Efficient Liveness Computation Using Merge Sets and DJ-Graphs

Dibyendu Das, Ramakrishna Upadrasta, Benoît Dupont de Dinechin

A

Efficient Liveness Computation Using Merge Sets and DJ-Graphs

DIBYENDU DAS, AMD India Pvt Ltd
BENOÎT DUPONT DE DINECHIN, Kalray Ltd
RAMAKRISHNA UPADRASTA, INRIA

In this work we devise an efficient algorithm that computes the liveness information of program variables. The algorithm employs SSA form and DJ-graphs as representation to build $Merge$ sets. The $Merge$ set of node n, $M(n)$ is based on the structure of the Control Flow Graph(CFG) and consists of all nodes where a ϕ-function needs to be placed, if a definition of a variable appears in n. The merge sets of a CFG can be computed using DJ-graphs without prior knowledge of how the variables are used and defined. Later, we can answer the liveness query (as a part of other optimization or analysis phase) by utilizing the knowledge of the use/def of variables, the dominator tree and the pre-computed merge sets. On average, merge sets have been shown to be of size comparable to the Dominance Frontier(DF) set of a CFG and can be computed efficiently for all kinds of applications consisting of both reducible and irreducible loops. This is an advantage over existing algorithms which require additional complexities while handling applications using irreducible loops. For cases where the merge sets have already been created during the SSA construction step, the cost of our algorithm reduces even further when we use these merge sets for liveness computation. We have compared our new algorithm with a recent algorithm for computing liveness based on SSA form, and show how it performs better in practice, though being simpler to understand and implement.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors-Compilers; D.3.3 [**Programming Languages**]: Language Constructs and Features-Control structures

General Terms: Liveness, Algorithms, Compilers

Additional Key Words and Phrases: Liveness, SSA, Merge Set, Dominator Tree, DJ-graph, Optimizing Compilers

ACM Reference Format:
Das, D., De Dinechin, B., and Upadrasta, R. 2011. Efficient Liveness Computation Using Merge Sets and DJ-Graphs. ACM Trans. Architec. Code Optim. V, N, Article A (January YYYY), 18 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Live variable analysis (or simply liveness analysis) is a classic data flow analysis [Morgan 1998; Cooper and Torczon 2004] performed by compilers to calculate for each program point the variables that may be potentially read before their next write. Thus, liveness information is an important aspect for various optimization phases. Some of the well-known optimization passes that require liveness analysis are register allocation [Chaitin et al. 1981; Briggs and Cooper 1994] and global instruction scheduling (software pipelining, trace scheduling) [Srikant and Shankar 2007], assuming these techniques work directly from the SSA form [Hack et al. 2006]. Classical dataflow-

Part of the work was done when the first author was with IBM. Author's addresses: Dibyendu Das, AMD India Pvt Ltd, No 10 Mantri Chambers, Richmond Road, Bangalore 560025, India, email:dibyendu.das@amd.com; Benoît Dupoint De Dinechin, Kalray Ltd, email:benoit.dinechin@kalray.eu; Ramakrishna Upadrasta, INRIA, email:ramakrishna.upadrasta@inria.fr.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

based iterative liveness analysis uses bitsets to represent live-in/live-out status of a variable at a program point. Given the use/def points of variables, it uses a set of dataflow equations to update the bitsets. Once the bitsets are computed for all program points, the question of whether a variable is live-in/live-out can be easily found using set membership tests on the bitsets. Though such iterative solutions may not be very expensive in practice, optimization phases may add new variables or modify the CFG. This invalidates the liveness information resulting in the entire iterative process to be repeated making the overall process costly.

In a recent work published by Boissinot et al. [Boissinot et al. 2008], an efficient algorithm is presented for fast liveness checking of programs in SSA form using a pre-computation step followed by the actual liveness computation step. The pre-computation step enables part of the liveness computation that is dependent on the topological structure of the CFG, to be stored early. Once, a variable, its use and define points and the point where the liveness question is being asked, are known, the pre-computed information can be combined with these to arrive at the answer quickly. One of the advantages of this method is that the liveness information survives all program transformations except for changes in the CFG.

The idea of a pre-computation step followed by the actual computation step for program analysis has previously been used to efficiently insert ϕ-functions for converting programs to SSA forms by Das et al. in [Das and Ramakrishna 2005]. In that paper the authors pre-compute the merge sets of the nodes of the CFG in a top-down iterative manner utilizing a DJ-graph [Sreedhar and Gao 1995]. This step is followed by the actual ϕ-function insertion step when the variable and its define points are known. The authors also show how such an iterative top-down algorithm is quite efficient in practice outperforming such standard algorithms as [Cytron et al. 1991]. In an earlier work on control dependence computation, Pingali and Bilardi [Pingali and Bilardi 1995] introduced the idea of pre-computing a data structure that is queried repeatedly during the analysis phase.

In the present work we show how the merge sets can also be effectively utilized to carry out liveness analysis in a fast and efficient manner for programs in SSA form. The merge set of a node n (belonging to the CFG) denoted as $M(n)$ can be derived solely from the CFG structure. Loosely, $M(n)$ encodes the set of nodes where multiple paths merge or join. Using this information added with the knowledge of how variables are used and defined, we can compute the liveness information using a two-step method. The first step involves the computation of $M(n)$ for all nodes n of the CFG while the second step involves using these $M(n)$ sets and the use/def of variables to compute liveness.

Our algorithm can handle arbitrary control flow graphs that may include irreducible loops [Havlak 1997; Ramalingam 2002]. In addition, if merge sets have already been used to compute the ϕ-function placement, while converting the program into SSA form, then, our algorithm has a very low-cost pre-computation step, as the merge sets are readily available without any need to re-compute them.

For a detailed introduction and discussion on merge sets one can refer to the work by Pingali and Bilardi [Bilardi and Pingali 2003]. For a description of how these sets can be computed efficiently in a compiler, one can refer to the work [Das and Ramakrishna 2005]. Nevertheless, in this paper we will introduce the merge sets briefly.

1.1. Our Contributions

The contributions of this paper are the following:

— Given a program in SSA form we show how liveness information can be computed simply and efficiently using merge sets. This is the first known instance of merge sets being used for computing SSA-based liveness information.
— Our work provides insight into how merge sets can be visualized as a unifying concept for computing liveness in SSA-form programs with both reducible and irreducible loops, without resorting to additional data structure (like loop nesting forests [Ramalingam 2002]) for handling irreducible loops.
— This work also shows how merge sets provide a scalable mechanism for liveness computation in SSA form due to linear memory requirements (in the size of the CFG) for storing merge sets as opposed to other forms of supporting data-structures which may not scale well with the size of the CFG.
— Finally we present experimental results showing how the new mechanism fares better than the existing mechanisms.

1.2. Paper Organization

The paper is organized as follows. Section 2 provides the preliminary definitions, a motivating example including an overview of merge sets and Boissinot's algorithm. In Section 3 we discuss our new algorithms for computing live-in and live-out for liveness queries at entry/exit of basic blocks. Section 4 contains the correctness proofs. Section 5 deals with experimental results. In Section 6 we provide related work. We end with conclusion and future work in Section 7.

2. PRELIMINARIES

In this section we briefly define the following :

CFG. $CFG = \langle V, E \rangle$ is a tuple consisting of a set of nodes V and a set of directed edges $E \subseteq V \times V$, with a special node $ENTRY(root) \in V$, from which, there exists a path to every node and a special node $EXIT \in V$ to which every node has a path. The $x \xrightarrow{+} y$ notation will represent a path from node x to node y in the CFG which is non-empty. Possibly empty paths will be denoted as $x \xrightarrow{*} y$. [1]

Back-Edge. A back-edge in a CFG is defined to be an edge of the form $u \rightarrow v$, when v is an ancestor of u in the DFS tree of the CFG. A back-edge free path $p \xrightarrow{+} q$ is a path which does not contain any back-edges. Havlak [Havlak 1997] and Ramalingam [Ramalingam 2002] have posited on various interpretations of a back-edge - especially for irreducible graphs. In this work, we will use the definition stated above.

Dominance. A transitive, antisymmetric, reflexive relation on V such that, a node v is said to dominate node w, v **dom** w if every path from $ENTRY$ to w has v. It can be represented as a tree with $ENTRY$ as root. If $v \neq w$, then v is said to strictly dominate w.

Dominator Tree. A dominator tree $DT = \langle V, E_{DomTree} \rangle$ is a tree whose $root$ node is the $ENTRY$ node of the CFG, the nodes are the nodes of the CFG and the edges (denoted as D-edges) are $idom(v) \rightarrow v$ where $idom(v)$ is the immediate dominator of v. The immediate dominator of v is the closest strict dominator on any path from $ENTRY$ to v. Though $idom(v) \rightarrow v$ is a directed edge, the reverse edge $v \rightarrow idom(v)$ is also maintained. This allows us to visualize DT as a undirected tree. [2]

[1] In subsequent diagrams we will omit $ENTRY$ and $EXIT$ nodes for simplicity.
[2] In subsequent diagrams we will omit the $root$ node for simplicity.

Level. Level of a node v is the distance of the node from the $ENTRY$ node in the dominator tree. $ENTRY$ has level 0, its children have level 1 and so on.

J-edges. If $e = (s,t) \in E$ but s does not strictly dominate t, then e is called a J-edge and s is the source and t the target nodes [Sreedhar and Gao 1995]. We will denote a J-edge as $s \xrightarrow{J} t$. A J-edge has the property that $idom(t)$ strictly dominates s and is an ancestor of s in the dominator tree. This follows from Lemma 3.1 in [Sreedhar 1995].

Tree-Path. Let $\langle V,T\rangle$ be a undirected tree. For $v,w \in V$, the notation $[v,w]$ represents the set of vertices on the simple path joining v and w. Similarly, the notation $[v,w)$ represents the set of vertices on the simple path joining v and w, not including w. For example, in the dominator tree of Figure 1(b) (ignoring the J-edges), $[9,3]$ denotes the set $\{9,8,3\}$, while $[6,2)$ denotes the set of nodes $\{6,3\}$. This definition can be found in Pingali and Bilardi [Pingali and Bilardi 1995]. Throughout this paper, the tree in question will be the dominator tree DT.

Shadow(e). Given an edge $e = s \xrightarrow{J} t$, where $idom(t)$ is known to strictly dominate s, $Shadow(s \xrightarrow{J} t)$ is defined as the set of nodes in the dominator tree from node s to node $idom(t)$, excluding node $idom(t)$. In tree-path notation, $Shadow(s \xrightarrow{J} t) = [s, idom(t))$. For example, $Shadow(10 \xrightarrow{J} 8) = [10,3) = \{10,9,8\}$ in Figure 1(b). Also, $Shadow(5 \xrightarrow{J} 6) = [5,3) = \{5\}$.

DJ-graph. Sreedhar et al. [Sreedhar and Gao 1995] define a DJ-graph as a dominator tree with the J-edges added. The DJ-graph is a directed graph and both the D-edges and J-edges are directed edges. Though a D-edge $idom(v) \rightarrow v$ is a directed edge, the reverse edge $v \rightarrow idom(v)$ is also maintained. This helps in traversing up and down the DJ-graph using these bi-directional edges. Though subsequent diagrams will show D-edges as directed in one direction, in reality these are bi-directional edges and will be assumed as such.

S(tatic) S(ingle) A(ssignment) Form. In compiler design, static single assignment form is an intermediate representation (IR) in which every variable is assigned exactly once. Existing variables in the original IR are split into versions, new variables typically indicated by the name with a subscript, so that every definition gets its own version. In SSA form, use-def chains are factored such that every use has a single definition.

I(mmediate) DOM(inator) P(ath). The immediate dominator path of a node n, denoted as $IDOMP(n)$, is the set of nodes lying on the dominator tree from the node n, all the way up to the root node of the dominator tree. Thus $IDOMP(n)$ is the tree-path $[n, root]$ in the dominator tree DT. For example $IDOMP(8) = [8, root] = \{8,3,2,1,root\}$ in Figure 1(b). $IDOMP(X)$ where X is a set of nodes is defined as $IDOMP(X) = \bigcup_{n \in X} IDOMP(n)$.

ϕ-function. When a program is being put in SSA form multiple definitions of the same variable converge at control-flow join points. In order to disambiguate which of the new variables to use, the SSA form introduces the abstract concept of ϕ-functions that select the correct one depending on control flow.

D(ominance) F(rontier). A node v is said to be in $DF(w)$, if a predecessor of v is dominated by w but v is not strictly dominated by w. The transitive closure of DF is referred to as DF^+. This relation can be represented by a graph called the DF-graph.

I(terated) D(ominance) F(rontier). Cytron et al. [Cytron et al. 1991] introduced this method for placing ϕ-functions. Finding the ϕ- points for a set N_α for a variable v is done by seeding the output set with N_α, and adding the DF of each element of the set to it, till no new nodes can be added. Thus, $IDF(x) = DF^+(x)$. Here, N_α is the set of nodes where variable v is defined.

IsLiveIn(n,a). A variable a is live-in at a node n of a CFG, if there exists a path from node n to a node u where a is used and that path does not contain any definition of a.

IsLiveOut(n,a). A variable a is live-out at a node n if it is live-in at a successor of n.

Throughout the paper we will use $def(a)$ to denote the node containing the single dominating definition of a variable a, while $uses(a)$ will denote the set of nodes which contain the uses of variable a, each such node being dominated by $def(a)$ as the program is in SSA form.

2.1. A Motivating Example

In the CFG shown in Figure 1(a) derived from [Boissinot et al. 2008], three variables w, x, y are defined in node 3 while x is used at node 9, y at node 5 and w at node 4. The corresponding DJ-graph and its merge sets are shown in Figure 1(b). The DJ-graph consists of the dominator tree of the CFG with the J-edges superimposed on it. An easy way to visualize the construction of the DJ-graph is to first create the dominator tree of a CFG. Consider $E_{DomTree}$ to be the set of edges in the dominator tree. Find the set $E - E_{DomTree}$ i.e. all the edges in the original CFG which are not edges in the dominator tree. This constitutes the set of J-edges. For example, the back edge from node 7 to node 2 in the CFG transforms to a J-edge in the DJ-graph as it does not figure as an edge in the dominator tree. The entire set of J-edges comprises of the edges $\{7 \xrightarrow{J} 2, 5 \xrightarrow{J} 6, 6 \xrightarrow{J} 5, 4 \xrightarrow{J} 5, 9 \xrightarrow{J} 6, 10 \xrightarrow{J} 8\}$.

If we use Figure 1(a) for the query $IsLiveIn(10, w)$ (i.e. whether the variable w is live at node 10), the answer we should get is a $false$ value. This follows from the fact that the only way to reach a use of w in node 4 from node 10 is via the following path $10 \rightarrow 8 \rightarrow 9 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 3 \rightarrow 4$ in the CFG. But due to variable w being defined in node 3, w is not live at node 10. On the other hand, for the query $IsLiveIn(8, y)$ we see that the path from node 8 that can reach a use of variable y in node 5 in the CFG is $8 \rightarrow 9 \rightarrow 6 \rightarrow 5$ with no $def(y)$ lying in the path. Hence the query should return a $true$ value.

2.2. Merge Set and Merge Relation

Let us define first the notion of a *join* set $J(S)$ for a given set of nodes S in a control flow graph.[3] Consider two nodes u and v and distinct paths from $u \xrightarrow{+} w$ and $v \xrightarrow{+} w$, where w is some node in the CFG ($x \xrightarrow{+} y$ denotes a non-empty path). If the two paths

[3]In English 'join' and 'merge' are synonyms, and in the literature, these two synonyms are used to mean distinct but very-similar and related concepts.

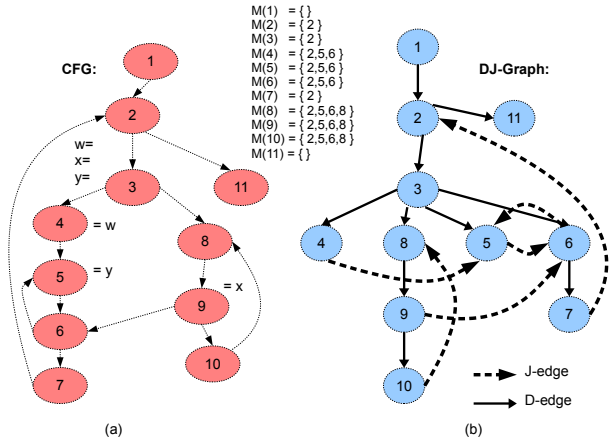

Fig. 1. An example (a) CFG and its corresponding (b) $DJ - graph$ and Merge Sets

meet only at w then w is in the join set of the nodes $\{u, v\}$. For instance, consider nodes 1 and 9 in Figure 1(a). The paths $1 \rightarrow 2 \rightarrow 3 \rightarrow 8$ and $9 \rightarrow 10 \rightarrow 8$ meet at 8 for the first time and so $\{8\} \in J(\{1, 9\})$.

The *merge* relation is defined as a relation $v = M(u)$ that holds between two nodes u and v whenever $v \in J(\{root, u\})$ [Bilardi and Pingali 2003]. We insert a ϕ-function at v for a variable that is assigned at u. For any node $u \in V$, $v \in M(u)$ if and only if there is a path $u \xrightarrow{+} v$ that does not contain $idom(v)$. The merge sets for the nodes of the CFG in Figure 1 are provided. For example, node $2 \in M(7)$ where the paths $root \xrightarrow{+} 2$ and $7 \xrightarrow{+} 2$ meet.

2.3. Merge Sets: Computation and Advantages

$M(n)$ of a node n can be efficiently computed using a top-down iterative pass over the DJ-graph. A simple and efficient algorithm for construcing merge sets, that handles in a unified manner, both the cases of CFGs with reducible and irreducible loops by a top-down iterative construction algorithm has been given by [Das and Ramakrishna 2005]. In the later sections, we will show that these advantages of merge sets, along with simplicity of construction and storing of DJ-graph could reflect well on the liveness computation.

2.3.1. Size of merge sets. One of the important observations on the sizes of merge sets is that for a node n, the average size of $|M(n)|$ is usually a small constant number irrespective of the size of the CFG. Hence, the total storage requirement for the merge sets of a CFG having V nodes is $O(|V|)$. In Table II for the experiments conducted on a set of benchmarks the total merge set storage requirement for a CFG varies from 1.5x to 2.3x of the total size of $|V|$ on an average.

2.3.2. Handling irreducible loops in CFGs. The other observation is that using merge sets removes the need for special cases that are needed for programs with irreducible loops. [Havlak 1997; Ramalingam 2002] use special and non-trivial constructions for handling the back edges in these programs with these kinds of loops.

Irreducible loops create issues as *back edges* are not well-defined in CFGs as shown in [Havlak 1997; Ramalingam 2002]. Identifying back-edge targets for irreducible loops usually requires pre-processing of the CFG. The pre-computation steps

in Boissinot's algorithm [Boissinot et al. 2008] is dependent on the formation of a set called the T_q set which contains all back-edge targets relevant for a liveness query at node q. When irreducible loops appear, this algorithm suffers from increased complexity due to additional pre-processing.

A merge set based pre-computation mechanism does not suffer from these drawbacks as CFGs with reducible and irreducible loops are handled in a unified manner by the top-down iterative merge set construction algorithms. Irreducible loops may lead to multiple top-down passes during merge set construction phase, depending on the number of nodes that make up such irreducible loops [Das and Ramakrishna 2005]. Since the number of nodes which make up an irreducible loop is small in practice, the number of passes are usually constrained to a small number. Also, our liveness query algorithms do not depend on finding back edges of loops.

2.4. Boissinot's Algorithm

In this part we will briefly describe Boissinot's approach to answer the $IsLiveIn/IsLiveOut$ queries for SSA-form programs. As stated in the introduction, the algorithm is divided into two phases. The first phase computes certain path-specific information of the CFG, which is subsequently used in the second phase to answer the liveness queries. For a variable a, assume that the dominating definition is at node d, a use is at node $u \in uses(a)$, and the liveness query is being asked at node q via $IsLiveIn(q, a)$.

Boissinot's algorithm relies on two observations. The first is that a is live-in at q if a back-edge-free path $q \xrightarrow{+} u$, exists (assuming $q \neq u$). Also, d should not be part of this path. Using this observation, Boissinot define a reduced graph $\widetilde{G}$ from G such that it contains all the nodes and edges of G, excepting the back edges. If $x \xrightarrow{+} y$ is a valid path in $\widetilde{G}$, it is said that y is reduced reachable from x. For each node x, the nodes that are reduced-reachable from x are stored in the set R_x. It can be seen that when $u \in R_q$, a is live-in at q.

The second observation derives from paths that contain back-edges. Even if there exists no reduced-reachable path from $q \xrightarrow{+} u$, one may still reach u from q by following a mix of reduced-reachable paths and back-edges. There may be multiple such paths which may reach u from q, but the existence of one where d does not appear suffices to make a live at q. To take care of the presence of back-edges, the relevant back-edge target nodes that may affect the answer of a liveness query at q are stored in the set T_q. A set of recursive data-flow equations are defined to compute T_q as shown below.

$T_q = \bigcup_{i=0}^{\infty} T_q^i$

$T_q^i = \bigcup_{t \in T_q^{i-1}} T_t^{\uparrow}, T_q^0 = \{q\}$

$T_t^{\uparrow} = \{t^{'} \in V \backslash R_t | \exists s^{'} \in R_t \wedge (s^{'}, t^{'}) \in E^{\uparrow}\}$, where, $E^{\uparrow}$ is the set of back-edges.

Boissinot's algorithm now proceeds to the second phase where it answers a query about the live-in status of a variable a at node q. For this, it includes only nodes that belong to T_q which are additionally dominated by d, calling it the $T_{(q,a)}$ set. For every node in $T_{(q,a)}$, if a u is reachable in $\widetilde{G}$, the algorithm returns a $true$ value. If none of the $u \in uses(a)$ is reachable it returns a $false$ value. The live-out status of a variable can be computed similarly.

For the example in Figure 1(a) let us try to evaluate $T_9 = \bigcup_{i=0}^{\infty} T_9^i = T_9^0 \cup T_9^1 \ldots$ which is required in order to answer the $IsLiveIn(9, w)$ query. From the equations, $T_9^0 = \{9\}$. $T_9^1 = \bigcup_{t \in T_9^0} T_t^{\uparrow}$. As $T_9^0 = \{9\}$, $T_9^1 = T_9^{\uparrow}$. $R_9 = \{6, 7, 10\}$. We now need to find $\{t^{'} \in V \backslash R_9 | \exists s^{'} \in R_9 \wedge (s^{'}, t^{'}) \in E^{\uparrow}\}$. $E^{\uparrow} = \{10 \rightarrow 8, 7 \rightarrow 2, 6 \rightarrow 5\}$ is the set of back-edges.

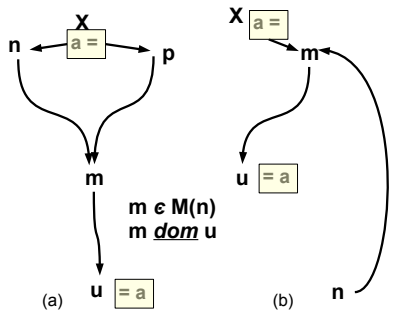

Fig. 2. The intuition behind using Merge Sets for Liveness

$T_9^{\uparrow}$ can now be computed as $\{2, 5, 8\}$ as these are the nodes in $E^{\uparrow}$ whose sources are in R_9 but targets are in $V \backslash R_9$. T_9 will eventually be $\{2, 5, 8\}$. Nodes 5 and 8 are two nodes which are part of T_9 and are dominated by $def(w)$. But from none of these nodes can we reach the $use(w)$ at node 4 using the reduced reachability graph. Hence $IsLiveIn(9, w)$ query returns a $false$ value.

The primary disadvantage of Boissinot's approach is that it is an involved algorithm and can theoretically perform sub-optimally in the presence of irreducible loops. In the next section, we present an algorithm that is theoretically much simpler and performs well practically.

3. NEW ALGORITHM FOR COMPUTING LIVENESS OF VARIABLES

We will now outline new algorithms called the $IsLiveInUsingMergeSet$ and $IsLiveOutUsingMergeSet$ for computing the liveness information of variables at the entry and exit of basic blocks. These can also be used for computing liveness information at arbitrary points inside basic blocks. The algorithms are designed along the lines of the $IsLiveIn$ and $IsLiveOut$ query algorithms as outlined in Boissinot et al. [Boissinot et al. 2008], the main difference being that our algorithms provide for liveness information using the pre-computed merge sets and dominator tree. Our algorithms assume that the merge sets of the nodes of a CFG have been pre-computed and $\forall n \in V$, the set $M(n)$ is available. As stated earlier, if a merge set based ϕ-function placement algorithm is used for SSA construction, these sets may be readily available. In optimizers, where conversion to SSA form happens through more conventional algorithms as [Cytron et al. 1991], merge sets will need to be pre-computed, preferably via the top-down iterative process mentioned earlier.

3.1. Informal reasoning of Using Merge Sets

To clarify the role of merge sets in liveness analysis, assume that we are trying to answer the $IsLiveIn(n, a)$ query. In Figure 2 we show two general cases where the variable a is defined at node X and used at node u and X dominates u (where $u \in uses(a)$). In the case in Figure 2(a), for paths without loops, since n does not dominate u, there are paths $n \xrightarrow{+} m$ and $p \xrightarrow{+} m$ where $p \neq m$. Also m **dom** u holds as given in the figure. Thus, m becomes a merge point for n and $m \in M(n)$. According to $IsLiveIn$, a is live-in at n because the path $n \xrightarrow{+} m \xrightarrow{+} u$ via which u is reachable from n does not have a define of variable a. Thus, $m \in M(n)$ and m **dom** u hold true when $IsLiveIn(n, a)$ is true.

For the case in Figure 2(b), which involves a back-edge, there are paths, $n \xrightarrow{+} m$ and $X \xrightarrow{+} m$. Also m **dom** u holds according to the figure and $m \in M(n)$. According to

ALGORITHM 1: Algorithm for $IsLiveInUsingMergeSet$

```
   Input: Node n, Variable a.
   Output: bool.
1  M^r(n) = M(n) ∪ {n}; // Create a new set from the merge set
2  // Iterate over all the uses of a
3  for t ∈ uses(a) do
4  |  while t ≠ def(a) do
5  |  |  if t ∩ M^r(n) then
6  |  |  |  return true;
7  |  |  end
8  |  |  t = dom-parent(t); // Climb up from node t in the DJ-Graph
9  |  |  // dom-parent returns the parent node in DT
10 |  end
11 end
12 return false ;
```

$IsLiveIn$, the path $n \xrightarrow{+} m \xrightarrow{+} u$ does not have a definition of a and hence a is live-in at n. In this case, too, $m \in M(n)$ and m **dom** u hold true when $IsLiveIn(n, a)$ is $true$. Thus, for both of these cases we see that for a use u, there exists a node m that dominates u and node m belongs to $M(n)$.

For the cases above, n does not dominate u. If there is a single path from n to u, then, n should dominate u. If $def(a)$ does not appear in $n \xrightarrow{+} u$, then $IsLiveIn(n, a)$ is also $true$. Hence, for liveness queries we need to check whether $\exists s \in \{M(n) \cup \{n\}\}$ and s **dom** u. This observation is the basis of our algorithms that follow. Similar observation can be made when $IsLiveIn(n, a)$ is $false$. In such cases, we encounter $def(a)$ before encountering any node in $\{M(n) \cup \{n\}\}$ when we climb up from u using the tree-path $[u, root]$. Note that node n may belong to the set $M(n)$.

We will use the notation $M^r(n)$ to denote $\{M(n) \cup \{n\}\}$. $IsLiveIn(n, a)$ can now be defined as:
$IsLiveIn(n, a) = \exists u \in uses(a) | M^r(n) \cap [u, def(a)) \neq \phi$
where $[u, def(a)) = (IDOMP(u) - IDOMP(def(a)))$.

3.2. Algorithm $IsLiveInUsingMergeSet$

As the firststep of computing $IsLiveInUsingMergeSet(n, a)$, we create the $M^r(n)$ set by adding the node n to the $M(n)$ set as shown in Line 1 of Algorithm 1. In the next step shown in Line 3, we loop over all the nodes in $uses(a)$. For each node $t \in uses(a)$, we climb up the dominator tree if we do not encounter one of the nodes in $M^r(n)$ before encountering the $def(a)$ node. In brief, we check the nodes in the tree-path $[u, def(a))$, where $u \in uses(a)$, to see whether we encounter a node $m \in M^r(n)$. This is depicted in the while loop from Lines 4 to 10. When $[u, def(a)) \cap M^r(n)$ is not an empty set, the variable a is designated live-in at node n and the algorithm returns a $true$ value. After traversing through all the nodes in $uses(a)$ if we do not encounter $M^r(n)$ in any $[u, def(a))$, then a is not regarded as live-in at n and the algorithm returns a $false$ value. The pre-computed (or available) merge sets are preserved and used during this liveness query step.

Let us use the new $IsLiveInUsingMergeSet$ to answer the liveness queries as in Section 2. For the query $IsLiveInUsingMergeSet(10, w)$ we first compute $M^r(10)$. Using the merge sets given in Figure 1 we see that $M^r(10) = \{2, 5, 6, 8, 10\}$. Also, $def(w) = \{3\}, uses(w) = \{4\}$ and $[u, def(a)) = [4, 3] = \{4\}$. If we climb up the dominator tree starting at node 4, the first node we reach is node 3 before we encounter any of the nodes in $M^r(10)$. This implies that the while loop at Line 4 exits and a $false$ value

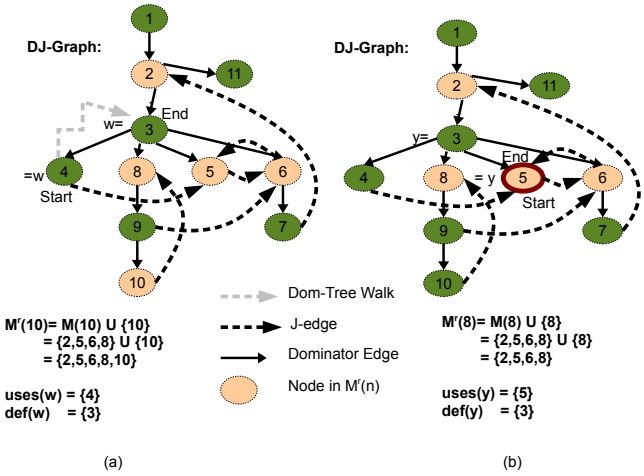

Fig. 3. Working of $IsLiveInUsingMergeSet$

ALGORITHM 2: Algorithm for $IsLiveOutUsingMergesSet$

Input: Node n, Variable a.
Output: bool.

```
1  if def(a) = n then
2  |  return uses(a)\def(a) ≠ φ ;
3  end
4  M_s(n) = φ ;
5  // Iterate over succ(n) - successors of node n
6  for w ∈ succ(n) do
7  |  M_s(n) = M_s(n) ∪ M^r(w) ;
8  end
9  // Iterate over all the uses of a
10 for t ∈ uses(a) do
11 |  while t ≠ def(a) do
12 |  |  if t ∩ M_s(n) then
13 |  |  |  return true ;
14 |  |  end
15 |  |  t = dom-parent(t) ;
16 |  end
17 end
18 return false ;
```

is returned. Hence, node w is not live-in at node 10 as $[4,3) \cap M^r(10)$ is empty. For the query $IsLiveInUsingMergeSet(8,y)$, $M^r(8) = \{2,5,6,8\}$. $def(y) = \{3\}, uses(y) = \{5\}$ and $[u, def(a)) = [5,3) = \{5\}$. Node 5 in the DJ-graph also belongs to $M^r(8)$. Hence $[5,3) \cap M^r(8)$ is non-empty and a $true$ value is returned by the algorithm implying that y is live-in at node 8.

In Figure 3(a) we show the case for $IsLiveInUsingMergeSet(10,w)$. The working of $IsLiveInUsingMergeSet(8,y)$ is depicted in Figure 3(b). The $Start$ and End markers specify the start and end nodes of the upward walks of the dominator tree.

ALGORITHM 3: Algorithm for $ComputeSuccMergeSetsInDJGraph$

Input: GraphNode n, bool $Visited[\ ]$ array.
Output: $M_s(n), \forall n \in DJ - Graph$.

```
1  M_s(n) = φ ;
2  Visited(n) = true ;
3  for w ∈ succ(n) do
4  |   M_s(n) = M_s(n) ∪ M^r(w) ;
5  |   if Visited(w) = false then
6  |   |   ComputeSuccMergeSetsInDJGraph(w) ;
7  |   end
8  end
```

ALGORITHM 4: Algorithm for optimized $IsLiveOutUsingMergeSet$

Input: Node n, Variable a, $M_s(n), \forall n \in DJ - Graph$.
Output: bool.

```
1  if def(a) = n then
2  |   return uses(a)\def(a) ≠ φ ;
3  end
4  // Iterate over all the uses of a
5  for t ∈ uses(a) do
6  |   while t ≠ def(a) do
7  |   |   if t ∩ M_s(n) then
8  |   |   |   return true ;
9  |   |   end
10 |   |   t = dom-parent(t) ;
11 |   end
12 end
13 return false ;
```

3.3. Algorithm $IsLiveOutUsingMergeSet$

In this section we outline how the liveout sets can be computed for variables without computing the live-in sets for all the successor nodes of a basic block where live-out sets are being computed. First, we need to compute the M_s sets using the merge sets of the successor nodes of node n. This is shown in Line 4–8 of Algorithm 2. The algorithm then proceeds to use the same logic as $IsLiveInUsingMergeSet$ i.e. traversing up the dominator tree from the $uses(a)$ set, the only difference being the use of M_s set instead of $M^r(n)$. The details are provided in Algorithm 2. The special case of live-out being computed for the $def(a)$ node can be found in Lines 1–3.

$IsLiveOutUsingMergeSet$ may be invoked for the same node multiple times. Hence, we can optimize Algorithm 2 further by carrying out an initial top-down pass over the DJ-graph to pre-compute the M_s sets as shown in Algorithm 4. This is done using a separate function called $ComputeSuccMergeSetsInDJGraph$ shown in Algorithm 3. The M_s sets can now be used without the need of computing them during each invocation of $IsLiveOutUsingMergeSet$.

3.4. Average Case Complexity of the Liveness Algorithms

The complexity of computing the merge sets is linear on an average as shown in [Das and Ramakrishna 2005]. The complexity of $IsLiveInUsingMergeSet$ is controlled by the outer loop of Line 3 in Algorithm 1, which depends on the size of the use sets of variables. Let $|uses(a)|$ denote the size of the use set for variable a, and $|uses(a)|_{avg}$ is the average number of nodes that need to be traversed in Line 3 of the for loop.

The while loop in Line 4 of the algorithm climbs up the dominator tree traversing the $IDOMP(x) = [x, root]$ where $u \in uses(a)$. The size of $IDOMP(u)$ is controlled by the height of the node u in the dominator tree. Assume that the average height of a node in the dominator tree is given by $h_{avg}^{DomTree}$ and given that the intersection operation in Line 5 takes a small constant time, the complexity of the algorithm is given by $O(|uses(a)|_{avg} * h_{avg}^{DomTree})$. Boissinot et al. show that for a large suite of benchmarks $|uses(a)|_{avg}$ is usually a small constant. Hence the complexity of $IsLiveInUsingMergeSet$ query is $O(h_{avg}^{DomTree})$ for all practical purposes.

$IsLiveOutUsingMergeSet$ has an added complexity in computing M_s sets whose complexity is $O(|V_{succAvg}|)$, where $|V_{succAvg}|$ is the average number of successor nodes of a node. Thus, the total complexity is $O(|V_{succAvg}| + (|uses(a)|_{avg} * h_{avg}^{DomTree}))$ where the second part of the complexity expression is due to Line 5–11 of Algorithm 4. The entire expression simplifies to $O(|V_{succAvg}| + h_{avg}^{DomTree})$ under the assumption of small constant size of $|uses(a)|_{avg}$. Rigorous complexity computation of our algorithms along the lines of [Blieberger 2006] has not been attempted in this paper.

4. CORRECTNESS PROOFS

The following lemmas demonstrate how liveness analysis can be correctly computed using merge sets and DJ-graphs. We will provide the proof of correctness of $IsLiveInUsingMergeSet(n, a)$. The correctness of live-out can be derived along similar lines. For the proofs we will only discuss the general case when there exists a path from node n to $u, u \in uses(a)$ and $n, M(n)$ and u are distinct. For other cases, proofs can be worked out on similar lines.

The basic step used in $IsLiveInUsingMergeSet(n, a)$ is a bottom-up traversal of the DJ-graph from each use $u \in uses(a)$, till an element $m \in M^r(n)$ is found. If found, the algorithm returns a $true$ value. If, instead, a $def(a)$ is encountered, $\forall u \in uses(a)$, then, a $false$ value is returned. Thus, the correctness of the algorithm is dependent on proving whether $IsLiveIn(n, a) = \exists u \in uses(a) | M^r(n) \cap [u, def(a)) \neq \phi$. We will assume absence of any un-initialized variables. Also we will use the terms $root$ and $ENTRY$ interchangeably.

LEMMA 4.1. *In a CFG, for nodes n and u, if there is a path $n \xrightarrow{+} u$ then $\exists m \in M^r(n)$ such that m **dom** u.*

PROOF. If all paths from $root \xrightarrow{+} u$ is of the form $root \xrightarrow{+} n \xrightarrow{+} u$ then n **dom** u holds. Also, by definition, $n \in M^r(n)$. Else, if n does not dominate u there are paths $root \xrightarrow{+} w \xrightarrow{+} u$ and $n \xrightarrow{+} w \xrightarrow{+} u$ such that w **dom** u. This implies $w \in M(n)$. Hence $w \in M^r(n)$. w is assumed to be distinct from u. □

For the lemmas that follow, if multiple nodes $m_1, \ldots, m_k$ dominate u, then we will use that node m such that $\forall m_i, level(m) > level(m_i)$. Thus, $\forall m_i, m_i$ **dom** m and assume m to be distinct from u.

LEMMA 4.2. *In an SSA-form program, for node n and variable a, $IsLiveIn(n, a)$ returns $true \iff \exists u \in uses(a) | M^r(n) \cap [u, def(a)) \neq \phi$*

PROOF. ($\Leftarrow$) : Assume for some $m \in M^r(n)$ there $\exists u \in uses(a)$ such that $m \cap [u, def(a)) \neq \phi$. Hence m dominates u and $def(a)$ dominates m. This is due to two reasons. Firstly, all $uses(a)$ should be dominated by $def(a)$. Secondly, $def(a)$ should dominate m otherwise in the $IDOMP(u)$ tree-path $def(a)$ will appear prior to m. Now if we assume that $IsLiveIn(n, a)$ is $false$, then, $def(a)$ should appear in each and every path from n to u, denoted as $n \xrightarrow{+} u$. Hence any such path should follow the pattern

$n \xrightarrow{+} def(a) \xrightarrow{+} u$ (assume $n, def(a)$ and u are distinct). Also, since $def(a)$ dominates m and m dominates u, the path should follow the pattern, $n \xrightarrow{+} def(a) \xrightarrow{+} m \xrightarrow{+} u$ (assume m and $def(a)$ are distinct). But we know that $m \in M^r(n)$. This means that, there must exist another path from the start node $ENTRY$ to m that bypasses n, otherwise m would not have been in the merge set of n. This implies the existence of a path of the form $ENTRY \xrightarrow{+} m \xrightarrow{+} u$. In such a case, $def(a)$ can no longer dominate u, as we can reach u without passing through $def(a)$. This is a contradiction as we assumed $def(a)$ to dominate $uses(a)$. Hence, $IsLiveIn(n, a)$ has to be $true$ (by contradiction).

($\Rightarrow$) : Assume now that $IsLiveIn(n, a)$ is $true$. This implies that $\exists u \in uses(a)$, such that the path $n \xrightarrow{+} u$ does not have $def(a)$ on that path. We will consider two cases:

Case 1: Assume that node n dominates u. This implies that $n \in IDOMP(u)$. As $n \in M^r(n)$, hence, $M^r(n) \cap IDOMP(u) \neq \phi$. As $def(a)$ does not appear on at least one path $n \xrightarrow{+} u$ by definition of $IsLiveIn$ being $true$, $def(a)$ cannot be dominated by n. On the other hand, if $def(a)$ does not dominate n we can construct an alternate path $ENTRY \xrightarrow{+} n \xrightarrow{+} u$ where $def(a)$ does not appear. This will violate the assumption that $def(a)$ **dom** u. So, $def(a)$ **dom** n holds. $def(a)$ can now be encountered only after n in the reverse walk of the dominator tree upward from u. This implies that $n \in [u, def(a))$. Hence, $\exists u \in uses(a)$, where, $M^r(n) \cap [u, def(a)) \neq \phi$.

Case 2: Assume that node n does not dominate u. But $def(a) \in IDOMP(u)$. Refer to possible scenarios as shown in Figure 2. In general, as Lemma 1 holds, $\exists m \in M^r(n)$ and m **dom** u. By virtue of node m being the first node in the reverse walk of the dominator tree from u, m cannot dominate $def(a)$ as otherwise $IsLiveIn$ will be $false$. This is because all paths from $n \xrightarrow{+} u$ will be of the form $n \xrightarrow{+} m \xrightarrow{+} def(a) \xrightarrow{+} u$. Since this is not allowed, $def(a)$ must dominate m as $def(a)$ **dom** u. Thus, $def(a)$ appears in $IDOMP(u)$ but only after m is encountered in a reverse walk of the dominator tree upward from u. Hence, $\exists u \in uses(a)$, where, $M^r(n) \cap [u, def(a)) \neq \phi$. □

LEMMA 4.3. *In an SSA-form program, for node n and variable a, $IsLiveIn(n, a)$ is $false \iff \forall u \in uses(a) | M^r(n) \cap [u, def(a)) = \phi$.*

PROOF. ($\Leftarrow$) : If $\forall u \in uses(a) | M^r(n) \cap [u, def(a)) = \phi$, it implies that $def(a)$ appears before any of the nodes in $M^r(n)$ is encountered in a tree-path $[u, root]$ from all u in $uses(a)$. Going by Lemma 1, for each $u \in uses(a)$, there is a node $m \in M^r(n)$ that dominates u. For this m, when $m \cap [u, def(a)) = \phi$, then $m \in IDOMP(u)$ and $m \in IDOMP(def(a))$. This implies that both the conditions m **dom** u and m **dom** $def(a)$ are valid. Hence, all paths from n to $u \in uses(a)$ should be of the form $n \xrightarrow{+} m \xrightarrow{+} def(a) \xrightarrow{+} u$ as $m \in M(n)$ by definition of a merge set. This means that $IsLiveIn(n, a)$ returns $false$ as all paths from n to u contain $def(a)$.

($\Rightarrow$) : As $IsLiveIn(n, a)$ returns $false$, for every path $n \xrightarrow{+} u$, where $u \in uses(a)$, the node $def(a)$ appears in the path i.e. $n \xrightarrow{+} def(a) \xrightarrow{+} u$ holds. As $m \in M^r(n)$, there are paths $ENTRY \xrightarrow{+} m$ and $n \xrightarrow{+} m$ which join at m by definition of merge. Now if the path $m \xrightarrow{+} u$ does not have $def(a)$, then the path $n \xrightarrow{+} m \xrightarrow{+} u$ can be constructed that does not have $def(a)$ leading to $IsLiveIn(n, a)$ being $true$. This is a contradiction. So $m \xrightarrow{+} u$ must contain $def(a)$ ($n \xrightarrow{+} m$ cannot contain $def(a)$ else $def(a)$ will not

dominate u). This leads to all paths from m to u taking the form $m \xrightarrow{+} def(a) \xrightarrow{+} u$. As m **dom** u holds, $m \in IDOMP(u)$, then, $m \in IDOMP(def(a))$ also. This results in $[u, def(a)) = \phi$. So, $M^r(n) \cap [u, def(a)) = \phi$ for all $u \in uses(a)$. □

LEMMA 4.4. $IsLiveIn(n, a) = \exists u \in uses(a) | M^r(n) \cap [u, def(a)) \neq \phi$.

PROOF. Follows from Lemma 2 and Lemma 3. □

5. EXPERIMENTS

5.1. Experimental Setup

Liveness checking of the SSA variables is especially useful for dismantling the SSA form, that is, converting back to a non-SSA program representation. This conversion is expensive when it includes coalescing of the copies inserted to replace the ϕ-functions, and such coalescing is required in order to obtain high-quality code. As observed by Sreedhar et al. [Sreedhar et al. 1999], SSA based coalescing removes copies that cannot be removed by traditional coalescing such as Chaitin's algorithm. The need for fast and effective copy coalescing while dismantling the SSA form motivates work by Budimlic et al. [Budimlic et al. 2002], and more recently, work by Boissinot et al. [Boissinot et al. 2009] that generalizes work by Sreedhar et al.

For our experiments, we adapted the Sreedhar et al. [Sreedhar et al. 1999] Method III algorithm, and also his SSA based coalescing algorithm, in order to use liveness checking of variables instead of the classic live-in and live-out sets. We selected the methods of Sreedhar as they are currently the most advanced for copy coalescing and dismantling the SSA form that are also correct, tolerant to non-split critical edges, and described with enough details to be implemented faithfully. The main changes to the Sreedhar Method III for using liveness checking are in the interference breaking phase (step 4) and the actual copy insertion phase (step 6). As in other SSA form dismantling and coalescing techniques [Budimlic et al. 2002], it is only necessary to consider the variables that are operands of the ϕ-functions or operands of the copy operations.

Besides our liveness checking technique and the modifications to the Sreedhar methods, we implemented the Boissinot et al. [Boissinot et al. 2008] method, using the same basic data-structures (dominance tree, bitsets, memory allocators, etc.) as in the implementations of our algorithms. So we provide an unbiased comparison with this technique. As our implementation choice, we represented as bitsets all the sets that contain control-flow nodes, including the dominance frontiers, the merge sets and M_s sets, and the Boissinot T_q(which consists of all back-edge targets relevant for a liveness query at node q) and R_v (which consists of the set of nodes that are reachable from node v for a back-edge free directed graph) sets.

All these implementations were conducted in the STMicroelectronics production compiler for ST200 VLIW family, which is based on the GCC front-ends, the Open64 optimizers, and the LAO code generator [Dupont de Dinechin et al. 2000]. Execution times are provided in milliseconds.

5.2. Compilation Times

We measure compilation time for the following compilation steps:

MergeSets	(MSets) Construction of the Merge Sets using the TDMSC-II algorithm of Das et al. [Das and Ramakrishna 2005], assuming the dominators are already computed.
Setup0	(St0) The setup time of the Boissinot et al. liveness check computation, which includes computation of the T_q and R_v sets.
LiveIn0	(LIn0) The total time spent in live-in checking of Boissinot et al.

LiveOut0 (LOut0) The total time spent in live-out checking of Boissinot et al.
Setup1 (St1) The setup time of our liveness check computation, which includes computation of the M_s sets.
LiveIn1 (LIn1) The total time spent in our live-in checking.
LiveOut1 (LOut1) The total time spent in our live-out checking including computation of M_s sets.

For each series of measures, we compute two ratios:

RatioA (RatA) is (**Setup0 + LiveIn0 + LiveOut0**) divided by (**MergeSets + Setup1 + LiveIn1 + LiveOut1**). This gives the speedup of our method in a pessimistic setting, as we include the cost of computing the merge sets.
RatioB (RatB) is (**Setup0 + LiveIn0 + LiveOut0**) divided by (**Setup1 + LiveIn1 + LiveOut1**). This gives the speedup of our method in a realistic setting, since the cost of computing the merge sets disappears if merge sets are already computed.

Table I. Performance Data for Merge Set based Liveness Computation

Benchmark	MSets	St0	LIn0	LOut0	St1	LIn1	LOut1	RatA	RatB
autcorr.240	0.044	0.171	0.003	0.162	0.054	0.002	0.083	1.83	2.41
bassmgt	0.111	0.430	0.020	0.765	0.141	0.028	0.641	1.32	1.50
compress.2	0.349	1.008	0.124	0.121	0.326	0.092	0.076	1.49	2.54
c-lex	4.674	15.456	0.514	2.480	5.820	0.592	3.266	1.29	1.91
dbuffer	1.697	5.550	0.554	1.933	1.842	0.455	1.459	1.47	2.14
fft32x32s	0.060	0.186	0.090	0.137	0.060	0.094	0.127	1.22	1.48
transfo	0.104	0.401	0.010	0.143	0.136	0.024	0.116	1.45	2.00
TOTAL	7.039	23.202	1.315	5.741	8.379	1.287	5.768	1.35	1.96
g721dec	0.441	1.412	0.008	0.259	0.461	0.011	0.173	1.55	2.60
g721enc	0.431	1.337	0.008	0.259	0.437	0.011	0.173	1.53	2.58
gsm	1.887	7.184	6.730	8.016	2.363	5.816	5.170	1.44	1.64
mipmap	72.569	292.075	9.777	640.193	105.433	14.151	523.023	1.32	1.47
osdemo	73.747	297.928	10.756	643.717	107.519	15.239	526.036	1.32	1.47
pegwit	2.272	9.680	1.039	5.306	3.160	0.861	3.603	1.62	2.10
rasta	5.244	24.183	0.528	42.224	8.383	0.480	35.258	1.36	1.52
texgen	72.808	293.244	9.790	642.496	105.828	14.163	524.562	1.32	1.47
TOTAL	229.399	927.043	38.636	1982.470	333.584	50.732	1617.998	1.32	1.47

The results for several benchmarks are displayed in Table I. The upper part of the table contains codes from the compiler regression base. The lower part of the table are Mediabench [Lee et al. 1997] benchmarks. Our algorithm fares better on an average when compared to the algorithm presented by Boissinot et al. due to much lesser precomputation time. The query times are comparable for both the methods. For some benchmarks the query times for Boissinot are faster than our method - mostly for live-in queries. Our method is almost always faster on live-out queries. For the benchmarks evaluated, we are faster by about 1.3x even when the merge set computation time is taken into account (RatioA). Without the merge set computation time, our algorithm performs around 1.5x-2x times faster (RatioB).

It may be noted that due to the low number of functions(variables) in some of the benchmarks, the compile times are low when compared to bigger ones like mipmap, osdemo and texgen. The compilation times reported by Boissinot et al. is also of the order of microseconds to milliseconds.

Though we have not measured it explicitly, we speculate that the compilation time spent in liveness computation in SSA form programs may not exceed 5-10% of the total compilation time of the program [Puzović 2007]. It may be higher in programs

where high optimization levels and very aggressive inlining are employed that may create very large functions. Similarly, the memory requirement of the SSA-based liveness computation may not exceed 5-10% of the total memory requirement of the entire compilation.

5.3. Benchmark Parameters

Using instrumentation inserted in the compiler, we measured various parameters for each benchmark considered in the previous section:

#F	The cumulative number of functions contained in the benchmark.
#V	The cumulative number of SSA variables considered by the Sreedhar III algorithms.
#B	The cumulative number of basic blocks seen by the Sreedhar III algorithms.
MergeSizes	(MgSz) The total number of elements in the merge sets.
MsaSizes	(MsaSz) The total number of elements in the M_s sets.
HqSizes	(HqSz) The total number of elements in the $T_t^{\uparrow}$ sets of Boissinot et al. liveness checking. This set is a subset of T_q with some additional properties of reachability.
TqSizes	(TqSz) The total number of elements in the T_q sets of Boissinot et al. liveness checking.
RvSizes	(RvSz) The total number of elements in the R_v sets of Boissinot et al. liveness checking.
Ratio	is (**HqSizes + TqSizes + RvSizes**) divided by **MsaSizes**.

Table II. Parameters and sizes of various sets used in our experiments

Benchmark	#F	#V	#B	MgSz	MsaSz	HqSz	TqSz	RvSz	Ratio
autcorr.240	1	82	28	56	84	18	46	392	5.43
bassmgt	1	127	63	199	262	42	105	1966	8.06
compress.2	7	231	160	319	485	67	242	1710	4.16
c-lex	7	710	984	2416	3450	272	1257	82604	24.39
dbuffer	14	1144	744	1603	2363	238	983	16740	7.60
fft32x32s	1	100	30	68	98	14	44	418	4.86
transfo	3	125	64	80	144	18	82	596	4.83
TOTAL	34	2519	2073	4741	6886	669	2759	104426	15.66
g721dec	17	1344	221	306	542	68	289	3587	7.28
g721enc	13	1148	209	309	528	62	271	3658	7.56
gsm	57	7961	1143	1369	2564	284	1428	22337	9.38
mipmap	638	247728	27547	40598	70903	9765	37313	1062017	15.64
osdemo	650	253690	28170	41652	72605	10419	38590	1087342	15.65
pegwit	90	11392	1596	1921	3575	489	2086	22684	7.07
rasta	64	25373	2980	9327	12579	3033	6014	118711	10.16
texgen	654	249686	27745	40780	71299	9863	37609	1063953	15.59
TOTAL	2183	798322	89611	136262	234595	33983	123600	3384289	15.10

As a general comment, the method of Boissinot et al. computes three auxiliary sets of control-flow nodes per basic block, whereas our method computes only the M_s sets beyond the merge sets. The $M^r(n)$ sets in our method need not be explicitly computed, as testing membership of t is same as $t == n$ or $t \in M(n)$. We can observe from Table II that the average storage requirement in our algorithm is considerably lesser - of the order of 15x.

6. RELATED WORK

Liveness analysis is a classical backward data-flow based analysis technique [Cooper and Torczon 2004; Morgan 1998]. One of the first attempts at solving the liveness problem using SSA is by Choi et al [Choi et al. 1991]. They use Sparse Data Flow Evaluation Graphs (SEG) to solve the liveness problem by first noting that ϕ-functions obscure liveness properties of variables and then using SEGs to compute liveness. The idea behind sparse evaluation graphs is to construct a smaller graph from the original graph G, from whose solution the solution of the original graph can be recovered [Ramalingam 1997]. Gerlek et al. [Gerlek et al. 1994] chain the ϕ-functions and create a graph representation from which strongly connected components are extracted to get the liveness information.

In a recent work Boissinot et al. derive an efficient mechanism to compute liveness information for variables. In this work they extract liveness information for SSA-form programs. They employ precomputation and dominator tree in their work in order to speed up liveness queries. However, in this algorithm the handling of CFGs with irreducible loops is not straightforward. And the space requirement of this algorithm is high which subsequently reduce the scalabilty of the algorithm. Our work is very closely related to this work. We also compute liveness information for SSA-form programs. However, we use merge sets which allow us to handle CFG with irreducible loops using a single unified algorithm. It also allows our algorithm to be scalable, simple to understand and implement. This is also the first instance of the application of merge sets for liveness computation.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented a novel approach to liveness analysis for SSA-form programs using merge sets. This is the first application of merge sets for liveness analysis. Merge sets have been used earlier for computing the ϕ-functions of programs as part of the SSA transformation. The advantages of our method compared to previous approaches lie in using the merge sets to handle CFGs consisting of reducible or irreducible loops in a unified and consistent manner. This makes our approach much cleaner and simpler. In addition, storage of merge sets in CFGs take up much lesser space compared to existing methods.

One of the future works is to study whether such merge-set based techniques can be adaped for computing liveness for programs not in SSA form. Also, in the liveness queries, time is spent to account for the fact that ϕ-function arguments are in fact used at the end of the corresponding predecessor basic blocks, instead of the beginning of the basic block where the ϕ-function textually appears. This is a classic rule for the liveness of ϕ-function arguments. Future work will also aim at reducing this overhead if possible. In addition, we may need to evaluate the compile-time and memory overheads of the SSA-based liveness computation phase when compared to the entire compilation flow.

ACKNOWLEDGMENTS

The authors would like to thank Albert Cohen of INRIA and Vugranam Sreedhar of IBM Research for various kinds of support and the anonymous referees for useful suggestions that improved the quality of the work.

REFERENCES

BILARDI, G. AND PINGALI, K. 2003. Algorithms for Computing the Static Single Assignment Form. *J. ACM 50(3) May*, 375 – 425.

BLIEBERGER, J. 2006. Average Case Analysis of DJ-Graphs. *J. Discrete Algorithms 4,* 4, 649–675.

BOISSINOT, B., DARTE, A., RASTELLO, F., DUPONT DE DINECHIN, B., AND GUILLON, C. 2009. Revisiting Out-of-SSA Translation for Correctness, Code quality and Efficiency. In *In Proceedings of the 2009 International Symposium on Code Generation and Optimization (CGO)*. 114 – 125.

BOISSINOT, B., HACK, S., GRUND, D., DUPONT DE DINECHIN, B., AND RASTELLO, F. 2008. Fast Liveness Checking for SSA-Form Programs. In *In Proceedings of 2008 International Symposium on Code Generation and Optimization (CGO)*. 35 – 44.

BRIGGS, P. AND COOPER, K. D. TORCZON, L. 1994. Improvements to Graph Coloring Register Allocation. *ACM Trans. on Programming Languages and Systems 16*, 428– 455.

BUDIMLIC, Z., COOPER, K. D., HARVEY, T. J., KENNEDY, K., OBERG, T. S., AND REEVES, S. W. 2002. Fast Copy Coalescing and Live-Range Identification. In *In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. 25 – 32.

CHAITIN, G. C., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. W. 1981. Register Allocation Via Coloring. *Computer Languages 6(1)*, 47 – 57.

CHOI, J. D., CYTRON, R., AND FERRANTE, J. 1991. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *In Proceedings of 18th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 55 – 66.

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

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. on Programming Languages and Systems 13(4)*, 451 – 490.

DAS, D. AND RAMAKRISHNA, U. 2005. A Practical and Fast Iterative Algorithm for ϕ-function Computation Using DJ-Graphs. *ACM Trans. on Programming Languages and Systems 27(3)*, 426 – 440.

DUPONT DE DINECHIN, B., DE FERRIERE, F., GUILLON, C., AND STOUTCHININ, A. 2000. Code Generator Optimizations for the ST120 DSP-MCU Core. In *In Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 93 – 102.

GERLEK, M., WOLFE, M., AND STOLZ, E. 1994. A Reference Chain Approach for Live Variables. Tech. rep., Oregon Graduate Institute of Science and Technology.

HACK, S., GRUND, D., AND GOOS, G. 2006. Register Allocation for Programs in SSA form. In *Compiler Construction*, A. Zeller and A. Mycroft, Eds. Springer-Verlag,New York.

HAVLAK, P. 1997. Nesting of Reducible and Irreducible Loops. *ACM Trans. on Programming Languages and Systems 19(4)*, 557 – 567.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. 330 – 335.

MORGAN, R. 1998. *Building an Optimizing Compiler*. Digital Press.

PINGALI, K. AND BILARDI, G. 1995. APT: A Data Structure for Optimal Control Dependence Computation. *SIGPLAN Not. 30*, 32–46.

PUZOVIĆ, N. 2007. SSA Form in an Embedded compiler. Tech. rep., HiPEAC/STMicroelectronics.

RAMALINGAM, G. 1997. On Sparse Evaluation Representations. In *In Proceedings of 4th Int'l Symposium on Static Analysis*. 1 – 15.

RAMALINGAM, G. 2002. On Loops, Dominators and Dominance Frontiers. *ACM Trans. on Programming Languages and Systems 24(5) Sep*, 455 – 490.

SREEDHAR, V. C. 1995. Efficient Program Analysis Using DJ Graphs. Ph.D. thesis, McGill University.

SREEDHAR, V. C. AND GAO, G. R. 1995. A Linear Time Algorithm for Placing ϕ-nodes. In *In Proceedings of 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 62 – 73.

SREEDHAR, V. C., JU, R. D., GILLIES, D. M., AND SANTHANAM, V. 1999. Translating Out of Static Single Assignment Form. In *In SAS '99: Proceedings of the 6th International Symposium on Static Analysis*. 194 – 210.

SRIKANT, Y. N. AND SHANKAR, P., Eds. 2007. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press.