Fast Liveness Checking for SSA-Form Programs

Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoît de Dinechin, Fabrice Rastello

Fast Liveness Checking for SSA-Form Programs

Research Report n°RR-2007-45

Benoit Boissinot
ENS Lyon *
benoit.boissinot@ens-lyon.fr

Sebastian Hack
INRIA*
sebastian.hack@ens-lyon.fr

Daniel Grund †
Saarland University
grund@cs.uni-sb.de

Benoît Dupont de Dinechin
STMicroelectronics
benoit.dupont-de-dinechin@st.com

Fabrice Rastello
INRIA*
fabrice.rastello@ens-lyon.fr

Abstract

Liveness analysis is an important analysis in optimizing compilers. Liveness information is used in several optimizations and is mandatory during the code-generation phase. Two drawbacks of conventional liveness analyses are that their computations are fairly expensive and their results are easily invalidated by program transformations.

We present a method to check liveness of variables that overcomes both obstacles. The major advantage of the proposed method is that the analysis result survives all program transformations except for changes in the control-flow graph. For common program sizes our technique is faster and consumes less memory than conventional data-flow approaches. Thereby, we heavily make use of SSA-form properties, which allow us to completely circumvent data-flow equation solving.

We evaluate the competitiveness of our approach in an industrial strength compiler. Our measurements use the integer part of the SPEC2000 benchmarks and investigate the liveness analysis used by the SSA destruction pass. We compare the net time spent in liveness computations of our implementation against the one provided by that compiler. The results show that in the vast majority of cases our algorithm, while providing the same quality of information, needs less time: an average speed-up of 16%.

Categories and Subject Descriptors F.3.2 [*LOGICS AND MEANINGS OF PROGRAMS*]: Semantics of Programming Languages—Program Analysis; F.3.3 [*LOGICS AND MEANINGS OF PROGRAMS*]: Studies of Program Constructs—Static Single Assignment, SSA

General Terms Algorithms, Languages, Performance

Keywords Liveness Analysis, SSA form, Dominance, Compilers, JIT-compilation

1. Introduction

Liveness analysis provides information about the points in a program where a variable carries a value that might still be needed. Thus, liveness information is indispensable for storage assignment/optimization passes. For instance optimizations like software pipelining, trace scheduling, and register-sensitive redundancy elimination make use of liveness information. In the code generation part, particularly for register allocation, liveness information is mandatory.

Traditionally, liveness information is computed by a data-flow analysis (e.g. see [9]). This has the disadvantage that the computation is fairly expensive and its results are easily invalidated by program transformations. Adding instructions or introducing new variables requires suitable changes in the liveness information: partial re-computation or degradation of its precision. Further, one cannot easily limit the data-flow algorithms to compute information only for parts of a procedure. Computing a variable's liveness at a program location generally implies computing its liveness at other locations, too.

In this paper, we present a novel approach for liveness checking ("is variable v live at location q?"). In contrast to classical data-flow analyses our approach does not provide the set of variables live at a block, only its characteristic function. The results of our analysis remain valid during most program changes and, at the same time, allow for an *efficient* algorithm. Its main features are:

1. The algorithm itself consists of two parts, a *precomputation* part, and an *online* part executed at each liveness query. It is not based on setting up and subsequently solving data-flow equations.
2. The precomputation is *independent of variables,* it only depends on the structure of the control-flow graph. Hence, precomputed information *remains valid* upon adding or removing variables or their uses.
3. An actual query uses the def-use chain of the variable in question and determines the answer essentially by testing membership in precomputed sets.
4. It relies on connections between liveness, dominance, and depth-first search trees, most of them only valid under static single-assignment form (SSA).

SSA is a popular kind of program representation that is used in most modern compilers. Earlier, SSA was only used as an intermediate representation of the program during compilation. Since then, SSA has also been proposed to be used in the backend of a compiler, see [15] for example. Nowadays, there exist several industrial and academic compilers using SSA in their backend, such as LLVM, Java HotSpot, LAO, and Firm. Most recent research on register allocation [3, 6, 12, 16] even allows for retaining the SSA property until the end of the code generation process. Even just-in-time compilers (Java Hot-Spot, Mono, LAO), where compilation

* Université de Lyon, LIP, ENS Lyon

† Partially supported by the German Research Foundation (GK 623)

time is a non-negligible issue, make use of its advantages. As we will see, the special conditions encountered in SSA-form programs make our approach possible at all.

Finally, we rely on the following prerequisites to be met:

- The program is in SSA form and the dominance property must hold.
- The control-flow graph $G = (V, E, r)$ of the program is available.
- The dominance tree of the control-flow graph is available. Otherwise it is computable in $O(|V|)$.
- A depth-first search tree of the control-flow graph is available. Also computable in $O(|V|)$.
- A list of uses for each variable, also known as def-use chain is available. Having an easy-to-maintain def-use chain is one of the major advantages of the SSA form. Hence, def-use chains are often available in SSA-based compilers. Updating the def-use chain when adding or removing uses of a variable incurs virtually no costs, quite contrary to updating liveness information on each change.

As one can see, our assumptions are weak and easy to meet for clean-sheet designs. The SSA requirement is the main obstacle for compilers not already featuring it.

In the next section we give a summary of control-flow graphs, dominance, SSA and liveness. The main contribution is presented in Section 3.3: it introduces the concepts of our approach and presents the main algorithm and its correctness proof. Section 4 provides additional details on optimization and extension of the algorithm. The main focus of Section 5 is implementation efficiency, and Section 6 gives and discusses evaluation results. Finally, Section 7 contrasts this paper with other work, and Section 8 concludes.

2. Foundations

This section introduces the notation used in this paper and presents the theoretical foundations we will use. Readers familiar with flow graphs, depth-first search, dominance and the SSA form can skip ahead to Section 3.3.

2.1 Control-Flow Graphs

A control-flow graph (CFG) $G = (V, E, r)$ is a directed graph with a distinguished node $r \in V$ that has no incoming edge. Normally, the nodes of the CFG are the basic blocks of a procedure each associated with a list of instructions.

Let $G = (V, E, r)$ be a CFG. A *path* $P = (V_P, E_P)$ is an induced subgraph of G for which holds:

$$E_P = \{(v_1, v_2), \ldots, (v_{n-1}, v_n)\} \quad \text{for } V_P = \{v_1, \ldots, v_n\}$$

This explicitly allows for trivial paths containing only a single node. Note, that the existence of a trivial path does not imply the existence of a self-loop in G. If a node v is contained in a path p, we write $v \in p$.

Dominance A node x in a control-flow graph dominates another node y if every path from r to y contains x. The dominance is said to be strict if additionally $x \neq y$. If x dominates y, we write x dom y and x sdom y if the dominance is strict. Further, we denote the set of dominated nodes of some node v by $dom(v)$. We write $sdom(v)$ for $dom(v) \setminus \{v\}$. The nodes of the CFG and the dominance relation form a tree.

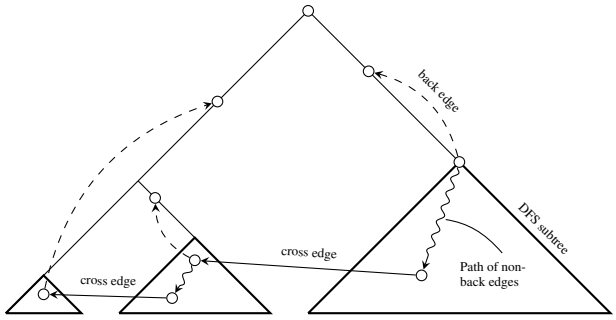


Figure 1: Back edges and cross edges

Depth-first Search A depth-first search (DFS), e.g. see [20], induces a spanning tree on the CFG. Furthermore, it subdivides the edges of the CFG into four classes:

tree edge Edge of the DFS tree.

back edge (u, v) where v is an ancestor of u in the DFS tree. In figures, we will draw back edges with dashed lines.

forward edge (u, v) where u is an ancestor of v in the DFS tree and (u, v) is not a tree edge.

cross edge All other edges.

Figure 1 sketches the different edge types in a DFS. Note that cross edges always point in the "same direction" as they lead to nodes that were already visited but are not ancestors of their source. Since back edges play a major role in this paper, we dedicate some notation to them:

$$E^{\uparrow} = \{(s, t) \in E \mid t \text{ is an ancestor of } s\}$$

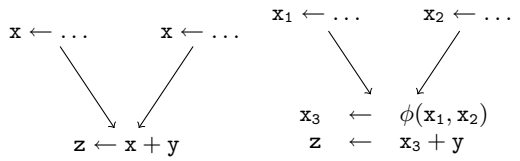
To avoid confusion, parents are called *parents* in the DFS tree and *immediate dominators* in the dominance tree; ancestors are called *(proper) ancestors* in the DFS and *(strict) dominators* in the dominance tree. Clearly, if x dom y then x is also an ancestor of y.

Reducible Control Flow A control-flow graph is called *reducible* if for each back edge (s, t) the target t dominates the source s (see [14]). To create irreducible control flow, loops with multiple entries are necessary. From a language perspective, `gotos` are necessary to create irreducible control flow. Because of its structural properties, the class of reducible control-flow graphs is (and has long been) of special interest for compiler writers. This is because the vast majority of programs (even with explicit use of `gotos`) exhibit reducible CFGs.

2.2 SSA Form

SSA (static single assignment, see e.g. [10]), is a popular program representation property used in many compilers nowadays. In SSA form, each scalar variable is defined only once in the program text. To construct SSA form, the n definitions of a variable are replaced by n definitions of n different variables, first. At control flow join points one may have to disambiguate which of the new variables to use. To this end, the SSA form introduces the abstract concept of ϕ-functions that select the correct one depending on control flow. A ϕ-function defines a new variable that holds the control-flow-disambiguated value. See Figure 2 for an example. We use the following notation: $def(\texttt{a})$ denotes the node in the control-flow graph where variable `a` is defined. Furthermore, $uses(\texttt{a})$ denotes the *set* of all control-flow graph nodes where `a` is used.

In this paper, we will require the program under SSA form to be *strict*. In a strict program with a CFG (V, E, r) every path from



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

Figure 2: Placement of ϕ-functions

r to a use of a variable contains a definition of this variable. Under SSA, because there is only a single (static) definition per variable, strictness is equivalent to the *dominance property*: each use of a variable is dominated by its definition.

Phi-Functions ϕ-functions are somewhat peculiar in terms of how they *use* their operands. Usually, an operation $\mathtt{z} \leftarrow \tau(\mathtt{x}, \mathtt{y})$ is evaluated strictly, i.e. the value of x and y have to be computed in order to compute z. However, ϕ-functions are evaluated lazily. Consider a ϕ-function $\mathtt{z} \leftarrow \phi(\mathtt{x}, \mathtt{y})$. Each operand is associated with a control flow predecessor of the ϕ-function's block. If the ϕ-function's block was reached via its i-th predecessor, the i-th argument of the ϕ-function is assigned to z.

This behavior suggests that the actual assignment is performed "on the way" from the predecessor to the ϕ-function's block i.e. on the corresponding edge. In fact, when leaving SSA, most compilers destruct ϕ-functions by inserting copies in the appropriate predecessor blocks (e.g., see [4]). This implies the following definition:

Definition 1 (Use)**:** *A variable* x *is used at a node* v *if:*

1. *Either* v *contains an instruction* $\ldots \leftarrow \tau(\ldots, \mathtt{x}, \ldots)$ *where* $\tau \neq \phi$*,*
2. *or* v *is the* i*-th predecessor of some node* v' *containing a* ϕ*-function* $\ldots \leftarrow \phi(\ldots, \mathtt{x}, \ldots)$ *where* x *is the* i*-th argument.*

2.3 Liveness

A variable is live at some point if both:

1. its value is available at this point. This can be expressed as the existence of a *reaching definition*, i.e. existence of a path from a definition to this point.
2. its value might be used in the future. This can be expressed as the existence of an *upward exposed use*, i.e. existence of a path from this point to a use that does not contain any definition of this variable.

In fact, the reaching definition constraint is useful only for non-strict programs. In such a case, an upward exposed use at the entry of the CFG is a *potential* bug in the program that usually lets the compiler dump a warning message (use of a potentially undefined variable). With our assumption the program being in strict SSA form (with dominance property), liveness can be defined as follows:

Definition 2 (live-in)**:** *A variable* a *is live-in at a node* q *if there exists a path from* q *to a node* u *where* a *is used and that path does not contain* $def(\mathtt{a})$*.*

Definition 3 (live-out)**:** *A variable* a *is live-out at a node* q *if it is live-in at a successor of* q*.*

3. SSA Liveness Checking

3.1 Overview

We present a decision procedure for the question whether a variable is live-in at a certain control-flow node. To avoid notational overhead we will from now on consider the live-in query of variable a at node q. The CFG node $def(\mathtt{a})$ where a is defined will be abbreviated by d. Furthermore, the variable a is used at a node u. The basic idea of the algorithm is simple. It is the straightforward implementation of Definition 2:

> *For each use* u *we test if* u *is reachable from the query block* q *without passing the definition* d*.*

Our algorithm is thus related to problems such as computing the transitive closure or finding a (shortest) path between two nodes in a graph. However, the paths relevant for liveness are further constrained: *they must not contain the definition of the variable.* Hence, a large part of this paper deals with describing these paths and how their presence (or absence) can be checked efficiently. To this end we split the problem: we search for such a path by trying to incrementally compose it of back-edge-free subpaths. The following section summarizes the basic concepts of our investigation. Section 3.3 presents the algorithm, and Section 3.4 provides its correctness proof.

3.2 Concepts

Simple Paths The first observation considers paths that do not contain back edges. If such a path starts at some node q strictly dominated by d and ends at u, all nodes on the path are strictly dominated by d. Especially, the path cannot contain d. Hence, the existence of a back-edge-free path from q to u directly proves a being live-in at q.

This gives rise to the reduced graph $\widetilde{G}$ of G which contains everything from G but the back edges. If there is a path from q to u in the reduced graph we say that u is *reduced reachable* from q.

To be able to efficiently check for reduced reachability we precompute the transitive closure of this relation. For each node v we store in R_v all nodes reduced reachable from v.

Definition 4 (R_v)**:**

$$R_v = \{w \in V \mid \exists\, \textit{path}\ v \to w\ \textit{in}\ \widetilde{G}\}$$

Paths Containing Back Edges Of course, for the completeness of our algorithm we must also handle back edges: consider Figure 3 and the query "is x live-in at node 10?". Although x is live-in at 10 no use of x is reduced reachable from 10. However, the use of x at 9 is reduced reachable from node 8, which is the target of the back edge $(10, 8)$. If a variable is live-in but no use is reduced reachable there must be some back edge target from which the use is reduced reachable. Consider the second query "is y live-in at 10?". The answer is "yes" but requires more indirection than the previous example. One must traverse the back edge to 8, a tree edge and a cross edge to 6, and finally the back edge reaching the use in 5.

Our goal is to answer a liveness query by testing for the reduced reachability of uses from back edge targets. Hence, a second part of our precomputation constructs for each node q a set T_q that contains all back edge targets relevant for this query. For this precomputation to make sense, these T_q must be independent of variables. Thus, they must contain all relevant back edge targets for *any* variable.

The first question is, given a specific query $(q, \mathtt{a})$, how do we decide which back edge targets of T_q to consider? Apparently, this choice depends on the variable or more precisely on its dominance subtree. Consider again node 10 but now with variable w. All back edge targets (8, 5, 2) are reachable from 10. But if we pick 2 to test if 4 (w's use) is reduced reachable, we get "yes", but obviously w is not live at 10.

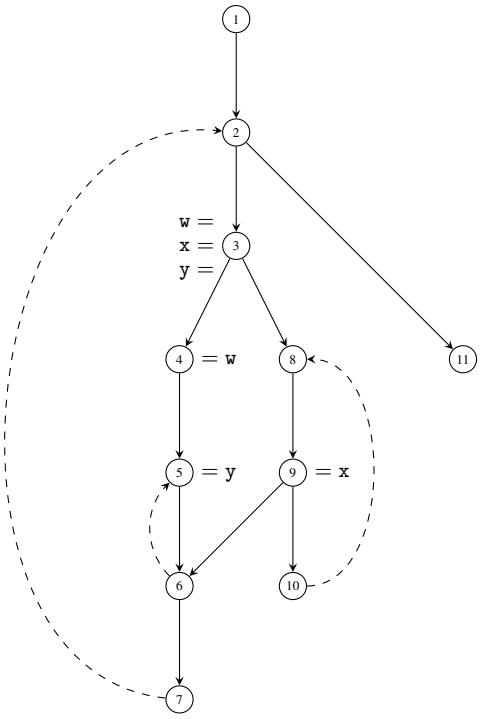


Figure 3: An example CFG

The problem is that 2 is not strictly dominated by $def(\mathtt{w})$. Thus, even if 2 is reachable from 10, reaching 4 from 2 requires passing $def(\mathtt{w})$ since 4 is dominated by 2. Therefore, it is necessary to exclude all back edge targets of T_q that are not strictly dominated by the definition of the variable in question.

However, this condition is not strong enough as we will see in the next example. Assume we want to test for x being live-in at 4. The back edge target 8 is reachable via 4, 5, 6, 7, 2, 3, 8 *and* is inside the dominance subtree of $def(\mathtt{x})$. However, x is not at all live at 4. The problem is that to reach 8 on a path from 4 the path must leave the dominance subtree of $def(\mathtt{x})$ and re-enter it.

The Main Principle The two last examples have in common that the paths first leave the dominance subtree and then re-enter it. Thus, they always contain the definition of the variable and do not comply with the requirements of Definition 2. The dominance subtree however depends on the actual variable whose liveness we are checking. That seems to be contradictory to the statement that we want to precompute the T_q sets *independently* of variables. However, in the following we will show that the T_q sets can be precomputed such that taking the intersection $T_q \cap sdom(def(\mathtt{a}))$ will yield a set of representatives that is suitable for testing a's liveness at q. Therefore, we will construct the T_q such that each $t \in T_q$ is reachable from q along a path that never re-enters *any* dominance subtree once it left it.

Definition 5 (T_q)**:**

$$
\begin{aligned}
T_t^{\uparrow} &= \left\{ t' \in V \setminus R_t \mid \exists s' \in R_t \wedge (s', t') \in E^{\uparrow} \right\} \\
T_q^0 &= \{q\} \\
T_q^i &= \bigcup_{t \in T_q^{i-1}} T_t^{\uparrow} \\
T_q &= \bigcup_{i=0}^{\infty} T_q^i
\end{aligned}
$$

T_q is defined recursively starting from q. To compute T_q^i, the set $T_t^{\uparrow}$ is computed for each back edge target t in the previous set T_q^{i-1}. $T_t^{\uparrow}$ contains exactly those back edge targets

1. whose sources are reduced reachable from t
2. from which t itself is *not* reduced reachable.

Hence, in each step, we will only add back edge targets that will provide new reachability information. Furthermore, this will establish the property mentioned above, as will be shown in Theorem 1. Section 5.2 describes how to compute the T_q efficiently.

3.3 The Algorithm

Now let us give the live-in checking algorithm that relies on the sets R_v and T_v being precomputed for each node v. Regarding a live-in query $(q, \mathtt{a})$, we first construct the set $T_{(q,\mathtt{a})} = T_q \cap sdom(def(\mathtt{a}))$ that contains all nodes of T_q that are strictly dominated by $def(\mathtt{a})$. Note that this set is empty if q is not strictly dominated by $def(\mathtt{a})$. Then we use these nodes in $T_{(q,\mathtt{a})}$ and the precomputed R_v to test for reachability of a use. The pseudocode of this procedure is given by Algorithm 1.

Algorithm 1 Live-In Check

```
1: function IsLiveIn(variable a, node q)
2:     T_(q,a) ← T_q ∩ sdom(def(a))
3:     for t ∈ T_(q,a) do
4:         if R_t ∩ uses(a) ≠ ∅ then return true
5:     return false
```

3.4 Correctness

Before the actual correctness proof, let us give a lemma and a corollary about back-edge-free d-dominated paths, which we will use in that proof.

Lemma 1: *Let d strictly dominate two nodes t and u. If there is a path p from t to u in G that is not strictly d-dominated, then p contains a back edge.*

Proof. Let y be a node of p that is not strictly dominated by d. The only way to reach u from y is via d since d dominates u. So p must contain d. Hence we have that d is reachable from t along p. But t is also reachable from d in $\widetilde{G}$ (d dominates t). So there is a cycle, consisting of a path from t to d (a subpath of p) and a path in $\widetilde{G}$ from d to t. Since the latter part contains no back edge, p must contain a back edge. □

Corollary 1: *If there is a path from t to u in the reduced graph and t and u are strictly dominated by some d, then every node on that path is also strictly dominated by d.*

Now, let us show the correctness of Algorithm 1. First we show the identity of liveness and the existence of strictly dominated paths, and then use this equivalence in the main correctness proof.

Lemma 2: *Variable* a *is live-in at block q if and only if there is a strictly d-dominated path from q to some use u of* a.

Proof. **"⇐"** Straightforward.

"⇒" According to Definition 2, if a is live-in at block q then there exists a path p from q to u that does not contain d. By contradiction: Suppose p is not strictly d-dominated. Then, there exists some node $y \in p$ that is not strictly dominated by d. As u is strictly dominated by d, any path from y to u must contain d.

□

Theorem 1: *Algorithm 1 is complete and sound.*

Proof.

Completeness We have to show that if there is a strictly d-dominated path from q to some use u, the algorithm returns $true$. Considering the loop in line 3 and the check in line 4, we have to show: If there is a strictly d-dominated path from q to u then there is a $t \in T_q$, strictly dominated by d, and u is reduced reachable from t.

Let p be the strictly d-dominated path. In the trivial case that $u \in R_q$ we have $q \in T_q^0$. Otherwise, p is decomposed into subpaths in $\widetilde{G}$ and back-edges $(s_j, t_j) \in E^{\uparrow}$:

$$p = q, \ldots, s_1, t_1, \ldots, s_k, t_k, \ldots, u$$

Without loss of generality, let p be minimal concerning the number of back edges. Suppose, p contains a back edge target t_j that is reduced reachable from t_{j-1}. Then s_{j+1} (or u if $j = k$) is reduced reachable from t_{j-1}, too, which contradicts the assumption that p is a shortest path. Thus, no back edge target in p is ruled out by the definition of $T_t^{\uparrow}$. Hence, by construction, for each $1 \leq i \leq k$: $t_i \in T_q^i$, and in particular $t_k \in T_q$ from which u is reduced reachable.

Soundness Here, we have to show: If the algorithm returns $true$ there exists a strictly d-dominated path from q to some use u. Again, considering Algorithm 1, this is identical to the statement: If there is a t in T_q, strictly dominated by d, from which u is reduced reachable, then there exists a d-dominated path from q to u.

If $t = q$ Corollary 1 applies. In the non-trivial case there is a path

$$p = q, \ldots, s_1, t_1, \ldots, s_k, t_k, \ldots, u$$

such that $(s_i, t_i) \in E^{\uparrow}$, $t_i \in T_q$, and d strictly dominates t_k.

We will show that all sub-paths of p are strictly d-dominated. Since t_k and u are strictly d-dominated, Corollary 1 shows this for that part of p. For the remaining sub-paths we show the property by induction.

base case t_k is strictly d-dominated by premise.

induction step Let t_i be strictly d-dominated.

the part s_i, t_i: Since t_i is strictly dominated by d and s_i is the direct predecessor of t_i, s_i is dominated by d. Hence it rests to prove that $d \neq s_i$. d is a proper ancestor of t_i because it strictly dominates t_i. Furthermore, t_i is an ancestor of s_i. Hence, d is a proper ancestor of s_i and $s_i \neq d$.

the part $t_{i-1}, \ldots, s_i$: Assume t_{i-1} is not strictly d-dominated. Then the path from t_{i-1} to s_i must contain d (d dom s_i). Since this sub-path contains no back edges, d is reduced reachable from t_{i-1}. Additionally, t_i is reduced reachable from d, since d dominates t_i by induction hypothesis. Together, t_i is reduced reachable from t_{i-1} contradicting the definition of $T_q^{\uparrow}$. Thus, t_{i-1} is strictly d-dominated and, again, Corollary 1 applies for the sub-path $t_{i-1} \ldots s_i$.

The remaining sub-path $q, \ldots, s_1$ is covered by thinking of the node q as t_0.

□

4. Further Details

4.1 Ordering the T_q

The number of iterations spent in the loop of Algorithm 1 (line 3) depends on the order in which the elements of T_q are iterated. Consider an iteration of that loop with some t. Trivially, if there has already been an iteration for some $t' \in T_q$ and t' sdom t then the iteration with t will not return $true$, either. This is because t is reduced reachable from t' and thus $R_t \subseteq R_{t'}$. Hence, it makes sense to order the back edge targets by dominance.

For reducible CFGs this order is even optimal, i.e. leads to the earliest exit possible. Furthermore, we show that dominance implies a total order on the T_q for reducible CFGs. Hence there is one $t \in T_{(q,\mathtt{a})}$ that dominates all others. Testing reduced reachability from this t will provide the result of the liveness query, and the loop can be left after the first iteration.

Lemma 3: *If the CFG is reducible, then for all q the dominance relation is a total order on T_q.*

Proof. If a strictly dominates b we say that a is the larger and b the smaller element. To prove the lemma, we will prove by induction that: First, for all i all nodes in some T_q^i are totally ordered by the dominance relation. And second, all nodes in T_q^{i+1} strictly dominate the largest element of T_q^i.

Let us start with T_q^1. Let $t_1 \in T_q^1$ and s_1 be its corresponding source. Because the CFG is reducible, t_1 strictly dominates s_1. By construction s_1 is reduced reachable from q. Hence, because t_1 is not reduced reachable from q (by construction $t_1 \in V \setminus R_q$), t_1 strictly dominates q. Now, because dominance is a tree order and all elements of T_q^1 dominate a common element q, they are totally ordered by the dominance relation.

The induction step from T_q^i to T_q^{i+1} is similar replacing q by the largest element of T_q^i and t_1 by an element of T_q^{i+1}. □

As noticed earlier, for t' sdom t both in $T_{(q,\mathtt{a})}$, if u is reduced reachable from t' then necessarily u is reduced reachable from t. This leads directly to the following theorem:

Theorem 2: *If the CFG is reducible and* a *is live-in at q then there is one unique $t \in T_{(q,\mathtt{a})}$ for which a use is reduced reachable from t. This node dominates all others in $T_{(q,\mathtt{a})}$.*

4.2 Live-Out

Now, let us use the results of the last section to implement checking for variables being live-out. Reconsider the definition of live-out (Definition 3).

Our goal is to prove the presence or absence of a path from a *successor* of q to a use u of a without running the live-in test for all successors. Clearly, if such a path exists, then there exists a *non-trivial* d-dominated path from q to u. Hence, the live-out test is similar to the live-in test but with two special cases:

1. If the query block q coincides with d, then a is live-out at q if and only if a has a use that is not in q. Hence, we can add a simple test; see line 2 in Algorithm 2.
2. Let the query block q be strictly dominated by $def(\mathtt{a})$. Then a is live-out at q if and only if there exists a *non-trivial* strictly $def(\mathtt{a})$-dominated path from q to a use u. The only difference to

the live-in check is that the path must be non-trivial, i.e. it must contain at least one edge. Clearly, if u is reduced reachable form a $t \in T_{(q,\mathtt{a})}, t \neq q$ then the corresponding path is non-trivial. Otherwise, if it is only reduced reachable from q (i.e. $t = q$), then the path is non-trivial only if $u \neq q$ or q is a back edge target (If q is a back edge target, there exists a non-trivial path from q to q). This condition is expressed in Algorithm 2 by the additional clause in line 8.

Algorithm 2 Live-Out Check

```
1:  function IsLiveOut(var a, node q)
2:      if def(a) = q then
3:          return uses(a) \ def(a) ≠ ∅
4:      if def(a) sdom q then
5:          T_(q,a) ← T_q ∩ sdom(def(a))
6:          for t ∈ T_(q,a) do
7:              U ← uses(a)
8:              if t = q and q is no back edge target then U ← U \ {q}
9:              if R_t ∩ U ≠ ∅ then return true
10:     return false
```

5. Practical Considerations

5.1 An Implementation using Bitsets

The liveness checks presented in the last section were discussed rather abstractly using sets and set operations. This section is concerned with the efficiency of a practical implementation. Since the average number of basic blocks is about 36 in our benchmarks (see Section 6) we chose to implement the precomputed sets as bitsets. For a 32-bit machine that makes two machine words per block, which is space- as well as time-efficient. Using bitsets requires a numeration of the objects we want to put into them. The results of Section 4.1 suggest using a preorder numeration of the dominance tree, such that if a node dominates another, it has a smaller number than the other one. The example graph of Figure 3 exhibits such a numeration.

1. When constructing the $T_{(q,\mathtt{a})}$ set we only consider nodes in T_q that are strictly dominated by $def(\mathtt{a})$. Let $num(q)$ be the preorder number of q and $maxnum(q)$ be the largest preorder number in the dominance subtree of q. The preorder numbers of all nodes strictly dominated by q lie in the interval $[num(q), maxnum(q)]$. Hence, we do not have to materialize the set $T_{(q,\mathtt{a})}$. We can simply iterate over T_q, starting at $num(def(\mathtt{a}))$ and stopping at $maxnum(def(\mathtt{a}))$.
2. The numeration will guarantee that dominating nodes have a lower index in the bitset (closer to 0) than dominated nodes. That means, if we traverse the bitset starting at index 0, we will always find the "more dominating" node first. According to Theorem 2, for reducible CFGs it suffices to test the $t \in T_q \cap sdom(def(\mathtt{a}))$ that dominates all the others. By using the proposed numeration, this node is given by the smallest set bit in the range $[num(q), maxnum(q)]$ of the bitset representing T_q.

Algorithm 3 shows the bitset-based liveness check. It is a straightforward implementation of Algorithm 1 using the facts stated above. A fact we discussed in Section 4.1 is used at the end of the while-loop: If we have tested whether u is reduced reachable from a node t, any test from a t' dominated by t yields the same result because t' is reducibly reachable from t. Hence, we can skip t's dominance tree completely and continue with the next node outside of it. The index of this node is obtained by adding 1 to the maximal index in t's dominance subtree. For reducible CFG's, Theorem 2 ensures that the while body is executed at most once. [1]

Algorithm 3 Bitset implementation of the live-in check

```
bool is_live_in(var a, int q) {
  int def     = get_def_block_num(a);
  int max_dom = get_max_num(def);

  if (q <= def || max_dom < q)
    return false;

  int t = bitset_next_set(T[q], def + 1);
  while (t <= max_dom) {
    for (each u in def-use chain of a)
      if (bitset_is_set(R[t], u))
        return true;

    t = get_max_num(t) + 1;
    t = bitset_next_set(T[q], t);
  }

  return false;
}
```

The function `bitset_next_set` searches the next set bit in a bitset starting form the given position (inclusive). It returns the position of the next set bit or `MAX_INT` if no further bit is set.

5.2 Precomputation

Let us briefly discuss how the sets R_v and T_v can be computed efficiently. First, the R_v sets can be computed using a topological order on the reduced graph (which is acyclic). Such a topological order is provided by a reverse postorder numeration created during the DFS on the CFG.

The T_v sets are calculated in a second pass since they rely on the R_v sets to be present (see Definition 5). Consider the following (directed) graph G_T: Let its nodes be the nodes of the CFG. For each node v let its set of successors be $T_v^{\uparrow}$. Clearly, T_v is the set of nodes reachable from v in this graph. Hence, the computation of T_v is similar to a transitive closure on G_T. The following theorem shows more: The graph G_T is acyclic and T_v can be computed by

$$T_v = \{v\} \cup \left[\bigcup_{w \in T_v^{\uparrow}} T_w \right] \tag{1}$$

Theorem 3: *For all $t' \in T_t^{\uparrow}$ the DFS preorder number of t' is smaller than the DFS preorder number of t.*

Proof. First, recall the definitions of back- and cross edges as given in Section 2. A back edge always leads to an ancestor and hence to a node with a *smaller* number. A cross edge always leads to a node already visited in *another* DFS subtree and thus also to a node with a *smaller* number.

Consider some $t' \in T_t^{\uparrow}$ and its corresponding source s' (see Definition 5). If t is an ancestor of s' then t' is a proper ancestor of t. Hence, it has a smaller number than t. If t is not an ancestor of s', s' was reached from t via one (or more) cross edge. Each cross edge leads to a DFS subtree in which each node has smaller numbers than the origin of the cross edge. Hence, s' has a smaller number than t and so has t'. See also Figure 1 for an illustration. □

[1] Of course, in that case the function can be further optimized by replacing the while with an if.

In practice, we first compute the T_v for all back edge targets using a DFS preorder exploiting Equation 1. Then, we compute the set $T_s \setminus \{s\}$ for each back edge source s by taking the union of the T_v sets of their back edge targets. The results of the second part are then propagated through the reduced graph, similar to computing the R_v sets, i.e. using a DFS postorder. Finally, v is added to T_v for each node.

6. Experimental Evaluation

We implemented our algorithm in the LAO open-source VLIW code generator and compared its performance to the available liveness analysis. The LAO code generator is used by STmicroelectronics to complement the Open64 framework in several production compilers. More important, the LAO code generator is also used by an experimental just-in-time compiler for the Common Language Infrastructure (CLI) program representation. For this reason, it has been carefully profiled and tuned.

Our experimental evaluation consists of two parts. A quantitative analysis of the sizes that influence liveness analysis to support our assumptions, and a runtime analysis of both methods in the described environment. We compiled a subset of ten programs of the integer part of the SPEC2000 benchmark suite with the LAO compiler. The benchmarks 252.eon and 253.perlbmk are missing because they use library functions incompatible with our runtime environment. Hence, they could not be compiled without larger modifications. In total 4823 procedures were compiled.

6.1 Quantitative Analysis

The main factors influencing the speed of our algorithm are

- the length of the def-use chain; used in the for loop of Algorithm 3.
- the number of basic blocks since it determines the size of the bitsets T_v and R_v.
- the number of CFG edges since they govern the time to precompute the T_v and R_v.

Table 1 shows the results of the quantitative evaluation: That is statistics about the number of basic blocks and the number of uses per variable for each benchmark program.

The number of uses (i.e. the length of the def-use chain) mainly governs the runtime of the liveness query. About 95% percent of all variables have less than five uses. Even more, over 70% of all variables have only one use. However, there are also cases in which variables have more than 600 uses.

The runtime of the precomputation is governed by the number of edges in the procedure to compile. As CFGs are sparse, the number of edges in a CFG depends linearly on the number of nodes. On average there were 1.3 edges per basic block with a total maximum of 1.9.

72.71% (87.18%) percent of the compiled procedures had less than or equal to 32 (64) blocks. This means that the bitsets T_v and R_v consume two or less machine words for most CFG nodes. Finally, 99.58% had less than 512 blocks, and the largest block count we encountered was 2240.

In total, the benchmarks contained 238427 edges of which 8701 were back edges. We encountered 60 edges whose back edge target did not dominate its source and hence contributed to irreducible control flow. Out of 4823 compiled functions, 7 contained irreducible control flow.

Discussion The structural parameters of the benchmark programs support our assumptions. The def-use chains are shorter than five elements in more than 95% of all cases. That justifies our presumption that the def-use chain is very short and iterating over it in the check is efficient in most cases.

Furthermore, calculating as well as storing the transitive closure of the reduced graph is a feasible approach as the compiled procedures have almost always less than 500 basic blocks. In terms of memory consumption there is a point where our algorithm needs more memory than the native liveness algorithm, which uses an ordered array per basic block to store live-in variables. This break-even point is reached if the number of basic blocks is larger than the size of such an array (measured in bits). Consider the ordered-array approach on a 32-bit architecture: If a variable is represented by a pointer and one assumes an array length of 32 variables then our method needs less storage if the procedure has less than $32 \times 32 = 1024$ blocks. Regarding the block counts given above we can say that this is nearly almost the case. However, for large block counts like 10,000 or more, the quadratic behavior of the precomputation becomes an issue, especially its memory consumption. Section 8 discusses possible solutions to this problem.

Computing and storing the T_v sets is negligible as the amount of back edges is fairly small (about 3.6% of all edges). Hence, future implementations could use sorted arrays instead of bitsets to save space in case of larger CFGs and speed up the loop iteration (by abandoning `bitset_next_set`). Also, the fact that the vast majority of programs exhibit reducible control flow supports our approach.

6.2 Runtime Analysis

We collected our data during the SSA destruction phase of LAO, which uses the third variant of the algorithm of Sreedhar et al. [19]. This algorithm tests interference of certain SSA variables (results and arguments of ϕ-functions) in order to make coalescing decisions. The interference test employed was proposed by Budimlić et al. [7] and uses SSA properties and liveness to determine if two variables interfere. Basically, it decides whether one variable is live directly after the instruction that defines the other one. This allows for circumventing the construction of an interference graph. We used the liveness queries of this algorithm to compare our method with the liveness facility implemented in LAO, which is described next.

The liveness analysis used in the LAO code generator is based on a classic iterative solver whose worklist is a stack. The stack is initialized with nodes that are pushed in CFG postorder. Implementing the worklist by a simple stack was shown to be effective for liveness analysis by Cooper et al. [8]. However, the distinguishing features of the LAO liveness analysis implementation is that it does not rely on bit vectors to implement sets of variables.

First, the universe of the variables to consider is collected in a table prior to liveness analysis. While doing so, variables are assigned dense indices. Second, the local liveness analysis is performed using the sparse sets of Briggs & Torczon [5]. Third, the global liveness analysis relies on sets represented as sorted dense arrays of pointers (to variables). For procedures with many variables, this has proven to be far more memory efficient than data-flow bit-vector implementations. Testing set membership only requires a binary search, which takes logarithmic time in the set cardinality. In case of SSA destruction, liveness information is only needed for the ϕ-related variables. This is exploited in LAO's liveness analysis (for SSA destruction) by ignoring non-ϕ-related variables completely. Table 2 shows the results of the runtime experiments. We ran LAO on the set of benchmark programs mentioned above and measured

1. the time for constructing the data structures (columns "Precomputation"). For our approach ("New") that is calculating the T_v and R_v. For the native liveness ("Native") this consists of computing for each block the set of live variables using data-flow analysis.

	# of Basic Blocks				# of Uses per Variable				
Benchmark	Average	Sum	$\% \leq 32$	$\% \leq 64$	Maximum	$\% \leq 1$	$\% \leq 2$	$\% \leq 3$	$\% \leq 4$
164.gzip	33.35	2735	69.51	85.36	51	65.64	86.38	92.81	95.94
175.vpr	34.45	7752	68.88	84.44	75	70.36	88.90	93.93	96.28
176.gcc	38.96	78666	72.85	86.03	422	73.99	87.81	92.42	94.84
181.mcf	20.31	528	84.61	100.00	46	66.91	83.50	89.33	94.46
186.crafty	69.28	7551	59.63	76.14	620	72.98	90.09	93.85	95.75
197.parser	23.60	7623	84.82	93.49	96	65.12	86.75	94.26	96.62
254.gap	32.89	28020	67.60	87.44	156	70.46	85.95	91.26	94.54
255.vortex	26.46	24425	77.57	90.68	254	65.99	90.80	95.02	96.97
256.bzip2	22.97	1700	78.37	91.89	36	69.89	89.89	94.47	96.17
300.twolf	56.97	10825	59.47	77.36	165	69.71	87.59	93.23	95.92
Total	35.21	169825	72.71	87.18	620	71.30	87.85	92.76	95.31

Table 1: Results of Quantitative Evaluation

2. the total time of all liveness queries (columns "Queries"). For our method that is the running time of Algorithm 3 and for the native liveness this is the lookup of a variable in the set of the corresponding block.

The numbers in the columns "Native" and "New" represent processor clock cycles that were taken by reading the processor's time stamp counter. The machine used for the experiments was a Dell Latitude X300 notebook using a Pentium M processor at 1.4 GHz and 640 MB of main memory running Ubuntu Linux 7.04. Hence, 1000 cycles correspond to 714 nanoseconds. For each group, the column "Spdup" gives the respective speedup. The column "Spdup" in "Both" gives the speedup resulting from the sum of the time spent in the precomputation and query part: # Proc. $\times$ Avg. cycles per proc. $+$ # Queries $\times$ Avg. cycles per query.

Discussion First, consider the precomputation. Our precomputation is about three times faster than the native liveness computation. Note that the native precomputation is already optimized for the SSA destruction pass by considering only ϕ-related variables. We measured that, on average, the live-sets computed by the native algorithm contained 3.16 elements. Our experiments show that its runtime is basically bounded by the number of set insertions and *not* by the number of data-flow iterations. Hence, a full liveness precomputation regarding all variables takes even longer: we measured an average fill ratio of 18.52 elements per set and an average precomputation time of 283403.5 which is 60% higher as in SSA destruction and about 4.7 times slower than our approach. Our precomputation is completely independent of the number of variables since it solely depends of the control-flow graph's structure.

Second, consider the query time. As expected, a liveness query in our approach is slower than a query in the native approach. A query in the native approach is an array lookup using binary search. Even if we assume 32 elements in such an array, the worst case query consists of 5 memory lookups on an array that is in the cache with a high probability. In our approach, we have bitset lookups and a traversal of the def-use chain that is not as cache-local as an array. We measured that our query is on average about 2.8 times slower than the set lookup of the native approach. Given the number of queries, we compensate this by the faster precomputation. Considering all the benchmarks, there were, on average, 5.19 queries per variable. However, in the case of 186.crafty, there were 26.53 queries per variable, which consumed more time than was gained by the faster precomputation.

7. Related Work

Liveness analysis has mostly been treated in the context of data-flow analysis. Data-flow analysis goes back to the 1960s and is thus very well explored. Much research in iterative data-flow analysis was dedicated to efficiently solve the data-flow equations. There exist approaches for determining efficient node orderings, exploiting structural properties of the program, and using more efficient data structures to accelerate the solvers. We will not discuss this in further detail as this is extensively covered in almost every available compiler textbook. For example, [9] gives a good overview of the seminal work in the area.

Gerlek et al. [11] use so-called λ-operators to collect upward exposed uses at control-flow split points. Precisely, the λ-operators are placed at the iterated dominance frontiers, computed on the reverse CFG, of the set of uses of a variable. These λ-operators and the other uses of variables are chained together and liveness is efficiently computed on this graph representation. The technique of Gerlek et al. can be considered as a precursor of the live variable analysis based on the Static Single Information (SSI) form [18]. In both cases, insertion of pseudo-instructions guarantee that any definition is post-dominated by a use.

The only liveness analysis we are aware of that relies on SSA properties is given in [2]. Similarly to our work, the algorithm uses the fact, that a variable can only be live inside the dominance subtree of its definition. It then uses the def-use chain to search all blocks lying on paths from the variable's definition to a use. The variable must be marked live at each of these blocks. Since it uses the def-use chain, there is no need to traverse the instructions inside a basic block. Hence, the algorithm's runtime corresponds exactly to the number of set insertion operations. Furthermore, it can be run on each variable separately. However, this method only differs from data-flow approaches in how the analysis data is computed and not how it is represented. Hence, it is as vulnerable to program modifications as the data-flow approaches.

8. Conclusions and Outlook

We presented a novel approach to liveness checking for SSA-form programs. In contrast to the existing data-flow based techniques, our analysis data solely depends on the CFG's structure and exploits the properties of the SSA form. Hence, adding, modifying, or removing an instruction does not invalidate our precomputed data, in contrast to prior approaches. This makes our approach especially attractive to compiler phases where keeping liveness information up to date is considered too expensive. Although being of quadratic complexity concerning the number of basic blocks, our benchmarks show that for procedure sizes encountered in our benchmark it is at least three times faster than data-flow based methods.

This acceleration of the precomputation of course has its price: The actual liveness check is slower than an ordinary set lookup. Hence, the performance of our approach strongly depends on the

	Precomputation				Queries				Both
		Avg. cycles per proc.				Avg. cycles per query			
Benchmark	# Proc.	Native	New	Spdup	# Queries	Native	New	Spdup	Spdup
164.gzip	82	174000.82	55054.62	3.12	90659	86.84	162.23	0.53	1.16
175.vpr	225	116963.18	54291.50	2.17	55670	85.71	179.38	0.48	1.41
176.gcc	2019	205923.64	67310.79	3.03	1109202	88.17	339.54	0.26	1.00
181.mcf	26	65544.73	35696.62	1.85	2369	84.09	190.37	0.44	1.39
186.crafty	109	437037.94	156418.57	2.78	858121	81.07	166.14	0.49	0.73
197.parser	323	85194.79	40392.45	2.13	38719	86.54	177.81	0.49	1.54
254.gap	852	191000.39	55515.27	3.45	245540	87.38	168.82	0.52	2.08
255.vortex	923	71444.18	42651.30	1.67	88554	85.09	187.21	0.45	1.32
256.bzip2	74	137544.10	40178.87	3.45	10100	95.00	184.86	0.51	2.32
300.twolf	190	446186.87	94197.44	4.76	184621	94.89	193.81	0.49	1.92
Total	4823	177655.50	60375.69	2.94	2683555	86.09	241.06	0.36	1.16

Table 2: Results of the Runtime Experiments

number of queries. We experimentally show that for the SSA destruction in LAO the number of queries is sufficiently low to outperform the highly tuned, data-flow based native liveness algorithm of LAO. As work in progress, we verify the competitiveness of our approach in other passes/optimizations, which exhibit a different query behavior than SSA destruction.

Our technique uses structural properties of the CFG and could take advantage of a precomputed loop nesting forest [17, 13]. In fact, our algorithm can be adapted to most loop nesting forest definitions. For the sake of brevity and generality, we did not elaborate this further. Furthermore, due to its quadratic nature, memory consumption becomes an issue for procedures with some thousand blocks. Studying more memory efficient ways of storing the transitive closure (e.g. see [1]) is subject to further investigation.

References

[1] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient Implementation of Lattice Operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

[2] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.

[3] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. 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, 2006.

[4] Preston Briggs, Keith D.Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software: Practice and Experience*, 28(8):859–881, July 1998.

[5] Preston Briggs and Linda Torczon. An Efficient Representation for Sparse Sets. *ACM Letters on Programming Languages and Systems*, 2(1-4):59–69, 1993.

[6] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial Time Graph Coloring Register Allocation. In *14th International Workshop on Logic and Synthesis*. ACM Press, 2005.

[7] Zoran Budimlić, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast Copy Coalescing and Live-Range Identification. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 25–32. ACM Press, 2002.

[8] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. Iterative Data-Flow Analysis, Revisited. Technical Report TR04-100, Rice University, 2002.

[9] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

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

[11] M. Gerlek, M. Wolfe, and E. Stoltz. A Reference Chain Approach for Live Variables. Technical Report CSE 94-029, Oregon Graduate Institute of Science & Technology, 1994.

[12] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register Allocation for Programs in SSA form. In Andreas Zeller and Alan Mycroft, editors, *Compiler Construction 2006*, volume 3923. Springer, March 2006.

[13] Paul Havlak. Nesting of Reducible and Irreducible Loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, 1997.

[14] M. S. Hecht and J. D. Ullman. Characterizations of Reducible Flow Graphs. *J. ACM*, 21(3):367–375, 1974.

[15] Allen Leung and Lal George. Static Single Assignment Form for Machine Code. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 204–214, New York, NY, USA, 1999. ACM Press.

[16] Fernando Magno Quintao Pereira and Jens Palsberg. Register Allocation via Coloring of Chordal Graphs. In *Proceedings of APLAS'05*, volume 3780 of *LNCS*, pages 315–329. Springer, November 2005.

[17] G. Ramalingam. On Loops, Dominators, and Dominance Frontier. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 233–241, New York, NY, USA, 2000. ACM Press.

[18] Jeremy Singer. Static Program Analysis Based on Virtual Register Renaming. Technical Report UCAM-CL-TR-660, University of Cambridge, Computer Laboratory, February 2006.

[19] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating Out of Static Single Assignment Form. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, pages 194–210, London, UK, 1999. Springer-Verlag.

[20] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.