A Non-iterative Data-Flow Algorithm for Computing Liveness Sets in Strict SSA Programs

Benoit Boissinot[1], Florian Brandner[1], Alain Darte[1], Benoît Dupont de Dinechin[2], and Fabrice Rastello[1]

[1] LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon
[2] Kalray

Abstract. We revisit the problem of computing liveness sets (the sets of variables live-in and live-out of basic blocks) for programs in strict static single assignment (SSA). In strict SSA, aka SSA with dominance property, the definition of a variable always dominates all its uses. We exploit this property and the concept of loop-nesting forest to design a fast two-phases data-flow algorithm: a first pass traverses the control-flow graph (CFG), propagating liveness information backwards, a second pass traverses a loop-nesting forest, updating liveness sets within loops. The algorithm is proved correct even for irreducible CFGs. We analyze its algorithmic complexity and evaluate its efficiency on SPECINT 2000. Compared to traditional iterative data-flow approaches, which perform updates until a fixed point is reached, our algorithm is 2 times faster on average. Other approaches are possible that propagate from uses to definitions, one variable at a time, instead of unioning sets as in data-flow analysis. Our algorithm is 1.43 times faster than the fastest alternative on average, when sets are represented as bitsets and for optimized programs, which have non-trivial live-ranges and a larger number of variables.

1 Introduction

Static single assignment (SSA) is a popular program representation used by most modern compilers. Initially developed to facilitate the development of high-level program transformations, SSA has gained much interest due to its properties that often lead to simpler algorithms and reduced computational complexity. Today, SSA is even adopted for the final code generation phase [26]. For instance, several industrial and academic compilers, static or just-in-time, use SSA in their back-ends, e.g., LLVM [28], Java HotSpot [25], LAO [17], LibFirm [27,14], Mono [31]. Recent research on register allocation [10,18,33] showed that SSA form can even be retained until the very end of the code generation process.

This work explores the use of SSA properties to simplify and accelerate *liveness analysis*, which determines, for each basic block, the variables whose values are eventually used by subsequent operations. This information is essential to solve storage assignment problems, eliminate redundancies, and perform

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 137–154, 2011.
© Springer-Verlag Berlin Heidelberg 2011

code motion. Optimizations like software pipelining, trace scheduling, register-sensitive redundancy elimination, if-conversion, and register allocation heavily rely on liveness information. Traditionally, liveness is obtained by data-flow analysis: liveness sets are computed for all basic blocks, all variables treated together, by solving a set of data-flow equations [3]. These equations are usually solved by an iterative algorithm, propagating information backwards through the control-flow graph (CFG) until a fixed point is reached. The number of iterations depends on the CFG structure and on the order in which basic blocks are evaluated.

We show that, for SSA-form programs, it is possible to design a fairly simple, two-passes, data-flow algorithm to compute liveness sets *that does not require to iterate* to reach a fixed point. Its first pass, very similar to the initialization phase of traditional data-flow analysis, computes partial liveness sets by traversing the CFG backwards. Its second pass refines the partial liveness sets within loops by traversing a loop-nesting forest, as defined by Ramalingam [35]. Such a loop hierarchy is already available in modern compilers for other optimizations that exploit the structure of loops. For the sake of clarity, we first present our algorithm for reducible CFGs, then we show that irreducible CFGs can be handled with a slight variation, with no need to modify the CFG itself.

Other approaches are possible, for example as proposed by Appel [3] or McAllester [29], that propagate liveness from uses to definitions, one variable at a time, instead of unioning sets as in standard data-flow analysis. For a broader comparison with state of the art, we designed optimized implementations of this path-exploration principle (improved to work at the granularity of basic blocks instead of instructions as in the original versions) and compared the efficiency of the resulting algorithms with our non-iterative data-flow algorithm.

Our experiments using the SPECINT 2000 benchmark suite demonstrate that the non-iterative data-flow algorithm outperforms the standard iterative data-flow algorithm by a factor of 2 on average. By construction, our algorithm is best suited for a set representation, such as bitsets, favoring operations on whole sets. In particular, for optimized programs, which have non-trivial live-ranges and a larger number of variables, our algorithm achieves a speed-up of 1.43 on average in comparison to the fastest alternative based on path exploration.

2 Related Work

Liveness information is usually computed with iterative data-flow analysis, which goes back to Kildall [24]. The algorithms are, however, not specialized to the computation of liveness sets and may incur overhead. Several strategies are possible, leading to different worst-case complexities and performance in practice. *Round-robin algorithms* propagate information according to a fixed block ordering derived from a depth-first spanning tree and iterate until it stabilizes. The complexity of this scheme was analyzed by Kam et al. [23], see Section 3.4. *Work-list algorithms* focus on blocks that may need to be updated because the liveness sets of their successors (for backward problems) changed. Empirical results by Cooper et al. [15] indicate that the order in which basic blocks are processed

is critical and directly impacts the number of iterations. They showed that, in practice, a mixed solution, called "single stack worklist", based on a worklist initialized with a round-robin order, is the most efficient one for liveness analysis. In contrast, our non-iterative data-flow algorithm requires at most two passes over the basic blocks, in all cases. In practice, for strict SSA programs, it is on average twice as fast as the "single stack worklist" approach (see Section 5).

An alternative way to solve data-flow problems is interval analysis [2] and other elimination-based approaches [36]. 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. Also, except for the Graham-Wegman algorithm, interval-based algorithms require the CFG (resp. the reverse CFG) to be reducible for a forward (resp. backward) analysis [36]. In practice, irreducible CFGs are rare, but liveness analysis is a backward data-flow problem, which frequently leads to irreducible reverse CFGs. In contrast, our algorithm does not require the *reverse* CFG to be reducible. If the CFG is irreducible, care must be taken when propagating liveness information backward in the CFG, but with no modification of the CFG itself (see Section 4.2).

Another approach to compute liveness was proposed by Appel [3, p. 429]. Instead of computing the liveness information for all variables at the same time, variables are handled individually by exploring paths in the CFG starting from variable uses. An equivalent approach using logic programming was presented by McAllester [29], showing that liveness analysis can be performed in time proportional to the number of instructions and variables. However, his theoretical analysis is limited to a restricted input language with simple conditional branches and instructions. A more generalized analysis will be given later, both in terms of theoretical complexity (Section 3.4) and of practical evaluation (Section 5).

Liveness analysis for strict SSA programs was first addressed by Boissinot et al. [9], but with a different perspective. They showed that queries such as "is variable v live at program point p?" can be performed quickly, thanks to a pre-processing step depending on the CFG structure only. Wimmer et al. [39] gave an algorithm, specialized to linear scan register allocation, to build the "intervals" of basic blocks where each variable is live. Although a possible extension to irreducible CFGs is sketched, the algorithm restricts itself to reducible CFGs or to a form of SSA where live-ranges are cut at loop-entry blocks with ϕ-functions. The algorithm we propose is a generalization[1] for computing liveness sets: it uses the concept of loop-nesting forest and is proved correct with no restriction on the CFG, on the strict SSA form, or on the loop-nesting forest as long as it respects the minimal properties stated by Ramalingam [35]. As a by-product, this proves the correctness of the algorithm of [39] and how a suitable order of basic blocks can be chosen thanks to a loop-nesting forest. Such orders were

[1] Actually, we designed this algorithm in 2009-2010 independently of [39].

also exploited for liveness analysis in static single information (SSI) [8]. The live-check algorithm of [9] was also reformulated using loop-nesting forests [6].

3 Foundations

This section recalls the concepts of control-flow graphs, loop-nesting forests, dominance, and SSA form. Readers familiar with them can skip this section.

3.1 Control-Flow Graph and Loop Structure

A *control-flow graph* $G = (V, E, r)$ is a directed graph, with nodes V, edges E, and a distinguished *root* $r \in V$ from which there is a path to any other node. 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.

Dominance. A node x in a CFG *dominates* a node y if every path from the root r to y contains x. The dominance is strict if $x \neq y$. The transitive reduction of the dominance relation forms a tree, the *dominator tree*.

Loop-Nesting Forest. To discuss previously-proposed constructions, Ramalingam [35] gave a minimal definition of loop-nesting forest by a recursive process:

1. Partition the CFG into its strongly connected components (SCCs). Every non-trivial SCC, i.e., with at least one edge, is called a *loop*.
2. For each loop L, select a non-empty set of nodes in L among those that are not dominated by any other node in L: its elements are called the *loop-headers* of L. (Different choices may lead to different forests.) Remove all edges in L that lead to a loop-header of L and call them the *loop-edges* of L.
3. Repeat this partitioning recursively for every SCC, after its loop-edges have been removed. The process stops when only trivial SCCs remain.

This decomposition can be represented by a forest whose leaves are the nodes of the CFG, while internal nodes, labeled by loop-headers, correspond to loops. The children of a loop's node represent all inner loops (i.e., all non-trivial SCCs) it contains 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. Note also that a loop-header cannot belong to any inner loop because all edges leading to it are removed before computing inner loops. In the rest of this paper, we make no other assumption on the way the loop-nesting forest is built: any of the algorithms analyzed in [34,35] can be applied.

Reducible Control-Flow Graphs. A CFG is *reducible* if every loop has a single node that dominates all other nodes of the loop [20]. 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 CFGs is of special interest for compiler writers. Indeed, most programs exhibit reducible CFGs. Also, as pointed out

earlier, unlike other approaches that compute liveness information, we only need to discuss the reducibility of the original CFG, not of the reverse CFG.

Computing a Loop-Nesting Forest. A loop-nesting forest can be computed in "almost linear time" $O(|E|\alpha(|E|, |V|))$ [34]. Unlike reducible CFGs, the loop-nesting forest of an irreducible CFG is not unique as some loops have several nodes not dominated by any other node in the loop. A simple-to-engineer construction algorithm is the generalization of Tarjan's algorithm [38] proposed by Havlak [19], later improved by Ramalingam [34] to fix a complexity issue. It identifies a loop as a set of descendants of a back-edge target that can reach its source. In that case, the set of loop-headers is restricted to a single entry node, the target of a back-edge. Also, while identifying loops, 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 (SSA) [16] is a popular program representation. In SSA, each scalar variable is defined only once textually. To build SSA, variables having multiple definitions are replaced by several new *SSA variables*, one for each definition. When a use in the original program was reachable from multiple definitions, the new variables are disambiguated by introducing ϕ-functions at control-flow joins. Each ϕ-function defines a new SSA variable by selecting the right SSA variable whose definition was traversed in the actual execution flow.

We require the program to be in *strict* SSA, i.e., every path from the root r to a use of a variable contains a definition of this variable. Because there is only one (static) definition per variable, strictness is equivalent to the *dominance property*, which states that each use of a variable is dominated by its definition. This is true for all uses including a use in a ϕ-operation by considering that such a use actually takes place in the predecessor block from where it originates.

3.3 Liveness

Intuitively, a variable is *live* at a program point when its value is used later by any dynamic execution. Statically, liveness can be approximated by following paths of the CFG, backwards, from the uses of a given variable to its definitions (unique definition in SSA). The variable is live at all program points along these paths. For a CFG node q, representing an instruction or a basic block, a variable `v` is *live-in* at q if there is a path, not containing the definition of `v`, from q to a node where `v` is used. It is *live-out* at q if it is live-in at some successor of q.

The computation of live-in and live-out sets at the entry and the exit of basic blocks is usually termed *liveness analysis*, i.e., algorithms operate at the granularity of blocks. It is indeed sufficient to consider only these sets since liveness within a block can be recomputed from its live-out set, either by traversing the block or by precomputing the variables *defined* and the variables *upward-exposed* in the block. A variable is upward-exposed in a block B when it is used in B and not defined earlier in B – in strict SSA, this simply means not defined in B.

However, the special behavior of ϕ-operations often causes confusion on where their operands are actually used and defined. Their liveness should be considered with care, especially when dealing with SSA destruction [11,37,5]. To make the description of algorithms easier, we follow the definition by Sreedhar [37]. For a ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$ in block B_0, where a_i comes from block B_i, then:

- a_0 is live-in for B_0, but, w.r.t this ϕ-function, not live-out for B_i, $i > 0$.
- a_i, $i > 0$, is live-out of B_i, but, w.r.t this ϕ-function, not live-in for B_0.

This semantics, which corresponds to placing a copy of a_i to a_0 on each edge from B_i to B_0, can be expressed by the following *data-flow equations*:

$$\begin{aligned} \text{LiveIn}(B) &= \text{PhiDefs}(B) \cup \text{UpwardExposed}(B) \cup (\text{LiveOut}(B) \setminus \text{Defs}(B)) \\ \text{LiveOut}(B) &= \bigcup_{S \in \text{succs}(B)} (\text{LiveIn}(S) \setminus \text{PhiDefs}(S)) \cup \text{PhiUses}(B) \end{aligned}$$

where $\text{PhiDefs}(B)$, resp. $\text{PhiUses}(B)$, denotes the variables defined, resp. used, by ϕ-operations at entry of B, resp. a successor block of B. The algorithms presented hereafter follow this semantics. They require minor modifications when other ϕ-semantics are desired. We will come back to these subtleties in Section 4.1.

3.4 Complexity of Liveness Algorithms

The running times of liveness algorithms depend on several parameters. Some can only be evaluated by experiments, e.g., the locality in data structures, the cost of function calls instead of inlined operations, etc. This is discussed in Section 5.

Complexity Parameters. Usually, liveness algorithms consider only the set W of *non-local variables*, i.e., the variables whose live-ranges cross some basic block boundary. The complexity of set operations is then measured in terms of $|W|$, the cardinality of W. However, to identify non-local variables, to identify uses and definitions, all instructions of the program P need to be visited. Traversing its internal representation is costly and not directly linked to $|W|$ as it involves all variables. Thus, any liveness algorithm requires at least $|P|$ operations to read the program and, in practice, it is better to read it only once.

After possibly some precomputations in $O(|P|)$ operations, liveness algorithms work on the CFG $G = (V, E, r)$. The number of operations can then be evaluated in terms of $|V|$ and $|E|$, i.e., the number of times blocks and control-flow edges are visited. Hereafter, we assume $|V| - 1 \leq |E| \leq |V|^2$. The costs of these operations depend on the data structures used – e.g., lists, bitsets, or sparse sets [12] – both for intermediate results (e.g., uses of a variable or upward-exposed uses in a block) and for the final results, i.e., the live-in and live-out sets. Here, we will mainly discuss the case of bitsets, as explained in Section 5.

Standard Data-Flow Approaches. The data-flow equations of Section 3.3 can be solved using a simple iterative worklist algorithm that propagates liveness information among the basic blocks of the CFG. The liveness sets are refined until a fixed point is reached. When the worklist contains CFG edges, the number

of set operations can be bounded by $O(|E||W|)$ [32], as each set can be modified (grow) at most $|W|$ times. As recalled in Section 2, another bound can be derived for the *round robin* algorithm [21,23], based on the *loop connectedness* $d(G,T)$ of the reverse CFG G, i.e., the maximal number of back edges (with respect to a depth-first spanning tree T) in a cycle-free path in G. The algorithm traverses the complete CFG on every iteration, at most $(d(G,T)+3)$ times, and thus results in $O(|E|(d(G,T)+3))$ set operations. These operations are mainly unions of sets, which can be performed in $O(|W|)$ for bitsets or ordered lists.

In addition, both need to precompute the upward-exposed uses and definitions of each basic block. This requires visiting every instruction once, thus in time $O(|P|)$ where $|P|$ is the size of the program representation. Thus, for bitsets, the overall complexity is either $O(|P|+|E||W|^2)$ or $O(|P|+|E||W|(d(G,T)+3))$ depending on the update strategy. Experiments indicate that a mixed approach combining the worklist and the round-robin principles performs best in practice [15]. In comparison, our liveness data-flow algorithm for strict SSA has complexity $O(|P|+|E||W|)$, thus is near-optimal as it includes the time to read the program, $O(|P|)$, and the time to propagate/generate the output, $O(|E||W|)$.

Path-Exploration Approaches. As Appel's path exploration [3, p. 429], the bottom-up logic approach of McAllester [29] works at the granularity of instructions, each variable considered independently. Its complexity is $O(|N||W|)$, for $|N|$ instructions, assuming that instructions have at most two successors, i.e., $|E| \leq 2|N|$, and at most two uses and one definition, thus $|P|$ is $O(|N|)$. With such assumptions, our complexity bound $O(|P|+|E||W|)$ is thus also $O(|N||W|)$. But the converse is not true if $|N|$ is not $O(|E|)$, i.e., working at the granularity of basic blocks gives a better complexity when basic blocks are large ($E \ll N$).

A direct generalization of McAllester's result to programs appearing in actual compilers – e.g., with Horn formulae expressed at the granularity of instructions and solved by the algorithm exposed by Minoux [30] – would lead to a sub-optimal complexity $O(|P||W|)$. Actually, it is important to avoid traversing the program multiple times to get $O(|P|)$ and not $O(|P||W|)$, or, even worse, a complexity that depends on the total number of variables, and not just non-local variables. In [7], we showed how to design optimized algorithms based on path exploration, operating at the basic block level, with complexity $O(|P|+|E||W|)$. Due to space limitations, we do not detail them here but we compare them to our novel data-flow algorithm in the experimental section (Section 5). In brief, compared to such path-exploration algorithms, the main interest of our loop-forest algorithm is that it operates directly on sets, i.e., all live variables at the same time, which leads to better locality and faster operations using bitsets.

4 Computing Liveness Sets Using Loop-Nesting Forests

Instead of computing a fixed point, we now show that liveness information can be derived in two passes over the blocks of the CFG by exploiting properties of strict SSA. The first version of the algorithm requires the CFG to be reducible.

We then show that arbitrary CFGs can be handled elegantly and with no additional cost, except for a cheap preprocessing step on the loop-nesting forest.

4.1 Liveness Sets on Reducible Control-Flow Graphs

The algorithm proceeds in two steps. This will be true for irreducible CFGs as well, with a slight modification described in Section 4.2. These two steps are:

1. A backward pass propagates partial liveness information, bottom up, using a postorder traversal of the CFG.
2. The partial liveness sets are then refined by traversing the loop-nesting forest, propagating liveness from loop-headers down to all basic blocks within loops.

Algorithm 1 shows the initialization to compute liveness in two passes.

Algorithm 1. Two-passes liveness analysis: reducible CFG

```
1: function Compute_LiveSets_SSA_Reducible(CFG)
2:     for each basic block B do
3:         mark B as unprocessed
4:     DAG_DFS(R)                ▷ R is the CFG root node (denoted r in Section 3.1)
5:     for each root node L of the loop-nesting forest do
6:         LoopTree_DFS(L)
```

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. Loop-edges are not considered during this traversal (Line 2). The next phase, traversing the loop-nesting forest, is shown by Algorithm 3. The live-in and live-out sets of all basic blocks in a loop are unified with the liveness sets of its loop-header. This is sufficient to compute valid liveness information because a variable whose live-range crosses a back-edge of the loop is live-in and live-out in all blocks of the loop (see Section 4.1).

Algorithm 2. Partial liveness, with postorder traversal

```
1: function DAG_DFS(block B)
2:     for each S ∈ succs(B) if (B, S) is not a loop-edge do
3:         if S is unprocessed then DAG_DFS(S)
4:     Live = PhiUses(B)         ▷ Variables used by φ-functions in B's successors.
5:     for each S ∈ succs(B) if (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. Propagate 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_N) \ PhiDefs(B_N)
5:         for each M ∈ LoopTree_children(N) do ▷ Visit children in the loop forest
6:             Let B_M = Block(M)                      ▷ Loop-header or block
7:             LiveIn(B_M) = LiveIn(B_M) ∪ LiveLoop
8:             LiveOut(B_M) = LiveOut(B_M) ∪ LiveLoop
9:             LoopTree_DFS(M)
```

Complexity. In contrast to iterative data-flow algorithms, our algorithm has only two phases. The first traverses the CFG once, the second traverses the loop-nesting forest once. The CFG traversal of Algorithm 2 performs $O(|V| + |E|)$ unions of sets and $O(|P|)$ set insertions. Thus, assuming $|V| - 1 \leq |E|$, the complexity of the first phase is $O(|E||W|+|P|)$ for bitsets. The size of the forest is at most twice the number of basic blocks $|V|$ in the CFG, because every loop node in the loop-nesting forest has at least one child node representing a basic block (a forest leaf). Thus, the loop-forest traversal in Algorithm 3 induces $O(|V|)$ set (union) operations. Since $|V| - 1 \leq |E|$, this phase does not change the overall complexity mentioned above. The same is true for the unmark initialization phase. Our non-iterative data-flow algorithm has thus the expected near-optimal complexity $O(|P|+|E||W|)$, as claimed before. It avoids the multiplicative factor that bounds the number of iterations in standard iterative data-flow algorithm.

Correctness. The previous algorithms were specialized for the case where ϕ-functions are interpreted as parallel copies at the preceding CFG edges. For the correctness proofs, we resort to the following, more generic, ϕ-semantics. A ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$ at basic block B_0, receiving its arguments from blocks B_i, $i > 0$, is represented by a fresh variable a_ϕ, a copy $a_0 = a_\phi$ at B_0, and copies $a_\phi = a_i$ at B_i, for $i > 0$. Now, with respect to this ϕ-function, a_i, $i > 0$, is not live-out at B_i and a_0 is not live-in at B_0 anymore. As for a_ϕ, since it is not an SSA variable, it is not covered by the following lemmas. But its live-range is easily identified: it is live-in at B_0 and live-out at B_i, $i > 0$, and nowhere else. Other ϕ-semantics extend the live-ranges of the ϕ-operands with parts of the live-range of a_ϕ and can thus be handled by locally refining the live-in and live-out sets. This explains why, in Algorithm 2, PhiUses(B) is added to LiveOut(B) (Line 4), PhiDefs(B) is added to LiveIn(B) (Line 11), and PhiDefs(S) is removed from LiveIn(S) (Line 6). This ensures that the variable defined by a ϕ-function is marked as live-in and its uses as live-out at the predecessors. A similar adjustment appears on Line 4 of Algorithm 3.

The first pass propagates the liveness sets using a postorder traversal of the *reduced graph* $\mathcal{F}_\mathcal{L}(G)$, obtained by removing all loop-edges[2] from G. The

[2] Again, for a reducible CFG, the loop forest and the loop-edges are uniquely defined.

following two lemmas characterize the variables that may not be marked as live-in to a block after this pass. Detailed proofs are provided in a research report [7].

Lemma 1. *Let G be a reducible CFG,* v *an SSA variable, and d its definition. If L is a maximal loop not containing d, then* v *is live-in at the loop-header h of L iff there is a path in $\mathcal{F}_{\mathcal{L}}(G)$, not containing d, from h to a use of* v*.*

Lemma 2 covers the case when no loop L satisfies the conditions of Lemma 1.

Lemma 2. *Let G be a reducible CFG,* v *an SSA variable, and d its definition. Let p be a node of G such that all loops containing p also contain d. Then* v *is live-in at p iff there is a path in $\mathcal{F}_{\mathcal{L}}(G)$, from p to a use of* v*, not containing d.*

Algorithm 2, which propagates liveness information along the DAG $\mathcal{F}_{\mathcal{L}}(G)$, can only mark live-in variables that are indeed live-in. Moreover, if, after this propagation, a variable v is missing in the live-in set of a CFG node p, Lemma 2 shows that p belongs to a loop that does not contain the definition of v. Let L be such a maximal loop. According to Lemma 1, v is correctly marked as live-in at the header of L. The next lemma shows that the second pass (Algorithm 3) correctly adds variables to the live-in/live-out sets where they are missing.

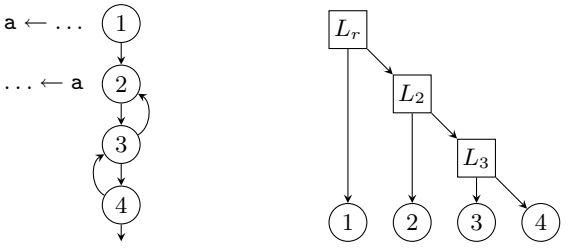
Lemma 3. *Consider a reducible CFG, L a loop, and* v *an SSA variable. If* v *is live-in at the loop-header of L, it is live-in and live-out at every CFG node in L.*

This lemma proves the correctness of the second pass, which propagates the liveness information within loops. Every CFG node, which is not yet associated with accurate liveness information, is properly updated by this second pass. Moreover, no variable is added where it should not.

Example 1. Figure 1a shows a pathological case for iterative data-flow analysis. The precomputation does not mark variable a as live throughout the two loops. An iteration is required for every loop-nesting level until the final solution is found. In our algorithm, after the CFG traversal, the traversal of the loop forest (Figure 1b) propagates the missing liveness information from the loop-header of L_2 within the loop's body and all inner loops, i.e., blocks 3 and 4 of L_3. □

4.2 Liveness Sets on Irreducible Control-Flow Graphs

It is well-known that every irreducible CFG can be transformed into a semantically *equivalent* reducible CFG, for example, using node splitting [22,1]. The graph may, unfortunately, grow exponentially during the processing [13]. However, when liveness information is to be computed, a relaxed notion of equivalence is sufficient. We first show that every irreducible CFG can be transformed into a reducible CFG, without size explosion, such that the liveness in both graphs is *equivalent*. Actually, there is no need to transform the graph explicitly. Instead, the effect of the transformation can be directly emulated in Algorithm 2, with a slight modification, so as to handle irreducible CFGs.



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

Fig. 1. Bad case for iterative data-flow analysis

For every loop L, *EntryEdges*(L) denotes the set of entry-edges, i.e., the edges leading, from a basic block that is not part of the loop L, to a block within L. *Entries*(L) denotes the set of L's entry-nodes, i.e., the nodes that are 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 *LoopEdges*(L). Given a loop L from a graph $G = (V, E, r)$, we define the graph $\Psi_L(G) = (V', E', r)$ as follows. The graph is extended by a new node δ_L, which represents the (unique) 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 to δ_L and additional edges are inserted leading from δ_L to L's former loop-headers. More formally:

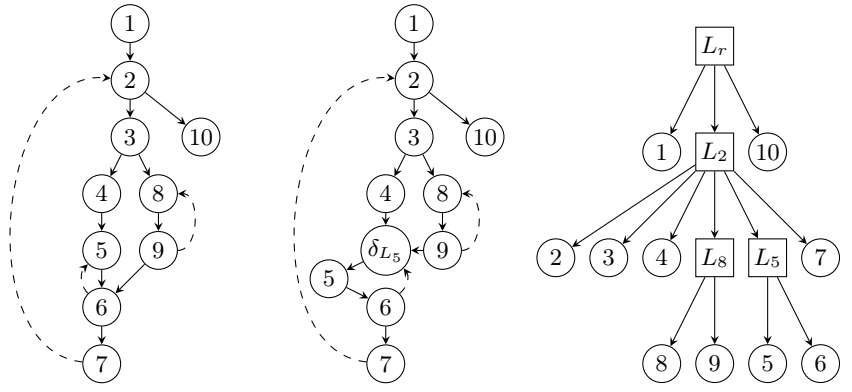
$$E' = E \setminus \mathit{LoopEdges}(L) \setminus \mathit{EntryEdges}(L) \cup \{(s, \delta_L) \mid s \in \mathit{PreEntries}(L)\}$$
$$\cup \{(s, \delta_L) \mid \exists (s, h) \in \mathit{LoopEdges}(L)\} \cup \{(\delta_L, h) \mid h \in \mathit{LoopHeaders}(L)\}$$

Repeatedly applying this transformation yields a reducible graph, slightly larger than the original graph, in which each node is still reachable from the root r. Depending on the order in which loops are considered, entry-edges may be updated several times during the processing in order to reach their final positions. But the loop-forest structure remains the same.

Ramalingam proposed a similar transformation [35, p. 473], which is intended to build an acyclic graph while preserving dominance. It is easy to see that his transformation does *not* preserve liveness and is thus not suited for our purpose.

Example 2. Figure 2c shows a loop forest for the CFG of Figure 2a, where node 5 was selected as loop-header for L_5, the loop containing the nodes 5 and 6. As both nodes are entry-nodes, via the preentry-nodes 4 and 9, the CFG is irreducible. The transformed reducible graph $\Psi_{L_5}(G)$ in Figure 2b might not reflect the semantics of the original program during execution, but it preserves the liveness of the original CFG, for a strict SSA program, as Theorem 1 will show. □

To avoid building this transformed graph explicitly, an elegant alternative is to modify the CFG traversal (Algorithm 2). To make things simpler, we assume



(a) G: Irreducible (b) $\Psi_L(G)$: Reducible (c) A loop-nesting forest

Fig. 2. Transformation of an irreducible CFG using a loop-nesting forest

that the loop forest is built so that, as in Havlak's loop forest construction [19], each loop L has a single[3] loop-header, which can thus implicitly be fused with δ_L. It is then easy to see that, after all CFG transformations, an entry-edge (s, t) is redirected from s to $\mathsf{HnCA}(s, t)$ the loop-header of the *highest non common ancestor* of s and t, i.e., of the highest ancestor of t in the loop forest that is not an ancestor of s. Thus, whenever an entry-edge (s, t) is encountered during the traversal, we just have to visit $\mathsf{HnCA}(s, t)$ instead of t, i.e., to visit the representative of the largest loop containing the edge target, but not its source. To perform this modification, we replace all occurrences of S by $\mathsf{HnCA}(B, S)$ at Lines 3 and 6 of Algorithm 2, in order to handle irreducible flow graphs.

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 of G. Assuming that HnCA is precomputed, the complexity results obtained previously still hold as the number of edges $|E|$ does not change. 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 $O(|V|^2)$. More involved algorithms, as for the lowest common ancestor problem [4], are possible, which process the tree in $O(|V|)$, so that subsequent HnCA queries may be answered in constant time per query. In other words, modifying the algorithm with HnCA to handle irreducible CFGs does not change the overall complexity.

[3] To handle loop forests with loops having several loop-headers, we can select one particular loop-header to be *the* loop representative (B_N in Algorithm 3). But then we need to add edges from this loop-header to any other loop-header.

Correctness. We now prove that, in strict SSA, the liveness of the resulting reducible CFG is equivalent to the liveness of the original one. The following results hold even for a loop forest whose loops have several loop-headers. First, to be able to apply the lemmas and algorithms of Section 4 to the reducible CFG $\Psi_L(G)$, we prove that any definition of a variable still dominates its uses.

Lemma 4. *If d dominates u in G, d dominates u in $\Psi_L(G)$.*

It remains to show that, for every basic block present in both graphs, the live-in and live-out sets are the same. This is proved by the following theorem.

Theorem 1. *Let* v *be an SSA variable, G a CFG, transformed into $\Psi_L(G)$ when considering a loop L of a loop forest of G. Then, for each node q of G,* v *is live-in (resp. live-out) at q in G iff* v *is live-in (resp. live-out) at q in $\Psi_L(G)$.*

5 Experiments

As previously shown, the theoretical complexity of our non-iterative liveness algorithm is near-optimal: it includes the time to read the program, $O(|P|)$, and the time to propagate/generate the output, $O(|E||W|)$. Moreover, variables are added to liveness sets only when actually needed. However, it is still important to evaluate the runtime of the algorithm in practice. We thus compared our algorithm with state-of-the-art approaches, namely an optimized iterative data-flow algorithm, following the "single stack worklist" approach of Cooper et al. [15], and two variants based on path exploration (see end of Section 3.4). The first variant, called use-by-use, traverses the program backwards and, for every encountered variable use, starts a backward depth-first search to find the variable's definition. The variable is added to the live-in and live-out sets along the discovered paths. The other variant, called var-by-var, processes one variable after the other and relies on precomputed def-use chains to find the variable's uses. Both variants are optimized to exploit SSA properties, achieving the near-optimal complexity $O(|P| + |E||W|)$ – for details consult the accompanying report [7].

The algorithms were implemented using the production compiler for the STMicroelectronics ST200 VLIW family, based on GCC as front-end, the Open64 optimizers, and the LAO code generator [17]. We computed liveness relatively late during code generation in the LAO optimizer, shortly before prepass scheduling. For this evaluation, we used bitsets to represent liveness sets, which offer faster accesses, but are often considered to be less efficient in terms of memory consumption and are expected to degrade in performance as the number of variables increases, due to more cache misses and memory transfers. We also investigated the use of ordered pointer-sets, which promise reduced memory consumption at the expense of rather costly set operations. But, as our experiments indicate that bitsets are overall superior [7], we limit our discussion to bitsets.

We applied the various algorithms to the C programs of the SPECINT 2000 benchmark suite to measure the time to compute liveness information. To obtain reproducible results, the execution time was measured using the instrumentation

and profiling tool *callgrind*. These measurements include the number of instructions executed and the memory accesses via caches. Using them, a cycle estimate is computed for the liveness computation only, which minimizes the impact of other compiler components and other running programs on the measurements. As the number of non-local variables depends largely on the compiler optimizations performed before liveness calculation, we investigated the behavior of the various algorithms for optimized and unoptimized programs using the compiler flags `-O2` and `-O0` respectively. All measurements are given relative to the iterative data-flow approach, which performed the worst in all our experiments.

Since most variables are kept in memory at optimization level `-O0`, the number of non-local variables is low (at most 19 in our experiments) and their live-ranges are short. The results (see [7] for details) thus mainly reveal the intrinsic overhead of the different implementations, including artifacts stemming from the host compiler and the host machine. The var-by-var algorithm, which simply iterates over the small set of non-local variables, performs best, as it is the least impacted by the number of basic blocks and operations in the program. The measurements account for the precomputation of the def-use chains, which appears to be less costly than the explicit traversal in the use-by-use algorithm. Our loop-forest algorithm cannot reach the performances of the two path-exploration solutions, which show an average speed-up of 1.80 for the var-by-var algorithm and 1.63 for the use-by-use variant (2.19 and 1.99 compared to iterative data-flow). However, we already observe a speed-up of 1.22 on average in comparison to the state-of-the-art iterative data-flow analysis.

The characteristics of optimized programs are different. The structure of the live-ranges is more complex and the liveness sets are larger. Table 1 shows the number of non-local variables, basic blocks, and operations for the optimized benchmarks. For such programs, the iterative data-flow analysis is still the worst but, now, the var-by-var algorithm is performing worse than the two others, see Figure 3. Our loop-forest approach clearly outperforms both

Table 1. Characteristics of optimized programs

	# 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

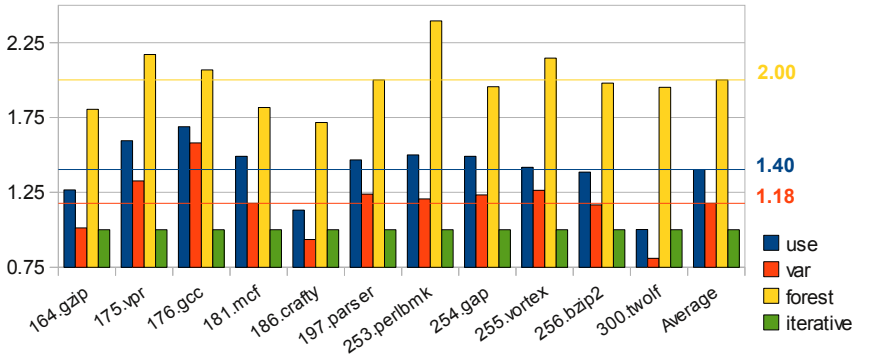


Fig. 3. Speed-up relative to iterative data-flow on optimized programs, with bitsets

path-exploration algorithms, with, on average, speed-ups of 1.69 (= 2.00/1.18) and 1.43 (= 2.00/1.40) respectively. This is explained by the relative cost of the fast bitset operations, in particular set unions, in comparison to the cost of traversing the CFG. Furthermore, the locality of memory accesses becomes a relevant performance factor. Both the use-by-use and our loop-forest algorithms operate *locally* on the bitsets surrounding a given program point. The inferior locality, combined with the necessary precomputation of the def-use chains, explains the poor results of the var-by-var approach in this experimental setting.

More detailed results in comparison to the iterative data-flow approach are given in [7] on a per-module basis, using one data point for every source file. The loop-forest and the use-by-use algorithms on average outperform the iterative one by a factor of 2 and 1.5. Some extreme cases, showing speed-ups by a factor higher than 8, are caused by unusual, though relevant, loop structures in source code generated by the parser generator *bison* (`c-parse.c` of `gcc`, `perly.c` of `perlbmk`), which increase the number of iterations of the iterative algorithm. On the other hand, all cases where the iterative algorithm outperforms the other algorithms are due to implementation artifacts: the analyzed functions do not contain any non-local variables thus slight variations in the executed code, its placement, and the cache state become relevant. The var-by-var approach is often even slower than the iterative one but on average shows a speed-up of 1.18.

Our new algorithm spends most of the time on the backward phase, which amounts to 68% and 53% of the analysis time for unoptimized and optimized programs respectively. The forward phase is almost negligible and contributes only 3% and 6% respectively (with a maximum of 19%). Setting up and initializing data structures, by contrast, takes a large share of 27% and 36% respectively. It is interesting to see that this share increases for optimized programs, due to the large number of variables. The time to construct the loop forest was excluded from these statistics, because loop information is needed for other optimizations anyway, e.g., register allocation, code motion, if-conversion. The naive (quadratic) loop-forest construction available in our framework takes only about 15% of the time of our new liveness algorithm, for optimized programs.

6 Conclusion

Liveness analysis is the basis for many compiler optimizations. However, code transformations often invalidate this information, which has to be repeatedly recomputed. Fast algorithms are thus required to minimize its overhead.

This work proposes an improvement to the traditional iterative data-flow analysis for programs in strict SSA form, which consists of *only two phases*. The first phase resembles the precomputation phase of the standard approach providing partial liveness sets. The second pass replaces the traditional iterative refinement by a *single* traversal of a loop-nesting forest of the control-flow graph.

Our algorithm has the same theoretical complexity as optimized techniques based on path exploration that we developed for comparison. But it operates directly on sets, i.e., all live variables at the same time, and thus is more likely to offer better locality and faster operations using bitsets. As our experiments show, our loop-forest algorithm outperforms the iterative method by a factor of 2 on average for the SPECINT 2000 benchmark suite. Also, for optimized codes, having a large number of non-local variables and complex control flow, our loop-forest approach outperforms by a factor at least 1.43 the path-exploration techniques we proposed, whereas, for unoptimized codes, having very few non-local variables, the path-exploration algorithms appear to be suited best.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (2006)
2. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Communications ACM 19(3), 137 (1976)
3. Appel, A.W., Palsberg, J.: Modern Compiler Implementation in Java, 2nd edn. Cambridge University Press (2002)
4. Bender, M., Farach-Colton, M.: The LCA Problem Revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
5. Boissinot, B., Darte, A., Dupont de Dinechin, B., Guillon, C., Rastello, F.: Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In: Symp. on Code Generation and Optimization (CGO 2009), pp. 114–125. ACM (2009)
6. Boissinot, B.: Towards an SSA-Based Compiler Back-End: Some Interesting Properties of SSA and its Extensions. PhD thesis, ENS-Lyon (September 2010)
7. Boissinot, B., Brandner, F., Darte, A., de Dinechin, B.D., Rastello, F.: Computing liveness sets for SSA-form programs. TR Inria RR-7503 (2011)
8. Boissinot, B., Brisk, P., Darte, A., Rastello, F.: SSI properties revisited. ACM Transactions on Embedded Computing Systems, Special Issue on Software and Compilers for Embedded Systems (2009) (to appear)
9. Boissinot, B., Hack, S., Grund, D., de Dinechin, B.D., Rastello, F.: Fast liveness checking for SSA-programs. In: IEEE/ACM Symposium on Code Generation and Optimization (CGO 2008), pp. 35–44. ACM (2008)
10. Bouchez, F., Darte, A., Guillon, C., Rastello, F.: Register Allocation: What does the NP-Completeness Proof of Chaitin et al. Really Prove? In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) KSEM 2006. LNCS, vol. 4382, pp. 283–298. Springer, Heidelberg (2007)

11. Briggs, P., Cooper, K.D., Harvey, T.J., Simpson, L.T.: Practical improvements to the construction and destruction of static single assignment form. Software—Practice and Experience 28(8), 859–881 (1998)
12. Briggs, P., Torczon, L.: An efficient representation for sparse sets. ACM Letters on Programming Languages and Systems (LOPLAS) 2, 59–69 (1993)
13. Carter, L., Ferrante, J., Thomborson, C.: Folklore confirmed: Reducible flow graphs are exponentially larger. In: ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages (POPL 2003), pp. 106–114. ACM (2003)
14. Click, C., Paleczny, M.: A simple graph-based intermediate representation. In: ACM Workshop on Intermediate Representations, pp. 35–49. ACM (1995)
15. Cooper, K.D., Harvey, T.J., Kennedy, K.: An empirical study of iterative data-flow analysis. In: 15th International Conference on Computing (ICC 2006), pp. 266–276. IEEE Computer Society, Washington, DC, USA (2006)
16. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadek, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (1991)
17. de Dinechin, B.D., de Ferrière, F., Guillon, C., Stoutchinin, A.: Code generator optimizations for the ST120 DSP-MCU core. In: Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2000), pp. 93–102. ACM (2000)
18. Hack, S., Grund, D., Goos, G.: Register Allocation for Programs in SSA Form. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 247–262. Springer, Heidelberg (2006)
19. Havlak, P.: Nesting of reducible and irreducible loops. ACM Transactions on Programming Languages and Systems 19(4), 557–567 (1997)
20. Hecht, M.S., Ullman, J.D.: Characterizations of reducible flow graphs. Journal of the ACM 21(3), 367–375 (1974)
21. Hecht, M.S., Ullman, J.D.: Analysis of a simple algorithm for global data flow problems. In: 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1973), pp. 207–217. ACM (1973)
22. Janssen, J., Corporaal, H.: Making graphs reducible with controlled node splitting. ACM Trans. on Programming Languages and Systems 19, 1031–1052 (1997)
23. Kam, J.B., Ullman, J.D.: Global data flow analysis and iterative algorithms. Journal of the ACM 23(1), 158–171 (1976)
24. Kildall, G.A.: A unified approach to global program optimization. In: Symposium on Principles of Programming Languages (POPL 1973), pp. 194–206. ACM (1973)
25. Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., Cox, D.: Design of the Java HotSpot™ client compiler for Java 6. ACM Transactions on Architecture and Code Optimization 5, 1–32 (2008)
26. Leung, A., George, L.: Static single assignment form for machine code. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 999), pp. 204–214. ACM Press (1999)
27. LibFirm: A library that provides an intermediate representation and optimisations for compilers, `http://pp.info.uni-karlsruhe.de/firm`
28. LLVM: The LLVM compiler infrastructure, `http://llvm.org`
29. McAllester, D.: On the complexity analysis of static analyses. Journal of the ACM 49, 512–537 (2002)
30. Minoux, M.: LTUR: A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. Inform. Processing Letters 29, 1–12 (1988)
31. Mono: NET development framework, `http://www.mono-project.com`
32. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc. (1999)

33. Pereira, F.M.Q., Palsberg, J.: Register Allocation Via Coloring of Chordal Graphs. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 315–329. Springer, Heidelberg (2005)
34. Ramalingam, G.: Identifying loops in almost linear time. ACM Transactions on Programming Languages and Systems 21(2), 175–188 (1999)
35. Ramalingam, G.: On loops, dominators, and dominance frontiers. ACM Transactions on Programming Languages and Systems 24(5), 455–490 (2002)
36. Ryder, B.G., Paull, M.C.: Elimination algorithms for data flow analysis. ACM Computing Surveys 18(3), 277–316 (1986)
37. Sreedhar, V.C., Ju, R.D.-C., Gillies, D.M., Santhanam, V.: Translating out of Static Single Assignment Form. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 194–210. Springer, Heidelberg (1999)
38. Tarjan, R.E.: Testing flow graph reducibility. Journal of Computer and System Sciences 9(3), 355–365 (1974)
39. Wimmer, C., Franz, M.: Linear scan register allocation on SSA form. In: Symp. on Code Generation and Optimization (CGO 2010), pp. 170–179. ACM (2010)