, which implies the existence of a d-free path to t. Together, this provides a d-free path from x to t, which proves that x is not post-dominated by d. So, d and x are not comparable for the post-dominance, and we define $a = \mathrm{LCA}(d, x)$ to be the least common ancestor of d and x in the post-dominance tree. Let x' and d' be the corresponding children of a. More formally, $\mathrm{ipdom}(x') = \mathrm{ipdom}(d') = a$, d' pdom d, and x' pdom x.

Since $\neg x'$ pdom d (by construction of x') and u pdom d (by definition of SSI), we have $\neg x'$ pdom u. Now, because x' pdom x (by construction of x') and $x \in$ LIVE, Lemma 3.4 (with $y = x'$) shows that $x' \in$ LIVE. Finally, because d' pdom d (by construction of d') and $\neg d'$ pdom x' (by construction of d' and x'), Lemma 3.5 shows that d' dom x'. Consequently, d' and its descendant d is visited after x' and its descendant x in a DFS of the post-dominance tree if children are selected according to the reverse of the dominance order. □

The next theorem characterizes the live-range of a variable in strong SSI as the descendants of u, in the post-dominance tree, minus the descendants of d.

THEOREM 3.7. *In strong SSI form, the live-range of a variable* v *is the set of nodes post-dominated by the last-use u but not post-dominated by the definition d.*

PROOF. First, let us show that the last-use u post-dominates the live-range. Suppose by contradiction that there exists a path from x to t which does not contain u. Because $x \in$ LIVE, there is also a path from x to u. This means that there are two disjoint paths, one to the last-use u and one to the pseudo-use at t. This would contradict the single upward-exposed-use property of strong SSI. Thus, the whole live-range is included in the set of descendants of u. Finally, Lemma 3.3 states that all descendants of u, except the descendants of d, are in LIVE. □

Theorem 3.6 along with Theorem 3.7 prove Theorem 3.2.

3.4 Proof for weak SSI form

The goal of this section is to extend Section 3.3 to weak SSI form, thus to prove that, in weak SSI, with an order as defined in Section 3.2, the live-range of each variable forms an interval. Note that all results of Section 3.3, except Theorem 3.7 and thus Theorem 3.2, hold even for a variable in weak SSI form.

THEOREM 3.8. *In the reverse pre-order of a post-dominance tree DFS, guided by a reverse dominance-loop based order, the live-ranges of variables in weak SSI form correspond to intervals.*

PROOF. We denote by $\leq$ the total order defined by such a DFS traversal. Thanks to Theorem 3.6, we already know that $d \leq y$ for all $y \in LIVE$. Now, to prove that the live-range corresponds to an interval, it remains to show that if, for any two nodes x and y such that $y \in LIVE$ and $d < x < y$, then $x \in$ LIVE.

Because the order $\leq$ follows the post-dominance order (it is based on a DFS), we know that $\neg d$ pdom x and $\neg x$ pdom y. If y pdom d then, again because the order is constructed from a DFS, y pdom x and, thanks to Lemma 3.3, we have $x \in$ LIVE. Likewise, if u pdom x then using Lemma 3.3 again (remember that $u \in$ LIVE and u pdom d), we have $x \in$ LIVE. So, we can assume $\neg u$ pdom x and $\neg y$ pdom d.

Since $x < y$, the subtree containing y is visited by the DFS before the subtree containing x and thus $\mathrm{LCA}(d, y)$ pdom $\mathrm{LCA}(d, x)$. Let us define d', y', x', and z as follows: $z = \mathrm{LCA}(d, y)$, d' pdom d, y' pdom y, x' pdom x, and $\mathrm{ipdom}(d') = \mathrm{ipdom}(y') = \mathrm{ipdom}(x') = z$. Since $\neg u$ pdom x, we have $\neg u$ pdom z. Since z pdom d (by definition of z) and u pdom d (weak SSI property), u must be between z and d in the post-dominance tree, which means that d' pdom u. Figure 4 illustrates the relationship between the variables regarding the post-dominance relation. Paths on the left are traversed before paths on the right in the DFS traversal.

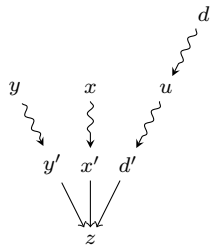


Fig. 4: Post-dominance relation between the program points used in the proof

We know that $\neg u$ pdom y' because y' and u are in separate branches of the post-dominance tree. Thus Lemma 3.4 shows that $y' \in$ LIVE. Then, using Lemma 3.5, we deduce that d' dom y' and u dom y'. As the dominance relation forms a tree, either u dom d' or d' dom u. If d' dom u, there is an elementary path from r to y' that goes through d', then through u, before reaching y'. Thus, there is a path from u to y' not containing d'. Also, because $\neg d'$ pdom y', there is a path from y' to t not containing d'. This contradicts the fact that d' pdom u. Thus, u dom d'.

Since $y' \in$ LIVE, there is a path from y' to u whose nodes are all dominated by d. Since d dom u with $d \neq u$, u dom d', and d' dom y', there is also a path starting at u, going through d', and ending at y', whose nodes are all dominated by d. This means that d', y', and u all belong to a circuit whose nodes are all strictly dominated by d. Let L be the loop of the loop nesting forest whose loop-edges removal breaks this circuit. This means that at least one node of the circuit is a loop-header for L. Therefore, d cannot belong to L because it dominates all nodes of the circuit and a loop-header should not be dominated by any node of the loop. Now, according to the particular DFS definition, since d', x', and y' are all children of the same node z, x' must be between y' and d' in the reverse dominance-loop based order. As the dominance-loop order respects the loop nesting property, x' is also part of the loop L. Therefore, there is a path from x' to u in L, thus not containing d, which proves that $x' \in$ LIVE. Finally, Lemma 3.3 shows that $x \in$ LIVE too. □

Theorem 3.8 thus proves that, even in weak SSI, the live-ranges of the variables can be represented as intervals in a total order of program points. Furthermore, this order depends only on the structure of the CFG and not on the relative positions of definitions and uses of the different variables. It can thus be pre-computed before any liveness analysis. Also, it is unchanged if instructions are moved, inserted, or removed as long as this does not change the CFG structure.

Note however an important difference with strong SSI: for weak SSI, the last point, i.e., the end of the interval, is not necessarily the last-use u and, furthermore, it does not necessarily post-dominate all points of the live-range as illustrated on Figure 4. This is the case for the example of Figure 3 if a variable is defined in node 1 and used only in node 3 (here, no ϕ-function and σ-function need to be introduced in weak SSI). It is then live everywhere in the two loops, thus in nodes 1 to 9. The order proposed in Section 3.1 for strong SSI would be incorrect here because 11 is between 1 and 6, but the variable is not live in 11. With the order proposed in Section 3.2 for weak SSI, the end of the interval is node 8, and 8 does not post-dominate 6.

3.5 Liveness analysis

Liveness analysis usually computes for each basic block of the CFG its set of *live-out* 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 register-to-register copies.

Historically, liveness analysis has been formulated as a dataflow analysis problem. Liveness analysis, and dataflow analysis in general, 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) compilation.

This section points out a nice consequence of Theorem 3.2: liveness analysis is much simpler in strong SSI form. Indeed, consider a DFS traversal of the post-dominance tree, guided by by a reverse dominance-based order. When each basic block is processed, the instructions within it are processed in reverse order. During this traversal, a variable is live exactly from the first time one of its uses is processed until its definition is processed. The consequences are twofold:

(1) If live sets are explicitly required, dataflow equations are not required throughout the traversal: the live-out set of a basic block is exactly the live-in set of the immediate previously-processed block. Dataflow analysis is reduced to its simplest form, and a single pass over the control-flow graph is needed.

(2) Algorithms such as some linear-scan register allocators use live-range intervals directly. In such cases, intervals can be computed directly without any need to build live-sets explicitly.

For weak SSI, this is a bit more complicated because the last-use u is not necessarily the end of the interval representing the live-range. We first need to identify, for each variable, the last point of the largest loop in the loop nesting forest that contains u but not d. Then, for both weak and strong SSI, determining if a variable is live at a given program point can be done with a $O(1)$ query: simply check that the program point is contained in the corresponding interval. Similarly, computing live-in and live-out sets can be done in one pass, once these intervals are identified.

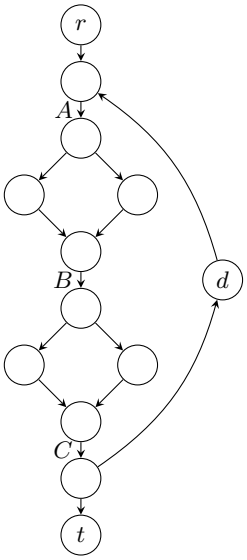


Fig. 5: (A, B) and (B, C) are canonical SESE that have a partial overlap: both contain node d.

4. DEBUNKING

4.1 Foundations of SESE

Most of the original work defining and using SSI is based on single entry single exit (SESE) regions and the program structure tree (PST) [Johnson et al. 1994]. Unfortunately, the theory related to the PST is built on an incorrect claim: the SESE regions, as they are defined, are not necessarily properly nested as wrongly stated in Theorem 1 of [Johnson et al. 1994]. A counter example is given in Figure 5. In this example, (A, B) and (B, C) are SESE regions (according to Definition 3), they are canonical (according to Definition 5). But from Definition 6 (a node belongs to a SESE if it is dominated by its entry edge and post-dominated by its exit edge) both regions (A, B) and (B, C) contain d, which contradicts the proper nesting of SESE regions. In fact, the error roots in the following claim (Proof of Theorem 1, parts 1 and 3): *"an edge cannot both dominate and postdominate a node"*. Here, all edges A, B, and C both dominate and post-dominate node d. A similar mistake was made in [Brisk and Sarrafzadeh 2007]: it was assumed that a node cannot both dominate and post-dominate another node, which is wrong.

4.2 Order that respects dominance

Intuitively, one may think that there must exist an order that respects dominance and for which live-ranges under SSI are intervals. Unfortunately, this intuition is false. This is one of the major mistake that led Brisk et al. [Brisk and Sarrafzadeh 2007] to propose an order based on a DFS traversal. This counter-intuitive result is illustrated by the example of Figure 6. In this strong SSI example, the variable `b` is live after its definition in node 1, in 2 and 4, and in 3 until its use. The variable `a` is live after its definition in node 2 and in 3 until its use. The only order in which

live-ranges are intervals is $[1, 4, 2, 3]$, but it does not respect the dominance (2 dominates 4 while 4 is before 2). However, there always exist an order respecting the post-dominance and for which live-ranges are intervals as Theorem 3.8 proves.

This example, which represents a natural loop, is also interesting because it has a node (node 2) that both dominates and post-dominates another one (node 4). Because of this, the PD-DFS procedure in [Brisk and Sarrafzadeh 2007], which attempts to build an order that satisfies both dominance and post-dominance would get stuck. Such an order does not exist on this example.

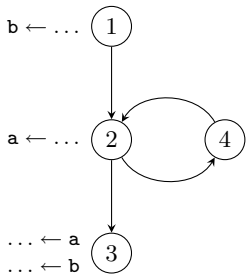


Fig. 6: The only acceptable order is 1432 while 2 dominates 4.

5. CONCLUSION

We have corrected and clarified a number of mistakes in prior literature regarding SSI form and the program structure tree (PST). We have formally distinguished two different variants, the weak and the strong SSI form. The strong SSI form corresponds to Ananian's definition, the weak SSI form is more general and corresponds to Singer's definition.

We proved that there is a linearization of basic blocks (and thus program points) such that all live-ranges of variables in SSI correspond to intervals, even in weak SSI. This result corrects Brisk and Sarrafzadeh's incorrect proof that the interference graph for a procedure in SSI is an interval graph. Using the linearization order, we also showed that single-iteration liveness analysis can be computed for a procedure in strong SSI. This order is also of interest for liveness analysis in weak SSI.

REFERENCES

Ananian, C. S. 1999. The Static Single Information Form. Tech. Rep. MIT-LCS-TR-801, Laboratory for Computer Science, Massachusetts Institute of Technology. Sept.

Appel, A. W. and Palsberg, J. 2002. *Modern Compiler Implementation in Java*, Second ed. Cambridge University Press.

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. Aug.

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 *The 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC'06), November 2-4, 2006, New Orleans, Louisiana*. LNCS. Springer Verlag.

Brisk, P., Dabiri, F., Jafari, R., and Sarrafzadeh, M. 2006. Optimal register sharing for high-level synthesis of SSA form programs. *IEEE Trans. on CAD of Integrated Circuits and Systems 25,* 5, 772–779.

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.

Choi, J.-D., Cytron, R., and Ferrante, J. 1991. Automatic construction of sparse data flow evaluation graphs. In *POPL*. 55–66.

Cooper, K. D. and Torczon, L. 2004. *Engineering a Compiler*. Morgan Kaufmann.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadek, F. K. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems 13,* 4 (Oct.), 451–490.

Golumbic, M. C. 2004. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland.

Hack, S., Grund, D., and Goos, G. 2006. Register Allocation for Programs in SSA form. In Compiler Construction 2006, A. Zeller and A. Mycroft, Eds. *Lecture Notes In Computer Science 3923*.

Havlak, P. 1997. Nesting of Reducible and Irreducible Loops. *ACM Transactions on Programming Languages and Systems 19,* 4, 557–567.

Johnson, R., Pearson, D., and Pingali, K. 1994. The program structure tree: Computing control regions in linear time. *ACM SIGPLAN Notices 29,* 6, 171–185.

Pereira, F. M. Q. and Palsberg, J. 2005. Register Allocation via Coloring of Chordal Graphs. In *Proceedings of APLAS'05*. LNCS, vol. 3780. Springer, 315–329.

Ramalingam, G. 2002. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems 24,* 5, 455–490.

Singer, J. 2006. Static Program Analysis Based on Virtual Register Renaming. Tech. Rep. UCAM-CL-TR-660, University of Cambridge, Computer Laboratory. Feb.

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

Steensgaard, B. 1993. Sequentializing program dependence graphs for irreducible programs. Tech. Rep. MSR-TR-93-14, Microsoft Research, Redmond, Washinghton.

Yannakakis, M. and Gavril, F. 1987. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters 24,* 2, 133–137.