Computing Liveness Sets for SSA-Form Programs

Florian Brandner, Benoit Boissinot, Alain Darte, Benoît Dupont de Dinechin, Fabrice Rastello

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Computing Liveness Sets for SSA-Form Programs

Florian Brandner — Benoit Boissinot — Alain Darte — Benoît Dupont de Dinechin — Fabrice Rastello

N° 7503

Janvier 2011

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--7503--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Computing Liveness Sets for SSA-Form Programs

Florian Brandner* , Benoit Boissinot* , Alain Darte* ,
Benoît Dupont de Dinechin† , Fabrice Rastello*

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet COMPSYS

Rapport de recherche n° 7503 — Janvier 2011 — 25 pages

Abstract: We revisit the problem of computing liveness sets, i.e., the set of variables live-in and live-out of basic blocks, for programs in strict SSA form. Strict SSA form guarantees that the definition of a variable always dominates all its uses. This property can be exploited in order to optimize the computation of liveness information.

Our first contribution is the design of a fast data-flow algorithm, which, similar to traditional approaches, first traverses the control-flow graph in postorder propagating liveness information backwards. A second pass then traverses the loop-nesting forest, propagating liveness information within loops. Due to the properties of strict SSA form, the iterative calculation of a fixed-point can be avoided.

Another maybe more natural approach is to identify, one path at a time, all paths from a use of a variable to its unique definition. Such a strategy was proposed by Appel in his "Tiger book" and is also used in the LLVM compiler. Our second contribution is to show how to extend and optimize algorithms based on this idea to compute liveness sets one variable at a time using adequate data structures.

Finally, we evaluate and compare the efficiency of the proposed algorithms using the SPECINT 2000 benchmark suite. The standard data-flow approach is clearly outperformed by the other algorithms, showing substantial speed-ups a factor of 2 on average. Depending on the underlying set implementation either the path-based approach or the loop-forest-based approach provides superior performance.

Key-words: Liveness Analysis, SSA form, Compilers

This work is partially supported by the compilation group of STMicroelectronics.

* UMR 5668 CNRS - INRIA - COMPSYS - LIP - ENS de Lyon - UCB Lyon
† Kalray

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Calcul des Ensembles de Vivacité des Programmes en Forme SSA

Résumé :

Nous ré-examinons le problème du calcul des ensembles de vivacité, c'est-à-dire des ensembles de variables en vie en entrée et sortie des blocs de base d'un programme en forme SSA (assignation unique statique) stricte. La forme SSA stricte garantit que la définition d'une variable domine toujours toutes ses utilisations. Nous exploitons cette propriété pour optimiser le calcul de vivacité.

Notre première contribution est la conception d'un algorithme de calcul de type flot de données qui, de même que les approches traditionnelles, propage les informations de vivacité en remontant le flot d'un parcours en profondeur. Un deuxième parcours propage, cette fois-ci en descendant une hiérarchie de boucles (loop-nesting forest), l'information dans les boucles. Par ce procédé et du fait de la propriété de la forme SSA stricte, le calcul itératif traditionnel pour obtenir un point fixe est inutile.

Une autre approche, peut-être plus naturelle, est d'identifier, un chemin à la fois, tous les chemins remontant de l'utilisation d'une variable jusqu'à sa définition. Une telle approche a été mentionnée par Appel dans son "Tiger book" et est également utilisée dans le compilateur LLVM. Notre seconde contribution est de montrer comment étendre et optimiser un algorithme basé sur cette idée pour calculer les ensembles de vivacité, une variable à la fois, et avec des structures de données adéquates.

Finalement, nous évaluons et comparons les performances des algorithmes proposés avec les benchmarks de SPECINT 2000. L'approche traditionnelle de flot de données est clairement surpassée par les autres algorithmes, avec un facteur d'amélioration de 2 en moyenne. Selon l'implantation des ensembles de vivacité, la meilleure approche est soit celle par remontée de chemin, soit celle utilisant la hiérarchie de boucles.

Mots-clés : Calcul des ensembles de vivacité, Assignation unique statique, Compilateur

1 Introduction

Static Single Assignment (SSA) is a popular program representation used by most modern compilers today. Initially developed to facilitate the development of high-level program transformations, SSA form has gained much interest in the scientific community due to its favorable properties that often allow to simplify algorithms and reduce computational complexity. Today, SSA form is even adopted for the final code generation phase [20], i.e., the backend. Several industrial and academic compilers use an SSA representation in their backends, e.g., LLVM, Java HotSpot, LAO [12], and LibFirm. Recent research on register allocation [5, 8, 13, 23] even allows to retain SSA form until the very end of the code generation process. Also just-in-time compilers, such as Java HotSpot, Mono, and LAO, where compilation time is a non-negligible issue, make use of its advantages.

This work investigates the use of SSA properties in order to simplify and accelerate *liveness analysis*, i.e., an analysis that determines for all variables the set of program points where the variables' values are eventually used by subsequent operations. Liveness information is essential to solve storage assignment problems, eliminate redundancies, and perform code motion. For instance, optimizations like software pipelining, trace scheduling, register-sensitive redundancy elimination, as well as register allocation heavily rely of liveness information.

Traditionally, liveness information is obtained by data-flow analysis: Liveness sets are computed for all basic blocks and variables in parallel by solving a set of data-flow equations [3]. These equations are usually solved by an iterative work-list algorithm, propagating information backwards through the control-flow graph (CFG) until a fixed-point is reached and the liveness sets stabilize. The number of iterations depends on the control-flow structure of the considered program, more precisely, on the structure of the program's loops.

In this paper, we present a novel non-iterative data-flow algorithm to compute liveness sets for SSA-form programs. The properties of SSA form are exploited in order to avoid the expensive iterative fixed-point computation. Instead, at most two passes over the CFG are necessary. The first pass computes partial liveness sets by traversing the CFG backwards – very similar to traditional data-flow analysis. The second pass refines the partial liveness sets and computes the final solution by traversing the loop-nesting forest – as defined by Ramalingam [25]. We first present an algorithm for reducible CFGs (Section 4.1). Irreducible CFGs can be handled by a slightly modified variant of this algorithm (Section 4.2). Since our algorithm exploits advanced program properties some prerequisites have to be met by the input program and the compiler framework:

- The control-flow graph of the input program is available.
- The program has to be in strict SSA form
- The loop-nesting forest of the CFG is available.

These assumptions are weak and easy to meet for clean-sheet designs. The SSA requirement is the main obstacle for compilers not already featuring it.

For SSA programs, another – maybe more natural – approach is possible that follows the classical definition of liveness: A variable is live at a program

point p, if p belongs to a path of the CFG leading from a definition of that variable to one of its uses without passing through another definition of the same variable. Therefore, the live-range of a variable can be computed using a backward traversal starting on its uses and stopping when reaching its (unique) definition. Such a strategy is used in the LLVM[1] compiler and in Appel's "Tiger book" [3]. Our second contribution (Section 5) is to show how to extend and optimize these algorithms to compute liveness sets using adequate data structures for SSA- and non-SSA-form programs. The efficiency of the resulting algorithms is compared with our novel non-iterative data-flow algorithm.

Our experiments using the SPECINT 2000 benchmark suite in a production compiler demonstrate that the non-iterative data-flow algorithm outperforms the standard iterative data-flow algorithm. However, depending on the program characteristics and the underlying representation of the liveness sets either the approach based on path exploration or the two-pass data-flow shows favorable execution times.

Before detailing our two-passes data-flow algorithm (Section 4) and the path-based algorithm (Section 5), we recall in Section 3 some concepts that form the theoretical underpinning of the presented algorithms. Experiments are described in Section 6 and we conclude in Section 7.

2 Related Work

Literature treating specifically the problem of liveness computation is rare. The general approach is to use iterative data-flow analysis, which goes back to Kildall [19]. The algorithms are, however, not specialized to the computation of liveness sets, and may thus incur considerable overhead. Kam et al. [18] explored the complexity of round-robin data-flow algorithms, i.e., those iterating using a fixed order until the analysis result stabilizes. They showed that the number of iterations for data-flow problems on reducible graphs is bound by $d(G) + 3$, where $d(G)$ denotes the *loop connectedness* of the control-flow graph G. Liveness is a backward data-flow problem, the CFG is thus reversed, which frequently leads to irreducible graphs. Empirical results by Cooper [10] indicate that the order in which basic blocks are processed is critical and has a direct impact on the iteration bound.

An alternative way to solve data-flow problems is interval analysis [2] and other elimination-based approaches [27]. Unfortunately, those approaches are often restricted or to complex – as before general data-flow problems are considered. The initial work on interval analysis [2] demonstrates how to compute liveness information using only three passes over the *intervals* of the CFG. However, the problem statement involves, besides the computation of liveness sets, several intermediate problems, including separate sets for reaching definitions and upward-exposed uses. Furthermore, the number of intervals of a CFG grows with the number of loops. Our non-iterative data-flow algorithm in contrast requires at most two passes over the CFG.

Another approach to compute liveness was proposed by Appel [3]. Instead of computing the liveness information for all variables at the same time, he proposes to handle each variable individually by exploring paths through the CFG starting from variable uses. An equivalent approach using logic programming

[1] `http://www.llvm.org/`

was presented by McAllester [21]. He shows that liveness analysis can be preformed in time proportional to the number of instructions and variables of the input program. However, the analysis is based on an input language with simple conditional branches having at most two successors only. A more generalized analysis will be given later in Section 5.4.

3 Foundations

This section introduces the notations used throughout this paper and presents the theoretical foundations necessary for various proofs. Readers familiar with flow graphs, loop-nesting forest, dominance, and SSA form can skip ahead to Section 4.

3.1 Control Flow and Loop Structure

A *control-flow graph* $G = (V, E, r)$ is a directed graph, with nodes V, edges E, and a distinguished node $r \in V$ with no incoming edges. Usually, the CFG nodes represent the basic blocks of a procedure or function, every block is in turn associated with a list of operations or instructions.

Paths Let $G = (V, E, r)$ be a CFG. A *path* P of length k from a node u to a node v in G is a non-empty sequence of nodes $(v_0, v_1, \ldots, v_k)$ 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 forms a (trivial) path of length 0. We assume that the CFG is connected, i.e., there exists a path from the root node r to u for every node u.

Dominance A node x in a CFG *dominates* another node y if every path from r to y contains x. The dominance is said to be strict if, in addition, $x \neq y$. A well-known property is that the transitive reduction of the dominance relation forms a tree, the *dominator tree*.

Loop-Nesting Forest A minimal definition of loop-nesting forests was given by Ramalingam [25]. It can be defined recursively as follows:

1. Partition the CFG into its strongly connected components (SCCs). Every non-trivial SCC, i.e., those not consisting of a single node, is called a *loop*.

2. 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*.

3. Remove all edges, inside the SCC, that lead to one of the loop-headers. Call these edges the *loop-edges*.

4. Repeat this partitioning recursively for every SCC after removing its loop-edges. The process stops when only trivial SCCs remain.

This decomposition can be represented by a forest, where each non-trivial SCC, i.e., every loop, is represented by a node. The children of a loop's node represent all inner loops as well as the regular basic blocks of the loop's body. The forest can easily be turned into a tree by introducing an artificial root node, corresponding to the entire CFG. Its leaves are the nodes of the CFG, while internal nodes, labeled by loop-headers, correspond to loops. Note also that a node of a loop-header cannot belong to any inner loop.

Reducible Control Flow A CFG is *reducible* if, for each loop, its loop-header is a singleton that dominates all nodes of the loop [15]. In other words, the only way to enter a loop is through its unique loop-header. Because of its structural properties, the class of reducible control-flow graphs is of special interest for compiler writers. Indeed, the vast majority of programs exhibit reducible CFGs.

Computing the Loop-Nesting Forest The loop-nesting forest of a reducible CFG is unique and can be computed in $O(|V| \cdot \log^*(|E|))$. Tarjan's algorithm [29] performs a bottom up traversal in a depth-first search tree of the CFG, identifying inner (nested) loops first. A loop is defined as a set of nodes from which there is a path to the source of a back-edge that does not go through its target. It is chased backward from the source of the back-edge target. The back-edge is the loop-edge of the loop, the target is the loop-header that represents the loop's nodes.

Because irreducible loops have more than one undominated node, the loop-nesting forest of an irreducible graph is not unique [25]. An interesting and simple-to-engineer loop-nesting forest algorithm is the one of Havlak [14], later improved by Ramalingam [24] to fix a complexity issue. Havlak's algorithm is a simple generalization of Tarjan's algorithm. It identifies a loop as a set of descendants of a back-edge target that can reach its source. In that case, the loop-header is restricted to a single entry node, the target of a back-edge. During the process of loop identification, whenever an entry node that is not the loop-header is encountered, the corresponding incoming edge (from a non-descendant node) is replaced by an edge to the loop-header.

3.2 Static Single Assignment Form

Static Single Assignment Form (SSA) [11], is a popular program representation used in many compilers nowadays. In SSA form, each scalar variable is defined only once statically in the program text. To construct SSA form, variables having multiple definitions are replaced by several new *SSA-variables*, one for each definition. A problem appears when a use in the original program was reachable from multiple definitions. The new variables need to be disambiguated in order to preserve the program's semantic. The problem is solved by introducing ϕ-functions that are placed at control-flow joins. Depending on the actual control flow a ϕ-function defines a new SSA-variable by selecting the SSA-variable corresponding to the respective definition of the original program.

In this paper, we require that the program under SSA form is *strict*. In a strict program, every path from the root r to a use of a variable contains the definition of this variable. 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.

3.3 Liveness

Liveness is a property relating program points to sets of variables which are considered to be *live* at those program points. Intuitively, a variable is considered live at a given program point when its value is used in the future by any dynamic execution. Statically, liveness can be approximated by considering paths through the control-flow graph leading from uses of a given variable to its

definitions - or in the case of SSA form to its unique definition. The variable is live at all program points along those paths.

Definition 1. A variable a is live-in at a CFG node q if there exists a directed path from q to a node u where a is used and that path does not contain the definition of a, denoted as def_a.

Definition 2. A variable a is live-out at a node q if it is live-in at least at one successor of q.

In practice, it is sufficient to consider only the live-in and live-out sets of basic blocks, since liveness within a basic block is trivial to recompute given the corresponding live-out set. The computation of those sets is usually termed *liveness analysis*.

Live-ranges are closely related to liveness. Instead of associating program points with sets of live variables, the live-range of a variable specifies the set of program points where that variable is live. It has been shown that live-ranges in programs under strict SSA form exhibit certain useful properties, some of which have been exploited for register allocation [8, 13, 23, 6], some of which can be exploited during the computation of liveness information. However, the special behavior of ϕ-operations often causes confusion where exactly the ϕ's operands are actually used and defined.

For regular operations it is clear, variables are used and defined at the program point of the respective operation. The semantic of ϕ-functions (and in particular the actual place of ϕ-uses) should be defined carefully, especially when dealing with SSA destruction. In all algorithms for SSA destruction, such as [7, 26, 28, 4], a use in a ϕ-operation is considered live somewhere inside the corresponding predecessor block but, depending on the algorithm and, in particular, the way copies are inserted, it may or may not be considered as live-out for that predecessor block. Similarly, the definition of a ϕ-operation is always considered to be defined at the beginning of the block but, depending on the algorithm, it may or may not be marked as live-in for the block.

These subtleties need to be taken into account when building liveness sets. In order to make the discussion and the description of algorithms easier, we follow the definition by Sreedhar [28]: Each use of a ϕ-operation is considered as live-out of the corresponding predecessor block; the variable defined by the ϕ-operation is considered to be live-in for the block where it is defined. The algorithms presented in this work require minor modifications when other ϕ-semantics are desired.

4 Data-Flow Approaches

A well-known and frequently used approach to compute the live-in and live-out sets of basic blocks is backward data-flow analysis [3]. The liveness sets are given by a set of *equations* that relate the *upward-exposed uses* and the *definitions* occurring within a basic block to the live-in and live-out sets of the predecessors and successors in the CFG:

$$\begin{aligned} \text{LiveIn}(B) &= \text{UpwardExposed}(B) \cup (\text{LiveOut}(B) - \text{Definitions}(B)) \\ \text{LiveOut}(B) &= \cup_{S \in \text{succs}(B)} \text{LiveIn}(B) \end{aligned}$$

A use is said to be upward-exposed when a variable is used within a basic block and no definition of the same variable precedes the use locally within that basic block. The set of upward-exposed uses and definitions does not change during liveness analysis and thus needs to be precomputed only once.

The equations of the data-flow analysis can be solved efficiently using a simple iterative work-list algorithm that propagates liveness information among the basic blocks of the CFG. The liveness sets are refined on every iteration of the algorithm until a fixed-point is reached, i.e., the algorithm stops when the sets cannot be refined any further. When the work-list contains edges of the CFG, the number of set operations can be bounded by the number of CFG-edges E and the number of variables V by $O(E \cdot V)$ [22]. The *round robin* algorithm [16, 18] allows another bound to be derived based on the *loop connectedness* of the control-flow graph G denoted by $d(G)$. The algorithm traverses the complete CFG on every iteration and thus results in $O(E \cdot d(G))$ set operations. Depending on the program structure either of the two algorithms leads to a faster termination. In addition, a precomputation is necessary in order to compute the upward-exposed uses and definitions of the basic blocks. This can be performed in $O(I \cdot V)$ by visiting every instruction of the program once. Assuming that the set operations can be performed in time proportional to $s(V)$, this gives a complexity bound of $O(I \cdot V + E \cdot V \cdot s(V))$ or $O(I \cdot V + E \cdot d(G) \cdot s(V))$.

Although the traditional data-flow approach is appealing due to its simplicity and flexibility,[2] it exhibits two major shortcomings: (1) The algorithm computes the liveness for all variables in parallel using expensive set operations, (2) computing the fixed-point solution possibly requires several iterations. A solution for the former weakness has been proposed by Appel [3]. He suggests to compute liveness by explicitly exploring paths through the CFG starting from variable uses backwards until the definition of the variable is reached. The variable is added to the live-in and live-out sets along the discovered paths, the use of expensive set operations is replaced by simple set insertions. We have extended and adopted this approach using suitable data structures. Furthermore, SSA form facilitates certain operations during liveness calculation leading to simpler and faster algorithms. We have also investigated how to exploit SSA form in order to avoid the second shortcoming. Indeed, the expensive iteration during traditional data-flow analysis can be avoided, thanks to the dominance property of strict SSA form.

4.1 Liveness Sets On Reducible Graphs

Instead of computing a fixed-point, liveness information can be derived in two passes over the CFG by exploiting properties of strict SSA form. The first version of the algorithm requires the control-flow graph to be reducible, however,

[2]The data-flow analysis can be generalized in order to solve other analysis problems besides liveness.

Algorithm 1 Non-iterative liveness computation for reducible flow graphs.

```
1: function COMPUTE_LIVESETS_SSA_REDUCIBLE(CFG)
2:     for each basic block B do
3:         unmark B
4:     Let R be the root node of the CFG
5:     DAG_DFS(R)
6:     for each root node L of the loop-nesting forest do
7:         LoopTree_DFS(L)
```

we will later extend the basic approach and show that arbitrary flow graphs can be handled elegantly. The algorithm proceeds in two steps:

1. A backward pass propagates partial liveness information upwards using a postorder traversal of the CFG.

2. The partial liveness sets are then refined by traversing the loop-nesting forest in a forward pass, propagating liveness from loop-headers down to all basic blocks within loops.

Algorithm 1 shows the necessary initialization and the high-level structure to compute liveness in two-passes. The postorder traversal is shown by Algorithm 2 which performs a simple depth-first search and associates every basic block of the CFG with partial liveness sets. The algorithm roughly corresponds to the precomputation step of the traditional iterative data-flow analysis. However, loop-edges are not considered during the traversal (line 2). Recalling the definition of liveness for ϕ-operations from Section 3.3, PhiUses(B) denotes the set of variables live-out of basic block B due to uses by ϕ-operations in B's successors. Similarly, PhiDefs(B) denotes the set of variables defined by a ϕ-operation in B.

The next phase, traversing the loop-nesting forest, is shown by Algorithm 3. The live-in and live-out sets of all basic blocks within loops are unified with the liveness sets of the respective loop-headers. This is sufficient in order to compute valid liveness information due to the fact that the live-ranges of variables crossing a back-edge of a loop have to be live in all basic blocks of that loop.

Algorithm 2 Computing partial liveness sets using a postorder traversal

```
1: function DAG_DFS(block B)
2:     for each S ∈ CFG_succs(B) such that (B, S) is not a loop-edge do
3:         if S not processed then DAG_DFS(S)
4:     Live = PhiUses(B)
5:     for each S ∈ CFG_succs(B) such that (B, S) is not a loop-edge do
6:         Live = Live ∪ (LiveIn(S) − PhiDefs(S))
7:     LiveOut(B) = Live
8:     for each program point p in B, backward do
9:         remove variables defined at p from Live
10:        add uses at p in Live
11:    LiveIn(B) = Live ∪ PhiDefs(B)
12:    mark B as processed
```

Algorithm 3 Propagating live variables within loop bodies.

```
1: function LoopTree_DFS(node N of the loop forest)
2:     if N is a loop node then
3:         Let B_N = Block(N)                          ▷ The loop-header of N
4:         Let LiveLoop = LiveIn(B) − PhiDefs(B_N)
5:         for each M ∈ LoopTree_succs(N) do
6:             Let B_M = Block(M)        ▷ The loop-header or basic block of M
7:             LiveIn(B_M) = LiveIn(B_M) ∪ LiveLoop
8:             LiveOut(B_M) = LiveOut(B_M) ∪ LiveLoop
9:             LoopTree_DFS(M)
```

4.1.1 Complexity

In contrast to iterative work-list algorithms, our algorithm consists of only two phases that traverse the CFG and the loop-nesting forest exactly once. The number of operations performed during the CFG traversal of Algorithm 2 can be bounded by $O(E{\cdot}s(V){+}I{\cdot}V)$, where E is the number of flow edges in the CFG and I the number of instructions in the program. The function itself is executed for every basic block of the program, bounding the number of executions for the statements outside of the three for-loops. The body of the two first loops, starting at line 2 and 5 respectively, are executed exactly once for every edge in the CFG. Similarly, the third loop is executed once for every instruction in the program. Since every instruction potentially references all variables, an additional factor has to be accounted for.

The traversal of the loop-nesting forest follows a similar pattern. The size of the forest is at most twice the number of basic blocks B in the program's CFG, because every loop node in the loop-nesting forest has at least one child node representing a basic block. The loop body is executed exactly once for ever node of the loop nesting forest. Giving a bound in the number of set operations for Algorithm 3 of $O(B)$.

This results in an overall bound of $O(E \cdot s(V) + I \cdot V)$ for the number of operations performed by our non-iterative data-flow approach, depicted by Algorithm 1. Note that the initialization phase can be neglected since the CFG is assumed to be connected; and thus $E \geq B$.

4.1.2 Correctness

The first pass propagates the liveness sets using a postorder traversal of the *reduced graph* of the CFG, denoted by $\mathcal{F}_{\mathcal{L}}(G)$, where all loop-edges are removed. We first show that this pass correctly propagates liveness information to the loop-headers of the original CFG.

Lemma 1. *In a reducible CFG, given a variable* v *and its definition* d*, for every maximal loop* L *with header* h *such that* L *does not contain* d*,* v *is live-in at* h *iff there is a path in* $\mathcal{F}_{\mathcal{L}}(G)$ *not containing* d *from* h *to a use.*

Proof. Given a variable v defined in d, and a maximal loop L, with header h, not containing d. If v is live at h, there exists a cycle-free path from h to a use of v in the CFG which does not go through d. Take this path and suppose there is a loop-edge (s, h') in this path, h' being the header of a loop L', and $s \in L'$.

It follows, $h' \neq h$, otherwise the path would contain a cycle, consequently this means $L \neq L'$.

- The header h cannot be in L', since L was the biggest loop not containing d. This would imply $d \in L'$ and h would dominate d, which contradicts the fact that v is live-in at h.
- Also, $h \notin L'$ is impossible, since the graph is reducible. The only way to enter L' is through h', which means there has to be a previous occurrence of h' in the path. This breaks our hypothesis that the path is cycle-free.

The path does not contain any loop-edges and thus is a valid path in $\mathcal{F}_{\mathcal{L}}(G)$. Conversely, if there exists a path in $\mathcal{F}_{\mathcal{L}}(G)$, then v is live-in at h, since $\mathcal{F}_{\mathcal{L}}(G)$ is a sub-graph of the original graph G. □

In the previous lemma, it is not guaranteed there exists a loop L satisfying the conditions. The following lemma covers this case:

Lemma 2. *In a reducible CFG, given a variable* v*, its definition* d*, and a program point* p*, such that all loops containing* p *also contain* d*,* v *is live-in at* p *iff there is a path in* $\mathcal{F}_{\mathcal{L}}(G)$ *not containing* d *leading from* p *to a use.*

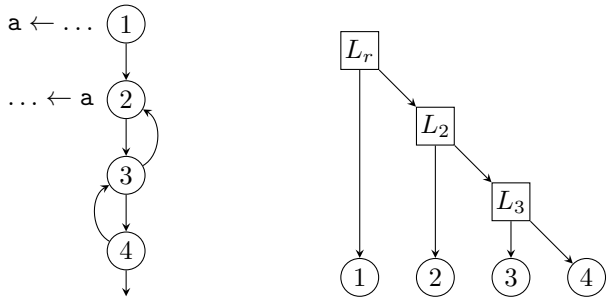
Proof. Given a reducible flow graph G, a variable v defined by d, and a program point p, such that all loops containing p also contain d, assume v is live at p. Since v is live at p, there exists a cycle-free path from p to a use of v in G that does not go through d. Take this path and suppose there is a loop-edge (s, h) in this path, h being the header of a loop L, and $s \in L$:

- It is impossible that $p \in L$. Since this would imply $d \in L$ and further that h would dominate d.
- It is also impossible that p is not in L. Remember that G is reducible and the only way to enter the loop L is through h. Since s is in the loop, we know there has to be a previous occurrence of h on the path. This breaks our hypothesis that the path is cycle-free.

It follows that the path cannot contain any loop-edges. The path is thus a valid path in $\mathcal{F}_{\mathcal{L}}(G)$. Conversely, if there exists a path in $\mathcal{F}_{\mathcal{L}}(G)$, then v is live-in at p, since the $\mathcal{F}_{\mathcal{L}}(G)$ is a sub-graph of the original graph G. □

Those two lemmas prove that if we propagate the liveness information on the reduced CFG, every program point will have a variable v in its live-in set when the program point is either (1) not part of a loop not containing the definition of that variable, or (2) is the header of the biggest loop not containing its definition.

Furthermore, the first lemma proves that if, after the first phase of our algorithm, a program point's live-in set is not accurate, then the missing variables are guaranteed to be in one of the live-in sets of the headers of the surrounding loops.



(a) Control-flow graph (b) Loop-nesting forest

Figure 1: A pathological case for iterative data-flow analysis.

Lemma 3. *If a variable* v *is live-in at a header of a loop, then* v *is live-in at every node within the loop.*

Proof. Given a loop L with header h, such that the variable v defined at d is live-in at h. Since v is live at h, d is not part of L. This follows from the dominance property, i.e., h is strictly dominated by d. Furthermore, there exists a path from h to a use of v which does not go through d. For every node of the loop, p, since the loop is a strongly connected component of the CFG, there exists a path, consisting only of nodes of L from p to h. Concatenating those two paths proves that v is live-in and live-out of p. □

This lemma proves the correctness of the second pass, which propagates the liveness information within loops. Lemma 1 proved that every program point, which is not yet associated with accurate liveness information, will properly be updated by the second pass. The last lemma, furthermore, proves that the variables propagated by the second pass indeed have to be marked live-in. Overall, this proves the correctness of our algorithm.

Example 1. Consider the example program shown in Figure 1. The CFG depicted on the left is a pathological case for iterative data-flow analysis. The initial precomputation phase is not able to determine that variable a is live throughout the two loops. Later on, an additional iteration is required for every loop-nesting level until the final solution is computed. The first phase of our loop-forest-based algorithm has a similar limitation. However, the subsequent traversal of the loop-nesting forest – shown in Figure 1b – efficiently propagates the missing liveness information from the loop-header of loop L_2 down to all blocks within the loop's body and all inner loops, i.e., blocks 3 and 4 of L_3. □

4.2 Liveness Sets on Irreducible Flow Graphs

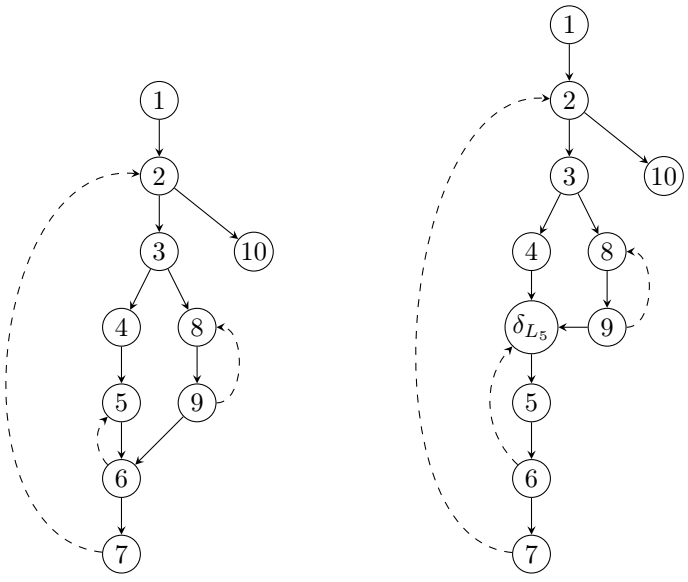
It is well-known that every irreducible CFG can be transformed into a semantically *equivalent* reducible flow graph – for example using node splitting [17, 1]. Unfortunately, the resulting graph may grow exponentially during the processing [9]. However, when liveness information is to be computed, a relaxed notion of equivalence is sufficient. We will show in the following that every irreducible CFG can be transformed into a reducible flow graph such that the liveness in both graphs is *equivalent*.

For every loop L, *EntryEdges*(L) denotes the set of entry-edges leading from a basic block that is not part of the loop to a block that is. *Entries*(L) denotes the set of L's entry-nodes, i.e., the nodes which are the target of an entry-edge. Similarly, *PreEntries*(L) denotes the set of blocks that are the source of an entry-edge. The set of loop-edges is given by *LoopBackEdges*(L). Given a loop L from a graph $G = (V, E, r)$, we define the graph $\Psi_L(G) = (E', V', r)$ as:

$$\begin{aligned} V' &= V \cup \{\delta_L\} \\ E' &= E - \mathit{LoopBackEdges}(L) - \mathit{EntryEdges}(L) \\ &\quad \cup \{p \to \delta_L | p \in \mathit{PreEntries}(L)\} \\ &\quad \cup \{p \to \delta_L | \exists (p \to x) \in \mathit{LoopBackEdges}(L)\} \\ &\quad \cup \{\delta_L \to h | h \in \mathit{Headers}(L)\} \end{aligned}$$

The graph is extended by a new node δ_L, which represents the loop-header of L after the transformation. All edges entering the loop from preentry nodes are redirected to this new header. The loop-edges of L are similarly redirected and additional edges are inserted leading from δ_L to L's former loop-headers.

Repeatedly applying this transformation yields a reducible graph, which may be larger then the original graph depending on the initial number of loops and the number of loop-headers. Furthermore, entry-edges may be updated several times during the processing in order to reach their final positions. Consequently, this transformation is potentially memory and time consuming.



(a) Irreducible Graph (b) Reducible Graph

Figure 2: Deriving a reducible CFG from an irreducible one. The two graphs are not necessarily equivalent in terms of semantics, but liveness is ensured to be equivalent.

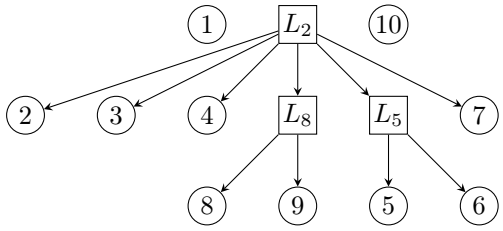


Figure 3: A loop-nesting forest associated with the CFG of Figure 2.

Example 2. Consider the flow graph shown in Figure 2a and the corresponding loop-nesting forest depicted in Figure 3. Loop L_5 can be entered via node 5 and 6 via the preentry nodes 4 and 9 respectively. The CFG is thus clearly irreducible. By transforming the graph according to the algorithm presented above, a reducible flow graph – see Figure 2b – can be derived. The graph might not reflect the semantics of the original program during execution, but preserves the liveness properties of the original graph. □

Fortunately, it is not necessary to construct the transformed graph explicitly. An elegant alternative is to modify the postorder traversal of the first phase of our algorithm as follows. Let $\mathsf{HnCA}(t, s)$ be the loop-header of the *highest non common ancestor* of t and s, i.e., of the highest ancestor of t in the loop forest that is not an ancestor of s. Whenever an entry-edge $s \to t$ is encountered during the traversal, we do not follow the edge. Instead, the highest non common ancestor of the edge's target t and its source s is visited. This is the representative of the largest loop containing the target, but not containing the source. It is sufficient to replace the occurrences of S by $\mathsf{HnCA}(B, S)$ in the for-loops of Algorithm 2 (line 3 and 6) in order to handle irreducible flow graphs.

The modification follows in fact the approach for the construction of Havlak's loop forest: An edge that leads to an entry-node that is not a loop-header is replaced by an edge to the loop-header. Notice that for Havlak, an edge might be replaced several times, until it points to the loop-header of the largest loop that does not contain its source. For an initial edge (n, s), the loop-header of the largest loop that contains s but not n is represented as $h = \mathsf{HnCA}(n, s)$.

4.2.1 Complexity

The changes to the original forest algorithm are minimal and only involve the invocation of HnCA to compute the highest non common ancestor. This function solely depends on the structure of the loop-nesting forest, which does not change. Assuming that the function's results are precomputed the complexity results obtained previously still hold. The highest non common ancestors can easily be derived by propagating sets of basic blocks from the leaves upwards to the root of the loop-nesting forest using a depth first search. This enumerates all basic block pairs exactly once at their respective least common ancestor. Since the overhead of traversing the forest is negligible, the worst case complexity can be bounded by $O(B^2)$.

4.2.2 Correctness

Let us now prove that the liveness of the resulting reducible graph is equivalent to the liveness of the original graph by showing that for every basic block present in both graphs the live-in sets are, in fact, the same:

Theorem 1. *Given a variable* v*, if the definition and the uses are placed in the same nodes in* $\Psi_L(G)$*, then for each node* X *from* V*,* v *is live-in in* G *iff* v *is live-in in* $\Psi_L(G)$*.*

Proof. We will show the equivalence property. Given nodes d, u and q from G, where d dominates u. There exists a path in G from q to u that does not contain d iff there exists such a path in G'.

Given p, a path in G, from q to u that does not contain d.

- If $d \in L$, then p does not contain any *LoopBackEdges* or *EntryEdges* (because all nodes in $Header \cup Entries$ are not dominated by any node from the loop L). So, p is a valid path in G'.
- Otherwise, if $d \notin L$. We replace every edge $x \to y \in (LoopBackEdges(L) \cup EntryEdges(L))$ of the path p by $x \rightsquigarrow \delta_L \to h \rightsquigarrow y$, where the path from h to y is fully contained in L.

Given a path p in G' from q to u that does not contain d.

- If $d \in L$, then p does not contain δ_L (because d cannot dominate δ_L), hence p is a valid path in G.
- Assuming $d \notin L$, if p does not contain δ_L, then p is a valid path in G. Otherwise we can pick $x \to \delta_L \to h$, from p, such that $h \in Headers(L)$. There exists h' in L such that $p \to h'$ is an edge in G. And there exists a path p' within the loop L from h' to h. So we can replace δ_L by p'. □

5 Liveness Sets using Path Exploration

Another maybe more intuitive way of calculating liveness sets is closely related to the definition of the live-range of a given variable. A variable is live at a program point p, if p belongs to a path of the CFG leading from a definition of that variable to one of its uses without passing through the definition. Therefore, the live-range of a variable can be computed using a backward traversal starting at its uses and stopping when reaching its (unique) definition. This ideas was first proposed by Appel in his "Tiger" book [3]. We distinguish two implementation variants of the basic idea.

5.1 Processing Variables by Use

The first variant relies solely on the CFG of the input program and does not require any additional preprocessing steps. Starting from a use of a variable all paths, where that particular variable is live, are enumerated by traversing the CFG backwards until the variable's definition is reached. Along the encountered paths the variable is appended to the live-in and live-out sets of the respective basic blocks.

Algorithm 4 Computing liveness sets by exploring paths from variable uses.

```
1: function Compute_LiveSets_SSA_ByUse(CFG)
2:     for each basic block B in CFG do      ▷ Consider all blocks successively
3:         for each v ∈ PhiUses(B) do      ▷ Used in the φ of a successor block
4:             LiveOut(B) = LiveOut(B) ∪ {v}
5:             Up_and_Mark(B, v)
6:         for each v used in B (φ excluded) do
7:             Up_and_Mark(B, v)          ▷ Traverse the block to find all uses
```

Algorithm 4 performs the initial traversal discovering the uses of all variables in the program. Every use is the starting point for a path exploration performed by Algorithm 5. The presented algorithm has some similarities to the liveness algorithm used by the open-source compiler infrastructure LLVM.

5.2 Processing Variables by Definition

The second variant follows the initial idea by Appel. Depending on the particular compiler framework a preprocessing step might be required in order to derive the def-use chains for all variables, i.e., a list of all uses for each SSA-variable.

In contrast to the Use-by-Use variant, it is no necessary to explicitly scan the program for uses of each variable when def-use chains are available. Algorithm 6 adopts the pseudo-code shown previously to make use of the def-use chains. The algorithm to perform the path exploration stays the same, i.e., Algorithm 5.

A nice property of this approach is that the processing of different variables is not intermixed, i.e., the processing of one variable is completed before the processing of another variable begins. This enables to optimize the

Algorithm 5 Exploring all paths from a variable's use to its definition.

```
1: function Up_and_Mark(B, v)
2:     if def(v) ∈ B (φ excluded) then return      ▷ Killed in the block, stop
3:     if v ∈ LiveIn(B) then return      ▷ Propagation already done, stop
4:     LiveIn(B) = LiveIn(B) ∪ {v}
5:     if v ∈ PhiDefs(B) then return      ▷ Do not propagate φ definitions
6:     for each P ∈ CFG_preds(B) do      ▷ Propagate backward
7:         LiveOut(P) = LiveOut(P) ∪ {v}
8:         Up_and_Mark(P, v)
```

Algorithm 6 Computing liveness sets per variable using def-use chains.

```
1: function Compute_LiveSets_SSA_ByVar(CFG)
2:     for each variable v do
3:         for each use of v, in a block B do
4:             if v used at exit of B then      ▷ Used in the φ of a successor block
5:                 LiveOut(B) = LiveOut(B) ∪ {v}
6:             Up_and_Mark(B, v)
```

Algorithm 7 Optimized path exploration using a stack-like data structure.

```
1: function Up_and_Mark_Stack(B, v)
2:     if def(v) ∈ B (φ excluded) then return        ▷ Killed in the block, stop
3:     if top(LiveIn(B)) = v then return      ▷ propagation already done, stop
4:     push(LiveIn(B), v)
5:     if v ∈ PhiDefs(B) then return          ▷ Do not propagate φ definitions
6:     for each P ∈ CFG_preds(B) do                          ▷ Propagate backward
7:         if top(LiveOut(P)) ≠ v then push(LiveOut(P), v)
8:         Up_and_Mark_Stack(P, v)
```

mark phase of the algorithm by using a stack-like set representation. The expensive set-insertion operations and set-membership tests can be avoided, as shown by Algorithm 7.

5.3 Path Exploration for non-SSA-form Programs

Interestingly, the path exploration approach can also be applied to programs that are *not* in SSA form. Similar to the precomputation of the def-use chains for the Variable-by-Variable approach, we have to precompute information on uses and definitions of all variables in the program. First, for each variable v, the list of blocks where v is live-in and upward-exposed is computed, i.e., those blocks where the first access to v is a use and not a definition. We also compute the list of blocks where the variable is defined, as shown by Algorithm 8. These sets are denoted by Definitions(v) and UpwardExposed(v) in the following algorithms.

The algorithm to compute the liveness information is similar to the optimized Variable-by-Variable algorithm presented in the previous section. The main difference is that multiple definitions of the same variable might appear in the program. In order to avoid expensive checks to find definitions during the path exploration, we use a marking scheme. Basic blocks can be marked with a variable during the processing. The marking indicates that the path exploration algorithm should stop following the current path any further. A block is marked with the current variable in two cases: (1) The basic block contains a definition of the variable or (2) the basic block has been visited by the path exploration algorithm previously, the variable is thus already known to be live. Algorithm 9 and 10 show the modified mark phase and the pseudo-code of the liveness algorithm for programs that are not in SSA form.

Algorithm 8 Computing the upward-exposed uses and definitions of variables.

```
1: function Compute_Killing_and_UpwardExposed_Stack(CFG)
2:     for each basic block B in the CFG do
3:         for each access to a variable v, from start to end of block do
4:             if top(Definitions(v)) ≠ B then                ▷ No definition yet
5:                 if v is a use then                         ▷ Upward-exposed use
6:                     if top(UpwardExposed(v)) ≠ B then
7:                         push(UpwardExposed(v), B)
8:                 else push(Definitions(v), B)               ▷ First definition
```

Algorithm 9 Computing liveness sets per variable for non-SSA-form programs.

```
1: function Up_and_Mark_NonSSA_Stack(B, v)
2:     if top(LiveOut(P)) ≠ v then push(LiveOut(P), v)
3:     if B is marked with v then return          ▷ Killed in the block, stop
4:     mark B with v
5:     push(LiveIn(B), v)
6:     for each P ∈ CFG_preds(B) do               ▷ Propagate backward
7:         Up_and_NonSSA_Mark_Stack(P, v)
```

5.4 Complexity

All path-based approaches yield essentially the same complexity results. The outermost loops of Algorithm 4 and 6 in the worst case visit every instruction once per variable in order to start a path traversal, which results in an $O(I \cdot V)$ bound. The depth-first traversal of the CFG similarly visits every edge of the graph once per variable, the number of set insertions, respectively stack operations, performed by the loop of Algorithm 5 and 7 is thus limited by $O(E \cdot V)$. The insertions outside of the loop are performed only once per basic block per variable, and thus do not appear in the final bound. The overall complexity is given by $O(I \cdot V + E \cdot V)$, assuming unit time set insertions.

The algorithm for programs not in SSA form shows a similar structure and thus also behaves similarly. However, we need to account for the precomputation of the upward-exposed uses and variable definitions for every block in the program – see Algorithm 8. The algorithm visits every instruction once per variable, which does not change the bound state above. The algorithm also incurs some initialization overhead due to the marking of basic blocks. The first for-loop is executed once for every basic block, while the second loop at line 5 of Algorithm 10 gives $O(B \cdot V)$. Assuming a connected CFG, this leaves the bound unchanged. All path-based algorithms thus share the same complexity bound.

This bound differs slightly from the result $O(I \cdot V)$ obtained by McAllester [21]. Firstly, McAllester does not consider basic blocks in his analysis, i.e., every instruction is considered a separate block. Given a connected CFG, this implies $I \leq E$. Secondly, he assumes an out-degree of at most two for the CFG nodes, which further adds $E \leq 2 \cdot I$. Thirdly, he assumes a bounded number of uses and definitions per instruction. Naively extending McAllester would violate assump-

Algorithm 10 Computing liveness per variable for non-SSA-form programs.

```
1: function Compute_LiveSets_NonSSA_ByVar_Stack(CFG)
2:     for each basic block B of CFG do
3:         mark B with ⊥
4:     for each variable v do
5:         for each block B in Definitions(v) do mark B with v
6:         for each block B in UpwardExposed(v) do
7:             push(LiveIn(B), v)              ▷ No duplication, no check needed
8:             for P ∈ CFG_preds(B) do         ▷ Propagate backward
9:                 Up_and_Mark_NonSSA_Stack(P, v)
```

tions in his complexity analysis and would lead to inferior bounds compared the approaches presented here. Accounting for those simplifications, his findings are, nevertheless, in line with the results obtained above – even though they are not applicable for general program representations appearing in actual compilers.

6 Experiments

We applied the various algorithms proposed in this work to a set of programs from the SPECINT 2000 benchmark suite in order to measure the time required to compute full liveness sets, i.e., for all basic blocks in the program the live-in and live-out sets for all global variables. The number of global variables, i.e., variables crossing basic block boundaries, depends largely on the compiler optimizations performed before the liveness calculation. Programs that are not optimized usually yield very few global variables since most values are kept in memory locations by default. However, optimized programs usually yield longer and more branched live-ranges. We thus investigate the behavior for optimized and unoptimized programs using the compiler flags `-O2` and `-O0` respectively. Table 1 shows the number of global variables, basic blocks, and operations for the optimized benchmark programs. The statistics for unoptimized programs are not shown, since the number of global variables never exceeds 19.

In order to obtain reproducible results, the execution time is measured using the instrumentation and profiling tool *callgrind*, which is part of the well-known *valgrind* tool. The measurements include the number of dynamic instructions executed as well as memory accesses via the instruction- and data caches. Using those measurements a cycle estimate is computed for the liveness computation only. The interaction with other compiler components and other programs running in parallel on the host machine is thus minimized.

The algorithms were implemented using the production compiler for the STMicroelectronics ST200 VLIW family, which is based on GCC as front-end, the Open64 optimizers, and the LAO code generator [12]. The liveness is computed relatively late during the final code generation phase of the LAO low-level

	# Variables			# Blocks			# Operations		
Benchmark	min	avg	max	min	avg	max	min	avg	max
164.gzip	11	104	586	2	32	212	22	226	1312
175.vpr	10	84	573	2	33	492	21	224	1734
176.gcc	10	119	36063	2	37	1333	11	282	41924
181.mcf	12	52	118	2	18	52	24	135	439
186.crafty	11	147	1048	2	67	2112	22	547	9836
197.parser	10	58	1076	2	21	343	21	126	1942
253.perlbmk	10	61	1947	2	28	731	16	180	4876
254.gap	10	95	6472	2	31	778	13	244	9169
255.vortex	10	51	645	2	26	667	21	166	3361
256.bzip2	10	73	972	2	22	282	21	163	1931
300.twolf	10	186	3659	2	53	715	12	458	8691

Table 1: Program characteristics for optimized programs.

optimizer, shortly before prepass scheduling is performed. In addition, all algorithms were implemented and optimized for two different liveness set representations. We evaluated the impact of pointer-sets, which promise reduced memory consumption at the expense of rather time-consuming set operations. In addition, plain bit-sets were evaluated, which are often considered to be inefficient in terms of memory consumption and are expected to degrade in performance as the number of variables increases.

6.1 Pointer-Sets

The pointer-sets in LAO are implemented as arrays ordered by numeric identifiers. This results in rather fast set operations such as union and intersection at the expense of a rather expensive insertion. Due to the ordering of the pointer-sets, insertions are the fastest when the inserted variable is known to have an index number larger than all other variables in the set. The implementation of the Variable-by-Variable algorithm thus performs best with the optimization for stack-like data structures presented in Section 5.2. Note that the ordering of the set is preserved throughout the computation.

Figure 4 and 5 compare the average execution times measured for the individual benchmarks of the Variable-by-Variable and the Use-by-Use approach in comparison to the non-iterative data-flow algorithm using loop-forests. Not surprisingly, the results indicate that the Variable-by-Variable algorithm outperforms the loop-forest-based approach by 74% and 64% for optimized and unoptimized programs respectively. The results for the Use-by-Use algorithm highly depend on the characteristics of the input program. In particular for larger optimized programs such as `gcc`, `perlbmk`, and `twolf` the Use-by-Use approach shows poor results. This can be attributed to the unordered processing of the variables, resulting in costly insertion operations. The non-

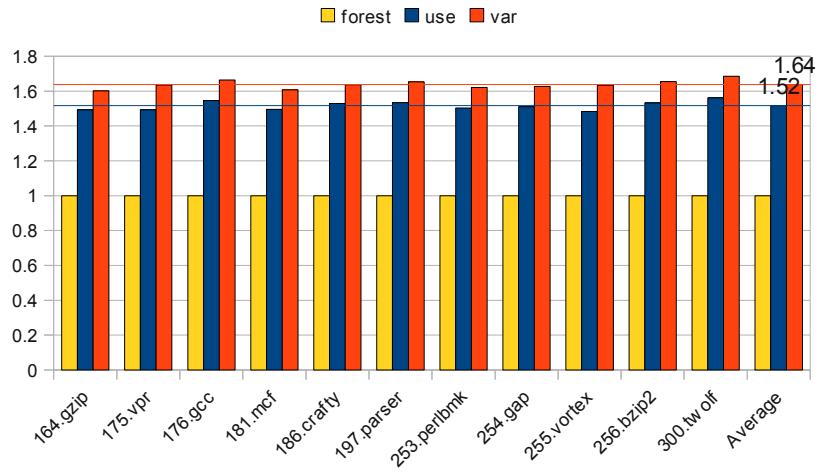


Figure 4: Speed-up of the path-based algorithms relative to our new non-iterative data-flow analysis using pointer-sets on non-optimized programs.

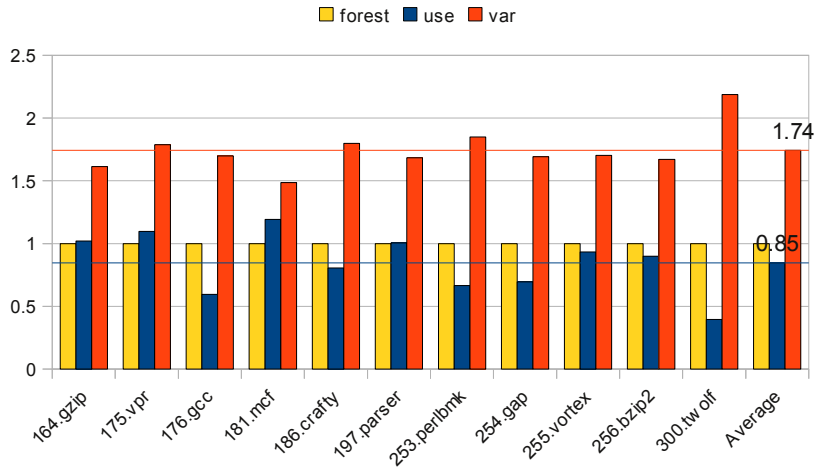


Figure 5: Speed-up of the path-based algorithms relative to our new non-iterative data-flow analysis using pointer-sets on optimized programs.

iterative data-flow analysis mainly applies set unification, which can be performed fast on the ordered set representation, and thus gains in comparison to the Use-by-Use variant.

However, considering the program characteristics from Table 1, sparse pointer-sets do not appear to be a good choice to represent liveness sets. The average number of variables per function is relatively low and does not exceed 184 for our benchmark set. In fact, 97% out of the 5848 functions contain less then 320 variables and almost 99% less then 640, which yields a size of merely 20 words on 32-bit machines in order to represent all variables as bit-sets for almost all functions considered. It is thus not surprising that the baseline iterative data-flow algorithm using bit-sets outperforms the same algorithm using pointer-sets by 69% and 85% for optimized and unoptimized input programs.

6.2 Bit-Sets

The use of bit-sets in data-flow analysis is very common since most of the required operations, such as insertion, union, and intersection, can be performed efficiently using this representation. In fact, our measurements show that these operations are so fast that the allocation and initialization of the bit-sets becomes a major factor in the overall execution time of the considered algorithms.

For unoptimized programs the results follow the observations for pointer-sets – see Figure 6. Since the number of variables is low and the extend of the respective live-ranges is short, the set representation is of less importance and a smart processing of the variables performs best. The program size, i.e., the number of basic blocks and operations, has less impact on the Variable-by-Variable, which simply iterates over the small set of variables. The other approaches, however, have to traverse the CFG and its operations in order to find upward-exposed uses. The Variable-by-Variable approach thus shows considerably shorter execution times than the loop-forest-based algorithm, resulting

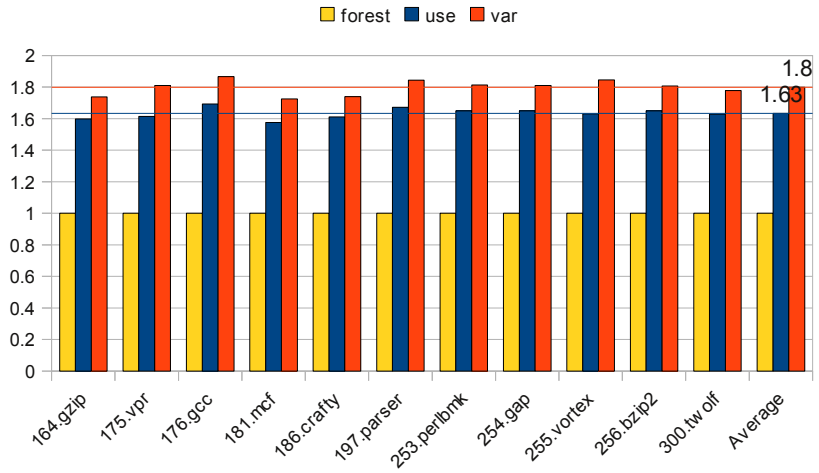


Figure 6: Speed-up relative to the non-iterative data-flow algorithm using bit-sets on unoptimized programs.

in an average speed-up of 80%. The Use-by-Use approach still has to inspect the program's operations and thus cannot reach the performance of the Variable-by-Variable variant. Nevertheless, in comparison to the loop-forest-based algorithm an average speed-up of 63% is observed over all benchmark programs.

The characteristics of optimized programs are, however, different. The previously fastest algorithm is now performing the least, as depicted by Figure 7. The forest-based approach clearly outperforms both path-exploration algorithms. This is explained by the relative cost of the fast bit-set operations in comparison to the cost of traversing the CFG. On average the non-iterative data-flow

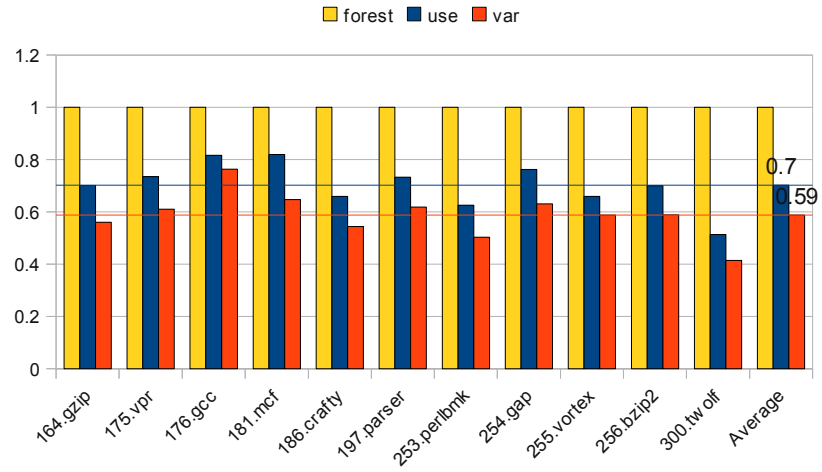


Figure 7: Speed-up relative to the non-iterative data-flow algorithm using bit-sets on optimized programs.

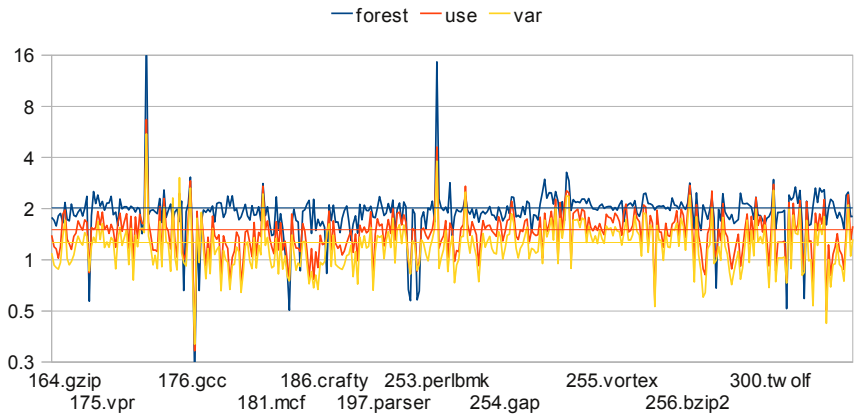


Figure 8: Speed-up of the Variable-by-Variable, the Use-by-Use, and the loop-forest-based algorithm relative to iterative data-flow analysis on optimized programs using bit-sets.

algorithm thus achieves speed-ups of 69% and 43% respectively. Furthermore, the locality of memory accesses becomes a relevant performance factor. Both, the Use-by-Use as well as the loop-forest-based algorithms, operate *locally* on the bit-sets surrounding a given program point. The inferior locality, combined with the necessary precomputation of the def-use chains, explains the poor results of the Variable-by-Variable approach in this experimental setting.

So far, we have only compared the path-exploration algorithms with our novel non-iterative data-flow approach. Figure 8 relates the standard iterative data-flow approach to the algorithms presented in this work on a per module basis, i.e., using one data point for every source file. The loop-forest and the Use-by-Use algorithm on average clearly outperform the iterative computation by a factor of two and by 49% respectively. The extreme cases showing speed-ups by a factor higher than 8 are caused by unusual loop structures in code generated by the parser generator *bison* (`c-parse.c` of `gcc`, and `perly.c` of `perlbmk`). On the other hand, all cases where the iterative approach outperforms the non-iterative are explained by implementation artifacts, i.e., the analyzed functions do not contain any global variables thus slight variations in the executed code, the code placement, and the state of the data-caches become relevant. The Variable-by-Variable approach is often even slower than the iterative computation and on average shows a moderate speed-up of 25%.

6.3 Non-SSA-form Programs

In addition to the algorithms that require SSA form to be available, we also considered the path-based approach for programs not under SSA. The implementation is based on pointer-sets, which showed the best speed-up for this algorithm variant in our previous experiments.

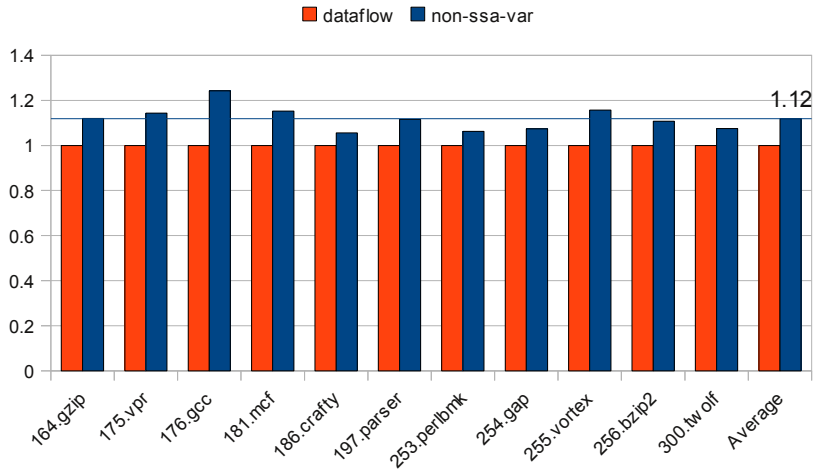


Figure 9: Speed-up of the Variable-by-Variable approach relative to iterative data-flow analysis using pointer-sets on unoptimized non-SSA programs.

The algorithm requires a precomputation step in order to determine the sets of defined variables and upward-exposed uses. The relative speed-ups are thus diminished in comparison to the SSA-based algorithms, which have this information readily available. For unoptimized programs the algorithm provides on average a gain of 12% in comparison to the standard iterative data-flow algorithm as depicted by Figure 9. The trend observed in our previous experiments is confirmed also in this setting. The results for optimized programs are much better; Figure 10 shows an average speed-up of 22% over all benchmarks. Also the results per-module, presented in Figure 11, follow the previous

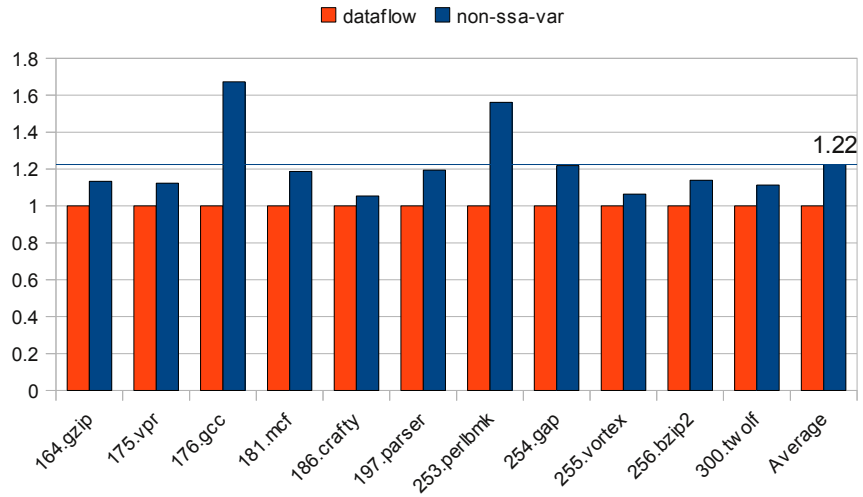


Figure 10: Speed-up of the Variable-by-Variable approach relative to iterative data-flow analysis using pointer-sets on optimized non-SSA programs.

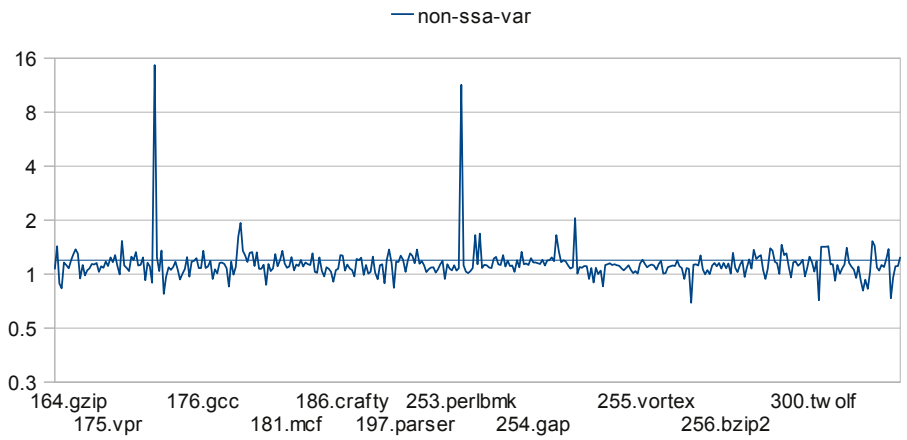


Figure 11: Speed-up of the Variable-by-Variable relative to iterative data-flow analysis on optimized non-SSA programs using pointer-sets.

findings, albeit with reduced gains. An interesting detail is that the magnitude and the number of spikes indicating a slowdown in comparison to the data-flow algorithm is much smaller. Inspecting the involved benchmarks revealed that functions where the number of variables is exceedingly increased by SSA form and where the number of ϕ-operations is high are particularly affected.

7 Conclusion

Liveness information forms the basis for many compiler optimizations and transformations. However, many of those transformations invalidate the liveness information by introducing new variables, new instructions, or by modifying the control-flow graph. Consequently, liveness analysis is performed several times throughout the compilation of an input program. Fast algorithms are thus required in order to minimize the penalty incurred by the steady recomputation.

This work presents an alternative to the traditional iterative data-flow analysis in order to compute liveness information for programs in strict SSA form. The algorithm consists of two major phases. The first resembles the precomputation phase of the original data-flow approach and provides partial liveness sets. The second phase replaces the iterative refinement of those partial liveness sets by a single traversal of the loop-nesting forest. Furthermore, we present two algorithms that rely on path exploration to compute the program points where individual variables are known to be live and need to be appended to the respective liveness sets. These algorithms similarly exploit properties provided by SSA-form programs in order to improve execution time. However, both variants can also be applied to programs not in SSA form.

As our experiments show, all of those algorithms outperform the iterative method by up to a factor of two on average for the SPECINT 2000 benchmark suite. Depending on the program characteristics and the underlying representation of the liveness sets either the non-iterative data-flow algorithm or the

algorithms using path exploration provide favorable execution times. For heavily optimized code having a high number of global variables and complex control flow the non-iterative approach based on loop-forests is suited best. Whereas, less optimized or even unoptimized code, having very few global variables, is best handled using one of the path exploration algorithms.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[2] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications ACM*, 19(3):137, 1976.

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

[4] B. Boissinot, A. Darte, B. Dupont de Dinechin, C. Guillon, and F. Rastello. Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In *International Symposium on Code Generation and Optimization (CGO'09)*, pages 114–125. IEEE Computer Society Press, March 2009.

[5] F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register allocation: What does the NP-completeness proof of chaitin et al. really prove? In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'06)*, LNCS. Springer Verlag, November 2006.

[6] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation and spill complexity under SSA. Technical Report RR2005-33, LIP, ENS-Lyon, France, August 2005.

[7] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience*, 28(8):859–881, July 1998.

[8] P. Brisk, F. Dabiri, J. Macbeth, and M. Sarrafzadeh. Polynomial time graph coloring register allocation. In *International Workshop on Logic and Synthesis (IWLS'05)*. ACM Press, 2005.

[9] Larry Carter, Jeanne Ferrante, and Clark Thomborson. Folklore confirmed: reducible flow graphs are exponentially larger. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL '03, pages 106–114. ACM, 2003.

[10] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. An empirical study of iterative data-flow analysis. In *CIC '06: Proceedings of the 15th International Conference on Computing*, pages 266–276, Washington, DC, USA, 2006. IEEE Computer Society.

[11] 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.

[12] B. Dupont de Dinechin, F. de Ferriere, C. Guillon, and A. Stoutchinin. Code generator optimizations for the ST120 DSP-MCU core. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00)*, pages 93–102. ACM Press, 2000.

[13] S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA form. In *International Conference on Compiler Construction (CC'06)*, volume 3923 of *LNCS*, pages 247–262. Springer, March 2006.

[14] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, 1997.

[15] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, July 1974.

[16] Matthew S. Hecht and Jeffrey D. Ullman. Analysis of a simple algorithm for global data flow problems. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 207–217, New York, NY, USA, 1973. ACM.

[17] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19:1031–1052, November 1997.

[18] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.

[19] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206, New York, NY, USA, 1973. ACM.

[20] A. Leung and L. George. Static single assignment form for machine code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 204–214. ACM Press, 1999.

[21] David McAllester. On the complexity analysis of static analyses. *Journal of the ACM (JACM)*, 49:512–537, July 2002.

[22] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[23] F. M. Q. Pereira and J. Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *LNCS*, pages 315–329. Springer, November 2005.

[24] G. Ramalingam. Identifying loops in almost linear time. *ACM Transactions on Programming Languages and Systems*, 21(2):175–188, March 1999.

[25] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems*, 24(5):455–490, 2002.

[26] F. Rastello, F. de Ferrière, and C. Guillon. Optimizing translation out of ssa using renaming constraints. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 265–276. IEEE Computer Society Press, March 2004.

[27] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[28] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *Proceedings of the 6th International Symposium on Static Analysis (SAS'99)*, pages 194–210. Springer-Verlag, 1999.

[29] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9(3):355–365, December 1974.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399