More precise construction of static single assignment programs using reaching definitions

Abu Naser Masud[1,*], Federico Ciccozzi[1]

[a]*School of Innovation, Design and Engineering, Mälardalen University, Västerås,Sweden*

Abstract

The Static Single Assignment (SSA) form is an intermediate representation used for the analysis and optimization of programs in modern compilers. The ϕ-function placement is the most computationally expensive part of converting any program into its SSA form. The most widely-used ϕ-function placement algorithms are based on computing *dominance frontiers* (DF). However, this kind of algorithms works under the limiting assumption that all variables are defined at the beginning of the program, which is not the case for local variables. In this paper, we introduce an innovative ϕ-placement algorithm based on computing *reaching definitions* (RD), which generates a precise number of ϕ-functions. We provided theorems and proofs showing the correctness and the theoretical computational complexity of our algorithms. We implemented our approach and a well-known DF-based algorithm in the Clang/LLVM compiler framework, and performed experiments on a number of benchmarks. The results show that the limiting assumption of the DF-based algorithm when compared with the more accurate results of our RD-based approach leads to generating up to 87% (69% on average) superfluous ϕ-functions on all benchmarks, and thus brings about a significant precision loss. Moreover, even though our approach computes more information to generate precise results, it is able to analyze up to 92.96% procedures (65.63% on average) of all benchmarks with execution time within twice the execution time of the reference DF-based approach.

Keywords: Static single assignment, program optimization, program transformation, reaching definition, dataflow analysis

1. Introduction

Most current compilers and virtual machines, including the well-known GNU Compiler Collection (GCC)[1], the LLVM Compiler Infrastructure (LLVM)[2], and the Java Hotspot[3], use the so-called *static single assignment* (SSA) form as an intermediate representation (IR) of programs. SSA programs are often used for efficient program analysis, transformation, optimization, and efficient register allocation. Programs represented in the SSA form require that each variable is defined

*Corresponding author

Email addresses: `masud.abunaser@mdh.se` (Abu Naser Masud), `federico.ciccozzi@mdh.se` (Federico Ciccozzi)

[1]`https://gcc.gnu.org/`

[2]`http://llvm.org`

[3]`https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html`

exactly once, but it may be used multiple times. Moreover, the variable definition should always appear before its use.

Any straight-line sequence of non-SSA code can be converted to SSA form by using a suitable renaming of the program variables that adheres to the definition of SSA program as shown in Figure 1. However, if the code contains branching instructions, the renaming process becomes complicated by the fact that multiple definitions of a program variable may reach at control flow merge points. For example, the `print` statement in Figure 2 receives two distinct definitions of Y from two different branches of the `if` statement. It may be hard, or even impossible, to statically decide which definition of that variable to use afterwards. Any non-SSA program is transformed to the SSA form by performing the following two steps:

(i) identifying the merge points in the control flow graph (CFG) of the program to place *pseudo-assignments* of the form $x = \phi(x, \ldots, x)$ for each variable x, where multiple distinct definitions of x may arrive through different branches of the control flow, and

(ii) renaming each x such that any assignment or pseudo-assignment to x (i.e., a definition of x) is uniquely renamed and uses the renamed x at each reference of that particular definition.

Each argument of the ϕ-function corresponds to a particular reaching definition of x coming from one of the branches. Thus, a so-called *join set* $J^+(S)$ identifying all those merge points requiring pseudo-assignments for each variable x needs to be constructed, where S is the set of CFG nodes containing assignments to x.

```
X=getInput();          ⇒        X_1 =getInput();
X=X+1;                          X_2 = X_1 + 1;
```

Figure 1: From non-SSA to SSA form of an straight-line program

```
                                X_1 =getInput();
X=getInput();                   if(X_1 > Y_1)
if(X>Y)                            Y_2 = Y_1 + 1;
   Y=Y+1;              ⇒        else Y_3 = X_1;
else Y=X:                       Y_4 = φ(Y_2, Y_3);
print Y;                        print Y_4;
```

Figure 2: From non-SSA to SSA form of a program containing branching instructions

Cytron et al. [1] have carried out pioneering work on the establishment of a pragmatically efficient construction of the SSA form based on computing so called *dominance frontiers* (DF), which are relations among CFG nodes based on dominators[4]. This approach is currently leveraged by most SSA construction algorithms used by modern compilers, such as LLVM. A closer look to Cytron et al.'s approach shows that, on the assumption that computing the aforementioned join set $J^+(S)$ is practically inefficient, an efficient alternative would be instead to compute the iterated dominance frontier $DF^+(S)$. This is an approximation that is possible thanks to the equality

[4]The dominance frontier of a node n is the set of all nodes m such that n dominates an immediate predecessor of m, but n does not strictly dominate m

relation $DF^+(S) = J^+(S)$, that holds when S contains the *Entry* node of the CFG [1, 2]. The set S contains all CFG nodes in which a particular program variable is defined. Since the *Entry* node is considered to be included into S due to $DF^+(S) = J^+(S)$, DF-based SSA construction methods implicitly consider that all program variables are defined at the beginning of the program. This is a limiting restriction and it cannot always hold, especially for local variables, which are mostly declared at the beginning, but defined later in the program. Thus, all DF-based SSA construction algorithms produce superfluous ϕ-functions and hence construct larger SSA programs than necessary.

Our Contribution. In this work, we explore the impact of the seemingly benign equality condition by which S includes the *Entry* node of $DF^+(S) = J^+(S)$. To do so, we provide an algorithm based on computing reaching definitions (RDs) [3] that can accurately compute the join set $J^+(S)$ where we can freely choose the set S of variable definitions. Computing RDs for SSA construction is nontrivial and more complex than computing RDs for program analysis. Our novel approach to compute the $J^+(S)$ set is efficient on most of our benchmarks. By including the *Entry* node into S, then DF-based approaches and ours produce the same number of ϕ-functions.

On the other hand, our algorithm is able to produce more accurate ϕ-functions by considering that only global variables and formal parameters are defined at the *Entry* node of the CFG. Our experiments on a number of benchmarks reveal that DF-based SSA construction approach generates (i) up to 87% and on an average 69% superfluous ϕ-functions compared to the ϕ-functions generated by our RD-based approach. Our approach is applicable to both structured and unstructured programs containing dense or sparse variable definitions. Moreover, along with constructing the SSA form, our RD information can be re-used to optimize the generated SSA program.

Note that this is an invited extended version of the paper entitled "Towards constructing the SSA form using reaching definitions over dominance frontiers" [4] and published at the IEEE International Working Conference on Source Code Analysis and Manipulation. More specifically, this article has been extended as follows:

- We have included a new section with a detailed description of where and how the DF-based approach loses precision in computing ϕ-functions (Section 3).
- We have extended the formal development of RD-based SSA construction in Section 4.2. Three new algorithms (Algorithms 3 - 5) are included to provide a complete and detailed picture of how we perform the computation.
- We have added a new section (Section 5) providing discussion about the proof of correctness of our algorithms and their computational complexity. Section 5.1 provides Theorem 1 and its proof along with some auxiliary lemmas to prove the correctness of our algorithms and Section 5.2 includes Theorem 2, some auxiliary lemmas and their proofs stating the computational complexity of our algorithms.
- We have extended our experimental evaluation by running our solution on six additional benchmarks, which confirmed our positive results (Section 6).

Paper organization. The remainder of the paper is organized as follows. In Section 2 we provide core concepts and terminology upon which our approach is based. In Section 3 we provide a description of precision loss in the DF-based approach. Our RD-based approach is described in detail in Section 4, while a discussion of the its correctness and computational complexity is given in Section 5. The extended experimental evaluation is presented in Section 6. Related works are outlined in Section 7 and the paper is concluded by Section 8 with a summary and future work.

2. Background and Terminology

Definition 1 (Control flow graph (CFG))**.** *The CFG of any given program is a directed graph $G = (N, E, entry)$ where*

- *N is the set of nodes and each node $n \in N$ represents a basic block containing straight-line sequence of code,*
- *$E \subseteq N \times N$ is the set of edges representing the program control flow, and*
- *entry is the unique Entry node representing the starting basic block from where the execution starts.*

Note that the above definition of CFG is intraprocedural. Since SSA construction is usually performed per procedure, we consider only intraprocedural CFGs. We denote an intraprocedural CFG G by (N, E) for brevity. A CFG can be *reducible* [5, 6], when it originates from structured code, and *irreducible* otherwise. The set of successor and predecessor nodes of any CFG node n is denoted by $succ(n)$ and $pred(n)$, respectively. A node $n \in N$ is called a *join* node if $|pred(n)| > 1$. The set of program variables that are defined at a CFG node $n \in N$ is denoted by $def(n)$.

A finite CFG path π of length $k \geq 0$ is the sequence of $k+1$ nodes $n_0, \ldots, n_k$ and denoted by $\pi : n_0 \to n_k$ such that $n_{i+1} \in succ(n_i)$ for all $0 \leq i \leq k-1$. We denote the set of all nodes in π by $N_\pi = \{n_0, \ldots, n_k\}$, and write $n \in \pi$ or $n \in N_\pi$ when n is some node n_i in π. A path $\pi : n_0 \to n_k$ is *non-trivial* if it contains at least two nodes (i.e. $k \geq 1$) and denoted by $\pi : n_0 \overset{+}{\to} n_k$. Sometimes, we abuse the notation and write $\pi - S$ instead of $N_\pi \setminus S$ to indicate the set of nodes that are in N_π but not in S.

Two nontrivial paths $\pi_1 : n_0 \overset{+}{\to} n_k$ and $\pi_2 : m_0 \overset{+}{\to} m_l$ *converge* at node n if the following conditions hold:
(i) $n_0 \neq m_0$,
(ii) $n_k = n$ and $m_l = n$, and
(iii) $n_i = m_j \implies i = k \vee j = l$.
Simply put, paths π_1 and π_2 converge when they start at different nodes and join at node n, while being node-disjoint.

Consider the CFG (N, E) and the set $S \subseteq N$. The join set $J(S)$ includes all join nodes m such that there is a pair of paths $n \overset{+}{\to} m$ and $n' \overset{+}{\to} m$ from distinct nodes $n, n' \in S$ that converge at m. If S is the set of CFG nodes containing definitions of a variable x, then $J(S)$ is the smallest set of join nodes requiring pseudo-assignment to x. As $J(S)$ includes the pseudo-assignments to x, intuitively, we may require pseudo-assignments to x in the set $J(S \cup J(S))$. Thus, it leads to the *iterated join* set $J^+(S)$, which can be computed iteratively as follows:

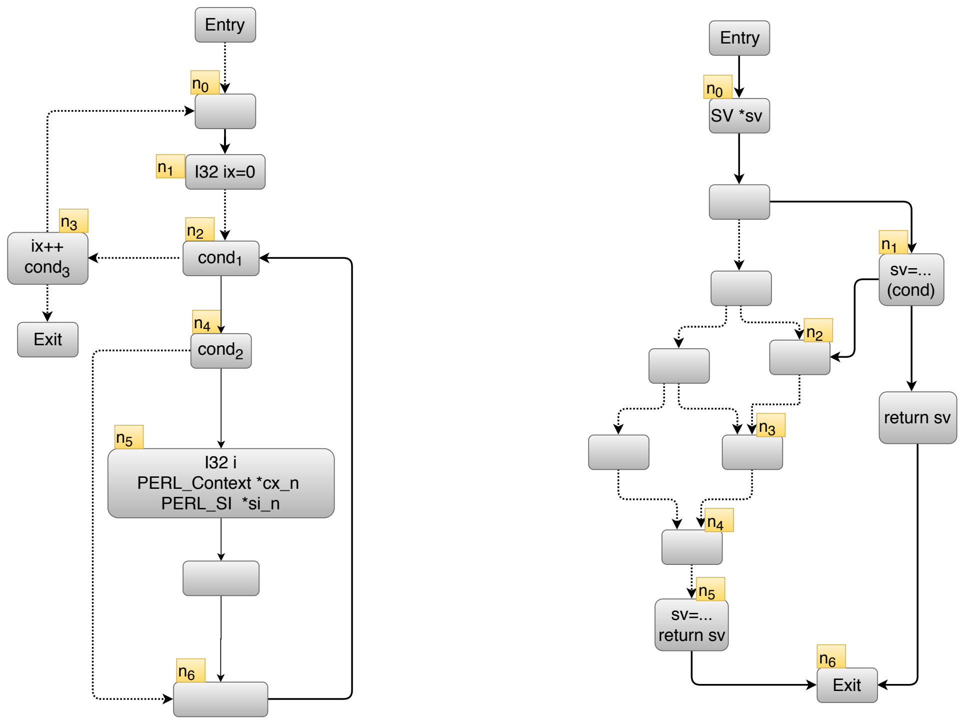
$$\begin{aligned} J^0(S) &= J(S) \\ J^{i+1}(S) &= J(S \cup J^i(S)) \end{aligned}$$

where there exists $k \geq 0$ such that $J^{k+1}(S) = J^k(S)$ and $J^+(S) = J^k(S)$.

Standard SSA construction algorithms rely on the concept of *dominance frontier* in order to identify join nodes that require the pseudo-assignments or placement of ϕ-functions. Node n *dominates* node m (denoted $n\,dom\,m$) if every path from the *entry* node to m passes through n. Node n *strictly dominates* node m if and only if $n\,dom\,m$ and $n \neq m$. The *dominance frontier* of a CFG node n, denoted $DF(n)$, is the set of all CFG nodes m such that n dominates a predecessor of m, but does not strictly dominate m. The *dominance frontier* of a set of nodes S is $DF(S) = \bigcup_{n \in S} DF(n)$. The *iterated dominance frontier* [1] $DF^+(S)$ of a set S of nodes can be obtained by the following iterative computation of DF:

$$\begin{array}{rcl} DF^0(S) & = & DF(S) \\ DF^{i+1}(S) & = & DF(S \cup DF^i(S)) \end{array}$$

and there exists $k \geq 0$ such that $DF^{k+1}(S) = DF^k(S)$ and $DF^+(S) = DF^k(S)$. In [1], Cytron et al. proved that $J^+(S) = DF^+(S)$ if $entry \in S$. Moreover, according to Weiss [2], we have $J(S) = DF^+(S)$, which leads to the conclusion that $J^+(S) = J(S)$. It is thus enough to obtain the smallest set of join nodes $J(S)$ to include ϕ-functions for the variable x defined by the nodes in S.



(a) (b)

Figure 3: CFG skeletons of source code obtained from the 500.perlbench_r benchmark in SPEC CPU2017 [7]. Solid arrows represent CFG edges and dashed arrows represent CFG paths

3. Precision loss of DF-based ϕ-placement methods

DF-based ϕ-placement methods lose precision due to the assumption that all program variables are defined at the beginning. In the following, we illustrate this precision loss with some examples obtained from a real-life benchmark suite SPEC CPU2017 [7].

Consider the CFG skeleton in Figure 3(a). The cycle $\mathtt{n_0, n_1, n_2, n_3, n_0}$ in the CFG represents a loop structure in the program code. Variable `ix` is locally declared inside the loop body at n_1 and defined at n_1 and n_3. This local declaration of the `ix` variable is thus not live at n_0 and n_0 should not require any *pseudo-assignment* to the `ix` variable. However, if we consider that variable `ix` is defined at the *Entry* node in the CFG, then `ix` is live at n_0. Moreover, *Entry* and n_3 are two distinct definitions of `ix` reaching n_0. Thus, node n_0 requires a ϕ-function for the variable `ix`.

There exist two more cycles $\mathtt{n_2, n_4, n_6, n_2}$ and $\mathtt{n_2, n_4, n_5, n_6, n_2}$ in the CFG representing different loop and branching structures in the program code. Variables `i`, `cx_n` and `si_n` are declared at n_5 which are local to these loop and branching instructions. Some nodes in the path from n_5 to n_6 define these variables and these definitions reach n_6. If we assume that these variables are also defined at the *Entry* node which will reach n_6 through n_4, then node n_6 will require ϕ functions for these variables. Moreover, the *pseudo-definitions* of these variables reach n_2 along with the definition at the *Entry* node and consequently, n_2 will require ϕ functions for these variables. However, all these *pseudo-assignments* are not necessary in generating SSA programs and this imprecision in generating ϕ-functions is due to the assumption that the *Entry* node contains definitions of all program variables. Since variable `ix` is not live at n_0 and variables `i`, `cx_n` and `si_n` are not live at n_2 and n_6, the pseudo-definitions of the variables generated at these nodes are also not live. In order to remove these dead ϕ-functions, we need to perform liveness analysis of variables which is computationally expensive.

DF-based ϕ-placement methods may even generate unnecessary live *pseudo-assignments* to variables. Consider the CFG in Figure 3(b) and the variable `sv`. This variable is declared at node n_0 and defined at node n_1. Either the control may reach node n_6 from node n_1 at which the procedure exits by returning `sv` or the control may reach node n_2 from node n_1. Variable `sv` is again defined at node n_5 from which the control reaches node n_6 and the procedure exits by returning `sv`. By considering that `sv` is defined at the *Entry* node, two distinct definitions of this variable reach n_2. Thus, node n_2 requires a ϕ-function and n_2 becomes a new definition of `sv`. Similarly, n_3 and n_4 will require pseudo-definitions for it. However, if we would not consider that this variable is defined at the *Entry* node, then these pseudo-definitions would not be required. Note that variable `sv` is a live variable during the entire control flow of the program and hence these pseudo-definitions are also live. So, these unnecessary ϕ-functions cannot be removed by performing liveness analysis of program variables.

4. SSA Construction Procedure

In this section, we provide methods to compute the join sets requiring ϕ functions without using the concept of dominance frontiers. The method is based on a forward dataflow analysis [3] accumulating facts about reaching variable definitions at different CFG nodes. In Section 4.1, we provide Algorithm 1 to perform the dataflow analysis collecting data flow facts which are called abstract and concrete reaching definitions. In Section 4.2, we develop methods to resolve all abstract definitions to concrete definitions. These concrete definitions are then used to find the join sets requiring ϕ functions. We provide Algorithm 4 (in Section 4.2.4) to resolve the abstract definitions and find the join sets which apply Algorithm 2 (in Section 4.2.2) and 3 (in Section 4.2.3) iteratively during the resolution process. We skip the renaming phase of the SSA construction in this paper.

Consider the CFG $G = (N, E)$ of the input program, the set of *pseudo nodes* N_u of N, and the set $\mathbf{Var}$ of program variables. The dataflow domain consists of the set $\mathcal{P}(N \cup N_u)$ of variable definitions. Thus, the definition of a variable $x \in \mathbf{Var}$ can be a node $n \in N$ at which it is defined in the program code, or a pseudo node $n_u \in N_u$ representing that x may be defined either at n or at another CFG node and that definition reaches n. We call any definition $n \in N$ a *concrete definition* (CD) and $n_u \in N_u$ an *abstract definition* (AD). Instead of performing a fixpoint-based dataflow analysis, we visit each edge exactly once during the dataflow analysis generating facts containing CDs and ADs. Afterwards, all ADs are resolved to CDs by performing flow analyses of concrete RDs. In what follows, we assume that $n \in N$, $n_u \in N_u$, and n_λ can be either n or n_u.

4.1. Forward Dataflow Analysis

The dataflow analysis is based on collecting dataflow facts containing CDs and ADs from the dataflow domain $\mathcal{P}(N \cup N_u)$. Any fact $A \subseteq \mathcal{P}(N \cup N_u)$ for any variable $x \in \mathbf{Var}$ represents that either x is defined at a CFG node $n \in A \cap N$ or there may exist a definition of x that reaches n for any $n_u \in A \cap N_u$. We use the mapping function $\mathcal{A}_\circ(n)$ and $\mathcal{A}_\bullet(n)$ for each CFG node $n \in N$ collecting relevant facts on *reaching definitions.* $\mathcal{A}_\circ(n)$ and $\mathcal{A}_\bullet(n)$ contain facts that are valid at the entry and exit of node n. The following definition specifies the properties of these mapping functions:

Definition 2 (Reaching Definition Functions (RDFs))**.** *Consider any CFG $G = (N, E)$, $n \in N$ and $\star \in \{\circ, \bullet\}$. The RDF $\mathcal{A}_\star(n)$ at the CFG node n maps program variables in* $\mathbf{Var}$ *to a set of dataflow facts in $\mathcal{P}(N \cup N_u)$ (i.e. $\mathcal{A}_\star(n) : \mathbf{Var} \to \mathcal{P}(N \cup N_u)$) such that for any $x \in$* $\mathbf{Var}$ *and $m_\lambda \in \mathcal{A}_\star(n)(x)$ the following holds:*

1. *if $m_\lambda \in N$, then $x \in def(m_\lambda)$,*
2. *$m_\lambda \neq n$ and $m_\lambda \in N$ (i.e., $m_\lambda = m$) imply that there exists a path $\pi : m \xrightarrow{+} n$ in G such that $x \notin def(n')$ for all $n' \in N_\pi \setminus \{m, n\}$ if $\star = \circ$ or $n' \in N_\pi \setminus \{m\}$ if $\star = \bullet$,*
3. *if $m_\lambda \in N_u$ (i.e., $m_\lambda = m_u$) and the AD m_u is resolved to a CD $m' \in N$, then m' satisfies conditions (1) and (2) above.*

So, $\mathcal{A}_\star(n)(x)$ is the set of all reaching definitions of x at n for $\star \in \{\circ, \bullet\}$. Note that we sometimes use the set notation of RDFs instead of the functional notation. Thus, we write $(x, m) \in \mathcal{A}_\star(n)$ instead of $m \in \mathcal{A}_\star(n)(x)$. The analysis collects facts into the set of reaching definitions (RD) $\mathcal{A}_\circ(n)(x)$ and $\mathcal{A}_\bullet(n)(x)$ for all $n \in N$ and $x \in \mathbf{Var}$ according to the following equations:

$$\begin{array}{rcl} \mathcal{A}_\bullet(n) & = & f_n(\mathcal{A}_\circ(n)) \\ \mathcal{A}_\circ(n) & = & \bigcup_{m \in pred(n)} g(m, \mathcal{A}_\bullet(m)) \end{array} \tag{1}$$

where f_n is the transfer function for n, and the function g is defined as follows:

$$g(m, \mathcal{A}_\bullet(m)) = \bigcup_{x \in \mathbf{Var}} \begin{cases} \{(x, m'_\lambda)\} & \text{if } (x, m'_\lambda) \in \mathcal{A}_\bullet(m) \\ \{(x, m_u)\} & \text{otherwise} \end{cases} \tag{2}$$

Any RD m'_λ of x at the exit of m is also the reaching definition of x at the entry of n (first case of g). According to Lemma 1 below, $\mathcal{A}_\bullet(m)$ can contain at most one definition of x and thus the

```
Input : $G = (N, E)$
forall ($n \in N \wedge x \in \mathbf{Var}$) do
    $\mathcal{A}_\circ(n)(x) = \emptyset$
    $\mathcal{A}_\bullet(n)(x) = \emptyset$
forall ($e \in E$) do $Visit(e) = false$
$W = \{(entry, n) : (entry, n) \in E\}$
while ($W \neq \emptyset$) do
    $(n, m) = select(W)$
    $W = W \setminus \{(n, m)\}$
    $\mathcal{A}_\bullet(n) = f_n(\mathcal{A}_\circ(n))$
    $Visit(n, m) = true$
    $\mathcal{A}_\circ(m) = \bigcup_{m' \in pred(m)} g(m', \mathcal{A}_\bullet(m'))$
    forall ($p \in succ(m)$ such that $\neg Visit(m, p)$) do
        $W = W \cup \{(m, p)\}$
```

Algorithm 1: ForwardSinglepassDFAnalysis

first case in g returns a singleton set. In the second case in g, no RD of x at the exit of m is known during the dataflow analysis, and thus it considers an AD m_u of x as the RD at the entry of n. The AD m_u will later be resolved to a CD of x satisfying conditions (1) and (2) in Definition 2, which we call *concretization* of m_u.

The transfer function f_n in Eq. 1 computes $\mathcal{A}_\bullet(n)$ from $\mathcal{A}_\circ(n)$ variable-wise. Thus, we have $f_n(\mathcal{A}_\circ(n)) = \bigcup_{x \in \mathbf{Var}} f_{n,x}(\mathcal{A}_\circ(n)(x))$ where $f_{n,x}$ is defined as follows:

$$f_{n,x}(A) = \begin{cases} \{n\} & \text{if } x \in def(n) \vee \mid A \setminus N_u \mid > 1 \\ \{n_u\} & \text{if } x \notin def(n) \wedge A \cap N_u \neq \emptyset \wedge \mid A \setminus N_u \mid \leq 1 \\ A & otherwise \end{cases} \quad (3)$$

$f_{n,x}$ produces $\{n\}$ if x is defined at n or n requires a pseudo definition (i.e. ϕ-function) since there exists more than one CD of x at n. In the second case, x is not defined at n, A contains some abstract RDs and at most one CD. Since the ADs in A are not concretized, the number of CDs of x at n are inconclusive. Hence, we do not conclude if n requires a pseudo definition for x, and $f_{n,x}$ produces the set $\{n_u\}$ of AD for x at n. The third case is applicable when A does not include any AD for x and contains at most one element, and $f_{n,x}(A)$ transfers A from the entry to the exit of n.

Lemma 1. *The set $f_{n,x}(A)$ (i.e. $\mathcal{A}_\bullet(n)(x)$) is at most a singleton and $\mid \mathcal{A}_\circ(n)(x) \mid \leq |pred(n)|$ for any $n \in N$ and $x \in \mathbf{Var}$.*

Proof. $f_{n,x}(A)$ is a singleton set due to the first two cases in Eq. 3. The last case is applicable when $\mid A \setminus N_u \mid \leq 1$ and $A \cap N_u = \emptyset$ implying that $\mid A \mid \leq 1$ and thus produces at most a singleton set. In Eq. 1, since $\mathcal{A}_\bullet(m)(x)$ is at most a singleton set for all $x \in \mathbf{Var}$, $g(m, A)$ produces at most a singleton set for each variable x and each predecessor m of n. Thus, $\mid \mathcal{A}_\circ(n)(x) \mid \leq |pred(n)|$. $\square$

Instead of performing a fixpoint computation, we perform the forward dataflow analysis in Algorithm 1 visiting each edge exactly once and generating RD sets $\mathcal{A}_\circ(n)(x)$ and $\mathcal{A}_\bullet(n)(x)$ for

```
typedef const char *T;
T Perl_ninstr(T big, T bigend, T
    little, T lend)
{
   const char *s, *x, fs = *little;
   bigend -= lend - little++;
   OUTER:
   while (big <= bigend)
   {
      if (*big++ == fs)
      {
         x=big; s=little;
         for (; s < lend; x++,s++)
         {
            if (*s != *x)
               goto OUTER;
         }
         return (char*)(big-1);
      }
   }
   return NULL;

}
```

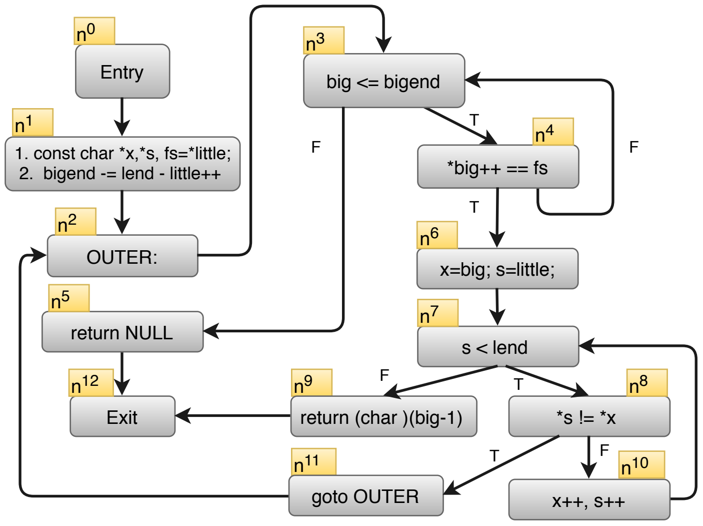


Figure 4: Code from the 500.perlbench_r benchmark in SPEC CPU2017 [7] (left) and its CFG (right)

each $x \in \mathbf{Var}$. The `select` function picks an element arbitrarily from the worklist W. We avoid chaotic fixpoint computation as it will be computationally expensive and will not resolve all ADs into CDs.

Example 1. *Consider the source code and its CFG in Figure 4. The sets of RDs generated by Algorithm 1 for the variable x is shown in Table 1. We assume, like DF-based ϕ-placement algorithms, that all variables are defined at the entry node n^0. However, unlike DF-based methods, our approach has the flexibility to consider any arbitrary set of variable definitions at the entry node. Note that the abstract RDs n^2_u, n^3_u, and n^7_u originate from the join nodes n^2, n^3, and n^7.*

In the next section, we provide a method to concretize all abstract RDs and detect nodes that require ϕ-functions which are also called ϕ nodes.

4.2. Concretization of Abstract RDs

The dataflow analysis in the previous section generates the sets $\mathcal{A}_\circ(n)(x)$ of RDs for all nodes $n \in N$ and program variables $x \in \mathbf{Var}$. $\mathcal{A}_\circ(n)(x)$ may contain ADs and/or CDs of x. If all RDs in $\mathcal{A}_\circ(n)(x)$ are CDs and $\mathcal{A}_\circ(n)(x)$ contains multiple concrete RDs (i.e. $|\mathcal{A}_\circ(n)(x)| > 1$), then we can conclude that n requires a ϕ function for x. If $|\mathcal{A}_\circ(n)(x)| \leq 1$ and all RDs in $\mathcal{A}_\circ(n)(x)$ are CDs, then we can also conclude that n does not require a ϕ function. However, if $\mathcal{A}_\circ(n)(x)$ contains ADs and since some ADs can be concretized to CDs already present in the $\mathcal{A}_\circ(n)(x)$ set, we may not be able to conclude if $\mathcal{A}_\circ(n)(x)$ will contain more than one CD of x, and hence we cannot conclude if

n	$\mathcal{A}_\circ(n)(x)$	$\mathcal{A}_\bullet(n)(x)$	Case of Eq. 3
n^0	$\emptyset$	$\{n^0\}$	Case (1)
n^1	$\{n^0\}$	$\{n^0\}$	Case (3)
n^2	$\{n^0, n_u^7\}$	$\{n_u^2\}$	Case (2)
n^3	$\{n_u^2, n_u^3\}$	$\{n_u^3\}$	Case (2)
n^4, n^5	$\{n_u^3\}$	$\{n_u^3\}$	Case (3)
n^6	$\{n_u^3\}$	$\{n^6\}$	Case (1)
n^7	$\{n^6, n^{10}\}$	$\{n_u^7\}$	Case (2)
n^8, n^9, n^{11}	$\{n_u^7\}$	$\{n_u^7\}$	Case (3)
n^{10}	$\{n_u^7\}$	$\{n^{10}\}$	Case (1)
n^{12}	$\{n_u^7, n_u^3\}$	$\{n_u^{12}\}$	Case (2)

Table 1: The RD sets generated by Algorithm 1 for the code in Figure 4. The last column lists the case number of Eq. 3 that is applied in computing $\mathcal{A}_\bullet(n)(x)$

n requires a ϕ-function or not for variable x. Thus, in order to decide if n requires a ϕ-function or not based on the number of CDs in $\mathcal{A}_\circ(n)(x)$, we need to concretize the abstract RDs.

Our approach to concretize the ADs is to create dependency graphs among abstract and concrete RDs and perform systematic flow analyses of CDs in the dependency graphs to discover the exact relationships among ADs and CDs and then resolve the ADs. To illustrate the procedure, consider the abstract definition n_u of x originated from the CFG node n, i.e. $\mathcal{A}_\bullet(n)(x) = \{n_u\}$ and we would like to concretize n_u. If $\mathcal{A}_\circ(n)(x) = \{n_\lambda^1, n_\lambda^2\}$, then resolving n_u depends on the concrete value of n_λ^1 and n_λ^2. If n_λ^i is an abstract definition, we consider the contents of $\mathcal{A}_\circ(n^i)(x)$ for $i = 1, 2$. Suppose $m^1 \in \mathcal{A}_\circ(n^1)(x)$ and $m^2 \in \mathcal{A}_\circ(n^2)(x)$ are the only RDs of x in these sets which are concrete definitions. We then build the dependency graph containing the nodes $n_u, n_\lambda^1, n_\lambda^2, m^1, m^2$ and the edges $(m^1, n_\lambda^1), (m^2, n_\lambda^2), (n_\lambda^1, n_u), (n_\lambda^2, n_u)$. Systematic flow analysis can reveal that $n_\lambda^1 = m^1$, $n_\lambda^2 = m^2$, and the $\mathcal{A}_\bullet(n)(x)$ set thus contains two CDs m^1 and m^2. Node n will then require a ϕ-function for x and we can conclude that $n_u = n$. In the following sections, we describe the general procedure to build the dependency graphs and systematic flow analyses to concretize the ADs.

4.2.1. Generating the Dependency Graph

Consider the mapping function $\mu(n, x) = \mathcal{A}_\circ(n)(x)$ for any *join* node $n \in N$ and variable $x \in \textbf{Var}$. We consider the mapping functions of join nodes because only join nodes can receive multiple RDs. Thus, $\mu(n, x)$ contains RDs of x which are abstract and/or concrete definitions. If all RDs in $\mu(n, x)$ are concrete definitions, then $\mu(n, x)$ is considered to be resolved. On the other hand, if $\mu(n, x)$ contains some ADs, then we need to consider the mapping functions of those CFG nodes from which the ADs in $\mu(n, x)$ have originated. Let $n = n^1$ and let $n_u^2 \in \mu(n^1, x)$. Thus, in order to resolve $\mu(n, x)$, we need to concretize n_u^2 which requires looking into the RDs in $\mu(n^2, x) = \mathcal{A}_\circ(n^2)(x)$. This, in turn, may require successive consideration of the RDs in the sets $\mu(n^3, x), \ldots \mu(n^k, x)$. Thus, we need to consider the following system of μ-equations to resolve $\mu(n, x)$:

$$\begin{array}{rcl} \mu(n^1, x) & = & S_1 \\ \mu(n^2, x) & = & S_2 \\ & \vdots & \\ \mu(n^k, x) & = & S_k \end{array} \tag{4}$$

where $n^1 = n$, $n^i \in N$ and $S_i \subseteq N \cup N_u$ for all $1 \leq i \leq k$. Moreover, n_u^j appears in S_t for at least all $1 \leq t < j \leq k$. So, each $\mu(n^j, x) = S_j$ equation is included in this system of equations due to the reference n_u^j in some S_t. Intuitively, any $n_u^i \in S_j$ and $n_u^l \in S_i$ imply that node n_j receives an AD of x from n^i, which in turn receives an AD of x from n^l and so on. Thus, Eqs. 4 is a slice of the results generated from Algorithm 1. We need to concretize the ADs in $\mu(n^i, x) \cap N_u$ for all $1 \leq i \leq k$ to decide if the CFG node n requires a ϕ-function for x. It is a challenging problem since Eqs. 4 may contain cyclic references: resolving $\mu(n^{i_1}, x)$ may require resolving $\mu(n^{i_2}, x)$ due to the reference $n_u^{i_2} \in \mu(n^{i_1}, x)$ which in turn may require successive resolution of the sets $\mu(n^{i_3}, x), \ldots, \mu(n^{i_k}, x)$ and finally resolving $\mu(n^{i_k}, x)$ requires resolving $\mu(n^{i_1}, x)$ for some $1 \leq i_1, \ldots, i_k \leq k$.

In the following sections, we provide an efficient and generic method to solve this system of equations originated from reducible or irreducible CFGs. We represent the system of μ-equations by the graph G_μ^α and perform a flow analysis of concrete RDs on this graph. α is a node in the graph G_μ^α which depends on the contents of $\mathcal{A}_\bullet(n)(x)$. If $\mathcal{A}_\bullet(n)(x)$ contains an AD such that $\mathcal{A}_\bullet(n)(x) \cap N_u \neq \emptyset$, then $\alpha \in \mathcal{A}_\bullet(n)(x) \cap N_u$. Thus, the concretization of α depends on the solution of Eqs. 4. However, if $\mathcal{A}_\bullet(n)(x) \cap N_u = \emptyset$, we simply set $\alpha = n$. We define the graph G_μ^α representing Eqs. 4 as follows:

Definition 3 (Graph G_μ^α). *The system of $\mu(n, x)$ equations (i.e., Eqs 4) form a directed graph $G_\mu^\alpha = (N_\mu, E_\mu)$ such that*

- $N_\mu = S_1 \cup \ldots \cup S_k \cup \{\alpha\}$, *and*
- $E_\mu = E_\mu^K$

where

- $E_\mu^0 = \{(m_\lambda, \alpha) : m_\lambda \in \mu(n, x)\}$,
- $E_\mu^{i+1} = \{(m_\lambda, m'_u) : (m'_u, m''_\lambda) \in E_\mu^i, m_\lambda \in \mu(m', x)\}$ *for any* $i > 0$, *and*
- *there exists* $K > 0$ *such that* $E_\mu^K = E_\mu^{K-1}$.

The above definition inductively defines the edges of the graph G_μ^α: (m_λ, α) is an edge in G_μ^α for any $m_\lambda \in \mu(n, x)$, and if (m'_u, m''_λ) is already an edge in the graph and there exists $m_\lambda \in \mu(m', x)$, then (m_λ, m'_u) is also an edge in G_μ^α. This process progressively generates edges in G_μ^α until no more edges can be added to the graph. Intuitively, if $m_u \in N_\mu$ and there exists a RD $m'_\lambda \in \mu(m, x)$, then $(m'_\lambda, m_u) \in E_\mu$ and any concretization of m_u depends on m'_λ. Note that if $m \in N_\mu$ is a concrete RD, then (m'_λ, m) is not an edge in G_μ^α for any $m'_\lambda \in \mu(m, x)$ since m is already concretized. Thus, the graph G_μ^α contains all the dependencies among abstract and concrete RDs.

4.2.2. Flow Analysis of Concrete Definitions

Let $C \subseteq N_\mu$ be the set of nodes in $G_\mu^\alpha = (N_\mu, E_\mu)$ representing CDs of a variable $x \in \mathbf{Var}$. The initial set of CDs includes all nodes $n_\lambda \in N_\mu$ such that $n_\lambda = n$ and x is defined at n. Since all nodes in $N_\mu \cap N$ include definitions of variable x, $C = N_\mu \cap N$ initially. If some $m_\lambda \in N_\mu$ is resolved to $m_\lambda = m$, then we consider m_λ as a CD of x. Let $\Sigma : N_\mu \to \mathcal{P}(\hat{C})$ be the function recording the flow of concrete RDs in $\hat{C} = \{n : n_\lambda \in C\}$ to each node in G_μ^α. We define Σ as follows:

Definition 4 (Σ). *Let $m_\lambda \in N_\mu$, let $x \in \mathbf{Var}$, and let C be the set of CDs of x. $\Sigma(m_\lambda)$ contains all CDs n such that there exists a path $\pi : n_\lambda \xrightarrow{+} m_\lambda$ from $n_\lambda \in N_\mu \cap C$ in G_μ^α where no node in $\pi - \{n_\lambda, m_\lambda\}$ is a CD of x.*

Thus, $\Sigma(m_\lambda)$ collects all concrete RDs of x at m_λ. Algorithm 2 performs the flow analysis of CDs according to Definition 4 and returns $\Sigma(m_\lambda)$ for all nodes m_λ in the graph G^α_μ. It is a worklist-based algorithm that records the flow of CD n to $\Sigma(m_\lambda)$ for the nodes $m_\lambda \in N_\mu$ and $n_\lambda \in C$. Note that Algorithm 2 receives Σ and C as input. We shall perform iterative flow analysis to update Σ. Algorithm 2 is applied iteratively in Algorithm 4 (see Section 4.2.4) to resolve all ADs to CDs. If there remain some ADs to be resolved to CDs after an application of Algorithm 2 that updated Σ to Σ', a new set of CDs C' is obtained from C and Σ' by applying Algorithm 3 (see Section 4.2.3), and Algorithm 2 is applied again for the inputs Σ', C' and the dependency graph G^α_μ to resolve the remaining ADs.

```
   Input : $C \subseteq N_\mu, G^\alpha_\mu = (N_\mu, E_\mu), \Sigma, \mu, x \in \mathbf{Var}$
1  $W = \emptyset$
2  forall $n_\lambda \in N_\mu$ do
3      $V(n_\lambda) = false$
4      forall $m_\lambda \in C$ do $\mathcal{P}(n_\lambda, m) = false$
5  forall $n_\lambda \in C \wedge m_\lambda \in succ_\mu(n_\lambda)$ do
6      $W = W \cup \{(m_\lambda, n)\}$
7  while $(W \neq \emptyset)$ do
8      $(m_\lambda, n) = select(W)$
9      $W = W \setminus \{(m_\lambda, n)\}$
10     if $(\neg V(m_\lambda))$ then $\Sigma(m_\lambda) = \emptyset$
11     $\Sigma(m_\lambda) = \Sigma(m_\lambda) \cup \{n\}$
12     $V(m_\lambda) = true$
13     $\mathcal{P}(m_\lambda, n) = true$
14     if $(\mu(m, x) \cap N_u \neq \emptyset \vee |\mu(m, x)| > 1)$ then
15         forall $(p_\lambda \in succ(m_\lambda) \wedge \neg\mathcal{P}(p_\lambda, n))$ do
16             $W = W \cup \{(p_\lambda, n)\}$
17 return $\Sigma$
```

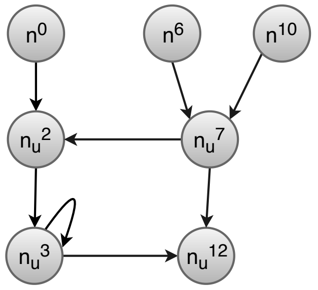
Algorithm 2: Flow$_C$

Two Boolean functions V and $\mathcal{P}$ are maintained in the algorithm so that $V(m_\lambda)$ and $\mathcal{P}(m_\lambda, n)$ decide if m_λ is visited and the CD n reaches m_λ before. The worklist W is initialized by the pair (m_λ, n) to transfer the CD n to $\Sigma(m_\lambda)$ where $n_\lambda \in C$ and m_λ is a successor of n_λ in G^α_μ. Note that (m_λ, n) is not necessarily an edge in the graph G^α_μ, rather n is always a CD to be transferred to $\Sigma(m_\lambda)$. The two `if` instructions at lines 10 and 14 affect the construction of Σ in the following ways:

- In the first visit to m_λ (i.e. $V(m_\lambda) = false$), we set $\Sigma(m_\lambda) = \emptyset$ which has a profound implication. Previous flow analysis may generate incorrect $\Sigma(m_\lambda)$ for some $m_\lambda \in N_\mu$. This may happen when $|\Sigma(m'_\lambda)| > 1$ for any parent node $m'_\lambda \in N_\mu$ of m_λ such that the CFG node m' may require a ϕ-function and thus $m'_\lambda = m'$, but the flow analysis generates $\Sigma(m_\lambda)$ that does not contain m' and may contain the RDs of $\Sigma(m'_\lambda)$ instead. This is because $m'_\lambda = m'$ is decided after the flow analysis. Resetting $\Sigma(m_\lambda) = \emptyset$ followed by including n to $\Sigma(m_\lambda)$ (line 11) remove all incorrect CDs of x from $\Sigma(m_\lambda)$ and $\Sigma(m_\lambda)$ will contain some but not all correct CDs of x reaching m_λ. Later, we reconstruct $\Sigma(m_\lambda)$ from all predecessors m'_λ of m_λ.

$$\begin{aligned}\mu(n^{12},x) &= \{n_u^3, n_u^7\}\\ \mu(n^3,x) &= \{n_u^2, n_u^3\}\\ \mu(n^2,x) &= \{n^0, n_u^7\}\\ \mu(n^7,x) &= \{n^6, n^{10}\}\end{aligned}$$

(a)



(b)

n_λ	$\Sigma(n_\lambda)$
n^0	$\{n^0\}$
n^6	$\{n^6\}$
n^{10}	$\{n^{10}\}$
n_u^2	$\{n^0, n^6, n^{10}\}$
n_u^3	$\{n^0, n^6, n^{10}\}$
n_u^7	$\{n^6, n^{10}\}$
n_u^{12}	$\{n^0, n^6, n^{10}\}$

(c)

Figure 5: (a) The system of μ equations obtained from the RDs in Table 1 to concretize n_u^{12}, (b) the graph $G_\mu^{n^{12}}$ generated according to Definition 3 from the equations in (a), and (c) the Σ values generated by the flow analysis in Algorithm 2 where $C = \{n^0, n^6, n^{10}\}$

- The second `if` instruction at line 14 ensures that m_λ is not resolved to a CD of x. As we shall see in the next section, if m requires a ϕ-function for x (and hence $m_\lambda = m$ is a CD), we set $\mu(m, x) = \{m\}$, and then we get $\mu(m, x) \cap N_u = \emptyset$ and $|\mu(m, x)| = 1$. However, if m is not a concrete RD of x, then the CD n can reach all the successor nodes p_λ of m_λ provided that n has not reached p_λ before.

Example 2. *Consider the RD sets $\mathcal{A}_\circ(n)(x)$ in Table 1 generated by Algorithm 1 for the code in Figure 4. Suppose we would like to concretize n_u^{12}. Figure 5(a) presents the system of μ equations obtained from the RD sets. The dependency graph $G_\mu^{n^{12}}$ in Figure 5(b) is generated according to Definition 3 from this system of μ equations. The table in Figure 5(c) shows the Σ after the flow analysis of concrete definition in $C = \{n^0, n^6, n^{10}\}$. The first three lines are due to initialization by Algorithm 4, and the remaining data are generated by the flow analysis in Algorithm 2.*

4.2.3. Necessary and Sufficient Conditions for Detecting ϕ Nodes

Any node $m_\lambda \in N_\mu$ such that $|\Sigma(m_\lambda)| > 1$ is a potential candidate that may require a ϕ-function for x. However, the condition $|\Sigma(m_\lambda)| > 1$ is necessary but not sufficient for any $m_\lambda \in N_\mu$ to require a ϕ-function. To see why the condition is not sufficient, consider the nodes n_u^2, n_u^3 and n_u^7 in Figure 5(b), and $\Sigma(n_u^2)$ and $\Sigma(n_u^3)$ in Figure 5(c). It is obvious that $n_u^7 = n^7$ as distinct CDs n^6 and n^{10} reach n_u^7 causing n^7 to require a ϕ function for x. n^7 is thus a CD of x. Also, $n_u^2 = n^2$ since the CDs n^0 and n^7 reach n_u^2. However, note that $\Sigma(n_u^2) = \Sigma(n_u^3)$ and $|\Sigma(n_u^3)| > 1$. Since n_u^2 require a ϕ-function, the pseudo-definition of x at n^2 will make n^2 a new CD of x which will invalidate $\Sigma(n_u^3) = \{n^0, n^6, n^{10}\}$ and a new flow analysis will eventually generate $\Sigma(n_u^3) = \{n^2\}$. In the following, we define $Nodes_\phi(C, \Sigma, G_\mu^\alpha)$ which is the set of all nodes $m_\lambda \in N_\mu$ that definitely requires a ϕ-function for x.

Definition 5 ($Nodes_\phi(C, \Sigma, G_\mu^\alpha)$)**.** *The set $Nodes_\phi(C, \Sigma, G_\mu^\alpha)$ contains all nodes $m_\lambda \in N_\mu$ such that the following conditions hold:*

1. *$|\Sigma(m_\lambda)| > 1$, and*

2. *there exists a path $\pi : n_\lambda \overset{+}{\rightarrow} m_\lambda$ in G_μ^α for any $n_\lambda \in \Sigma(m_\lambda)$ such that $|\Sigma(n'_\lambda)| \leq 1$ and n'_λ is not a CD for all $n'_\lambda \in \pi - \{m_\lambda, n_\lambda\}$.*

```
Input : $C, \Sigma, (N_\mu, E_\mu), \mu, completed, x \in \mathbf{Var}$
1  $Nodes_\phi(C, \Sigma, G^\alpha_\mu) = \emptyset$
2  $W = \bigcup_{m_\lambda \in C} succ_\mu(m_\lambda)$
3  while $(W \neq \emptyset)$ do
4      $n_\lambda = select(W)$
5      $W = W \setminus \{n_\lambda\}$
6      if $(\neg completed(n_\lambda) \wedge |\Sigma(n_\lambda)| > 1)$ then
7          $\mu(n, x) = \{n\}$
8          $Nodes_\phi(C, \Sigma, G^\alpha_\mu) = Nodes_\phi(C, \Sigma, G^\alpha_\mu) \cup \{n_\lambda\}$
9      else if $(\neg completed(n_\lambda) \wedge |\Sigma(n_\lambda)| \leq 1)$ then
10         $completed(n_\lambda) = true$
11         $W = W \cup \{n'_\lambda : n'_\lambda \in succ_\mu(n_\lambda), \neg completed(n'_\lambda)\}$
12 return $Nodes_\phi(C, \Sigma, G^\alpha_\mu)$
```

Algorithm 3: computeNodes$_\phi$

Definition 5 provides the necessary and sufficient conditions for any $m_\lambda \in N_\mu$ to require a ϕ-function for x. Intuitively, we find the first node $m_\lambda \in N_\mu$ along any nontrivial path $\pi : n_\lambda \to m_\lambda$ from any $n_\lambda \in \Sigma(m_\lambda)$ such that $\Sigma(m_\lambda)$ contains multiple distinct CDs, and $\Sigma(n'_\lambda)$ is at most a singleton for all nodes n'_λ in the path π excluding m_λ and n_λ. Moreover, no node in $\pi - \{m_\lambda, n_\lambda\}$ should be a CD of x. Since n is a concrete definition of x for any $n_\lambda \in C$ which reaches m_λ through the path π and no other CD can reach m_λ through π except n due to $|\Sigma(n'_\lambda)| \leq 1$ for all $n'_\lambda \in \pi - \{m_\lambda, n_\lambda\}$, there must be another CD of x in $\Sigma(m_\lambda)$ along with n due to $|\Sigma(m_\lambda)| > 1$, and hence m_λ requires a ϕ-function for x.

Algorithm 3 computes the set of nodes $Nodes_\phi(C, \Sigma, G^\alpha_\mu)$ requiring ϕ functions for x. It visits each path π that starts from any successor of any node in C. The sequence of nodes $n^1_\lambda, \ldots, n^k_\lambda$ in π are visited with the following conditions:

- Each node n^i_λ is visited if $completed(n^i_\lambda)$ is false for all $1 \leq i \leq k$. $completed(n^i_\lambda) = false$ if the concretization process could not decide if n^i_λ requires a ϕ-function or not.
- For all n^i_λ such that $1 \leq i < k$, $|\Sigma(n^i_\lambda)| \leq 1$. As we prove Lemma 4 in Section 5.1, $|\Sigma(n^i_\lambda)| \leq 1$ implies that n^i_λ will never require a ϕ function for x. Thus, we set $completed(n^i_\lambda) = true$ to indicate that we already have a decision about n^i_λ and thus it does not require any further processing.
- We have two possible scenarios for n^k_λ: either (i) $completed(n^{k+1}_\lambda) = false$ and $|\Sigma(n^k_\lambda)| > 1$, or (ii) $completed(n^{k+1}_\lambda) = true$. In the former case, n^k_λ is the first reachable node in π such that $\Sigma(n^k_\lambda)$ contains multiple CDs and thus the CFG node n^k requires a ϕ function for x according to Definition 5. We include n^k_λ in $Nodes_\phi(C, \Sigma, G^\alpha_\mu)$ and set $\mu(n^k, x) = \{n^k\}$ to indicate the concretization $n^k_\lambda = n^k$. In the latter case, n^k_λ is already resolved and thus the successors of n^k_λ are not visited.

Example 3. *Consider the graph $G^{n^{12}}_\mu$ and Σ in Figure 5. We obtain $Nodes_\phi(C, \Sigma, G^\alpha_\mu) = \{n^2_u, n^7_u\}$, hence there exist paths $\pi_1 : n^0 \to n^2_u$ and $\pi_2 : n^6 \to n^7_u$ satisfying the conditions in Definition 5. Thus, n^2_u and n^7_u are resolved to CDs n^2 and n^7. Note that $n^3_u \notin Nodes_\phi(C, \Sigma, G^\alpha_\mu)$ even though*

$|\Sigma(n_u^3)| > 1$. *We do not conclude that n^3 requires a ϕ-function for x as there exists $n_u^2 \in n^0 \to n_u^3$ such that $|\Sigma(n_u^2)| > 1$. In fact, the newly resolved CD n^2 of x will be the only element in $\Sigma(n_u^3)$ in the next flow analysis. Thus, n_u^3 is resolved to n^2, and we conclude that n^3 does not require a ϕ-function for x after the next flow analysis.*

4.2.4. Iterative Flow Analysis

Let $C_0 = N_\mu \cap N$ be the set of all initial concrete definitions of x in G_μ^α. We apply Algorithm 2 to systematically perform the *flow analysis* of the concrete definitions in C_0 to construct Σ and then apply Algorithm 3 to compute the set $Nodes_\phi(C_0, \Sigma, G_\mu^\alpha)$ of ϕ nodes. Since each node $m_\lambda \in Nodes_\phi(C_0, \Sigma, G_\mu^\alpha)$ will contain pseudo-definition of x, m is considered to be a new concrete definition of x. So, we obtain the set $C_1 = C_0 \cup Nodes_\phi(C_0, \Sigma, G_\mu^\alpha)$ of nodes in G_μ^α that contain definitions or require pseudo-definitions of x after the first flow analysis of C_0 in G_μ^α. Next, for each $m_\lambda \in Nodes_\phi(C_0, \Sigma, G_\mu^\alpha)$, we need to perform the flow analysis of m to the nodes in G_μ^α. The flow analysis will produce $C_2 \supseteq C_1$ containing more nodes requiring ϕ-functions. Thus, we iteratively perform the flow analysis and compute the set of nodes $Nodes_\phi(C_i, \Sigma, G_\mu^\alpha)$ for $i \geq 0$ requiring ϕ-functions for x until no new CDs can be generated. In particular, the following equations describe a series of computations to detect the set of ϕ nodes C_{i+1} for $i \geq 0$:

$$\begin{aligned} \Sigma_{i+1} &= Flow_C(C_i, G_\mu^\alpha, \Sigma_i) \\ C_{i+1} &= C_i \cup Nodes_\phi(C_i, \Sigma_{i+1}, G_\mu^\alpha) \end{aligned} \tag{5}$$

where $C_0 = N_\mu \cap N$, $\Sigma_0(n_\lambda) = \emptyset$ for all $n_\lambda \in N_\mu \setminus C_0$, $\Sigma_0(n_\lambda) = \{n\}$ for all $n_\lambda \in C_0$, the $Flow_C(C_i, G_\mu^\alpha, \Sigma_i)$ set is computed by the flow analysis in Algorithm 2, and $Nodes_\phi(C_i, \Sigma_{i+1}, G_\mu^\alpha)$ set is computed by the ϕ node detection method in Algorithm 3. The iterative process terminates when $C^{L+1} = C^L$ for any $L > 0$. If $|C_0| \leq 1$, no node in G_μ^α and hence no CFG node n^i in G will require a pseudo-definition of x as each node m_λ in G_μ^α cannot receive more than one CD. So, we need a sequence of flow analysis on G_μ^α, and since G_μ^α is much smaller than the CFG, the flow analysis is very efficient.

As we mentioned in Section 4.2.2, the flow analysis may produce Σ_{i+1} which may contain incorrect CDs. Nevertheless, the conditions in Definition 5 are necessary and sufficient to produce the correct set of nodes $Nodes_\phi(C_i, \Sigma_{i+1}, G_\mu^\alpha)$ that will require ϕ-functions. The flow analysis in the next iteration will generate Σ_{i+2} set which will remove some incorrect flow of CDs present in the Σ_{i+1} set. For example, $\Sigma(n_u^2)$ includes both n^6 and n^{10} in Figure 5. However, since $\Sigma(n_u^7)$ satisfies the conditions in Definition 5, n^7 is a concrete definition of x that should reach n_u^2 instead of n^6 and n^{10}. Nevertheless, both conditions in Definition 5 are satisfied by n_u^2 to conclude that n^2 requires a ϕ-function for x (i.e. $n_u^2 = n^2$). Such conclusion is due to the following reasons: n^0 is definitely a concrete RD of x at n_u^2 due to condition (2) in Definition 5, and even though n^6 and n^{10} are not valid CDs of x at n_u^2, there must be at least one CD due to n^6 and/or n^{10} (in this case n^7). Further flow analysis with respect to $\{n_u^2, n_u^7\}$ will generate Σ which will not contain this incorrect flow of CDs.

The overall procedure to detect all ϕ nodes of any given CFG (N, E) for the variables in **Var** is provided in Algorithm 4, where Algorithms 2 and 3 are applied iteratively. For each variable $x \in$ **Var**, it computes the set of ϕ nodes $C_R(x)$. The *createGraph* procedure generates the dependency graph (N_μ, E_μ) according to Definition 3. The instructions at line 10 of the $Flow_C$ procedure in Algorithm 2 reset $\Sigma(n_\lambda)$ for some node n_λ in the dependency graph which not only removes some incorrect CDs but also some correct CDs as well from the $\Sigma(n_\lambda)$ set. We reconstruct $\Sigma(n_\lambda)$ by applying the *reConstruct*Σ procedure at line 20 of Algorithm 4 and assign the reconstructed

```
    Input : $(N, E), \mathbf{Var}$
1   Apply Alg. 1 to compute $\mathcal{A}_\circ$ and $\mathcal{A}_\bullet$ sets
2   forall $(n \in N \wedge x \in \mathbf{Var})$ do  $\mu(n, x) = \mathcal{A}_\circ(n)(x)$
3   forall $x \in \mathbf{Var}$ do
4       $C_R(x) = \emptyset$
5       forall (join nodes $n \in N$) do  $resolved(n) = false$
6       forall (join nodes $n \in N$ such that $resolved(n) = false$) do
7           $(N_\mu, E_\mu) = createGraph(n, \mu, \mathcal{A}_\bullet)$
8           $C_0 = \{n_\lambda : n_\lambda \in N_\mu, x \in def(n)\}$
9           forall $(n_\lambda \in N_\mu)$ do
10              $\Sigma(n_\lambda) = \emptyset$
11              $completed(n_\lambda) = false$
12          forall $(m_\lambda \in C_0)$ do $\Sigma(m_\lambda) = \{m_\lambda\}$
13          $C = C_0$
14          while $(C \neq \emptyset)$ do
15              $\Sigma = Flow_C(C, (N_\mu, E_\mu), \Sigma, \mu, x)$
16              forall $(m_\lambda \in C)$ do  $completed(m_\lambda) = true$
17              $C = computeNodes_\phi(C, \Sigma, (N_{\mu, E_\mu}), \mu, completed, x)$
18              $C_R(x) = C_R(x) \cup C$
19          forall $(n_\lambda \in N_\mu)$ do
20              $\mathcal{A}_\circ(n)(x) = reConstruct\Sigma(n_\lambda, \Sigma, (N_\mu, E_\mu), x, \mu)$
21              $resolved(n) = true$
22  return $(C_R)$
```

Algorithm 4: IterativeFlow

```
    Input : $n_\lambda, \Sigma, (N_\mu, E_\mu), x \in \mathbf{Var}, \mu$
1   $\Sigma(n_\lambda) = \emptyset$
2   forall $(n'_\lambda \in pred_\mu(n_\lambda))$ do
3       if $(\mu(n', x) = \{n'\} \vee (n'_\lambda = n_\lambda \wedge x \in def(n)))$ then
4           $\Sigma(n_\lambda) = \Sigma(n_\lambda) \cup \{n'\}$
5       else
6           $\Sigma(n_\lambda) = \Sigma(n_\lambda) \cup \Sigma(n'_\lambda)$
7   return $(\Sigma(n_\lambda))$
```

Algorithm 5: reConstructΣ

value to $\mathcal{A}_\circ(n)(x)$ such that it contains all the concrete RDs of x at n. Algorithm 5 illustrates the *reConstruct*Σ procedure. After resetting $\Sigma(n_\lambda)$ to an empty set, it considers each predecessor n'_λ of n_λ and update $\Sigma(n_\lambda)$ in one of the following ways:

- The condition $\mu(n', x) = \{n'\}$ holds if n' requires a ϕ function for x (see line 7 in Algorithm 3). n' is thus a CD of x and n' is included in $\Sigma(n_\lambda)$.
- The condition $n_\lambda = n'_\lambda$ corresponds to a self loop from n_λ to itself in the dependency graph (N_μ, E_μ). This happens when there is a cycle in the CFG through the join node n and no node in any path from n to itself defines n. However, if the CFG node n defines the variable x then we consider n as a CD of x and include n in $\Sigma(n_\lambda)$. Otherwise, we ignore any RD coming through this edge as it is the same RD that goes out from n_λ and it should not contribute deciding if n requires a ϕ function or not for x.
- In any other case, n'_λ is not a CD and thus $|\Sigma(n'_\lambda)| \leq 1$ and we include the element of $\Sigma(n'_\lambda)$ in $\Sigma(n_\lambda)$.

Our experiments that we discuss in Section 6 show that this approach is fairly efficient for most benchmark codes since G^α_μ is usually very small in size and the source code contains sparse variable definitions. Our algorithm may take longer time if the code contains dense variable definitions.

5. Correctness and Computational Complexity

5.1. Correctness of Computing ϕ Nodes

In the remainder of this section, we assume the following:

- (N, E) is the CFG of any program,
- N_u is the set of pseudo nodes related to N
- $G^\alpha_\mu = (N_\mu, E_\mu)$ be any dependency graph generated according to Definition 3,
- $C \subseteq N_\mu$ be the set of CDs for the variable $x \in \mathbf{Var}$,
- Σ records the flow of CDs in C in Algorithm 2, and
- $Nodes_\phi(C, \Sigma, G^\alpha_\mu)$ is the set of ϕ nodes computed in Algorithm 3.

In this section, we provide Theorem 1 to state the correctness of Algorithm 4. We prove this theorem with the aid of some auxiliary lemmas which also give us more insight about the algorithms provided in Section 4.

Lemma 2. *Algorithm 2 computes Σ according to Definition 4.*

Proof. During the initialization phase in Algorithm 2, two boolean functions $V(m_\lambda)$ and $P(m_\lambda, n)$ are set to false for all nodes $m_\lambda \in N_\mu$ and all CDs $n_\lambda \in C$. $V(m_\lambda)$ and $P(m_\lambda, n)$ indicate that the iterative flow analysis in the algorithm has not visited m_λ and the CD n is not included to $\Sigma(m_\lambda)$, respectively. During the iterative flow analysis, $\Sigma(m_\lambda)$ is reset to an empty set to discard previously stored values in $\Sigma(m_\lambda)$ when $V(m_\lambda)$ is false. The worklist W is initialized by the pair (m_λ, n) to transfer the CD n to $\Sigma(m_\lambda)$ where $n_\lambda \in C$ and m_λ is a successor of n_λ in G^α_μ.

At each iteration in the `while` loop, Algorithm 2 traverses the path π in Definition 4 by selecting (and removing) an element (m_λ, n) from the worklist W and including the CD n to $\Sigma(m_\lambda)$. Next, if m_λ is not resolved to a CD (i.e. the conditions at line 14 in the algorithm are satisfied) satisfying the condition on path π in Definition 4, and n is not transferred to $\Sigma(p_\lambda)$ for the successor p_λ of m_λ (i.e. $P(p_\lambda, n) = false$), (p_λ, n) is included in W to transfer n to $\Sigma(p_\lambda)$. Thus, all CDs n is transferred to $\Sigma(m_\lambda)$ for each visited node m_λ in π in Algorithm 2 until there are no successors to m_λ or m_λ is resolved to a CD. □

Lemma 3. *Algorithm 3 computes $Nodes_\phi(C, \Sigma, G^\alpha_\mu)$ according to Definition 5.*

Proof. Algorithm 3 receives the boolean function *completed*(n_λ) as input such that *completed*$(n_\lambda) =$ *true* if $n_\lambda \in C$, and *completed*$(n_\lambda) = false$ otherwise. It starts visiting the path π in Definition 5 from the successors of the nodes in C by keeping them in the worklist W and selecting each of them to process at a time. At each iteration in the `while` loop, if *completed*$(n_\lambda) = false$ implying that n_λ is not yet visited, and $|\Sigma(m_\lambda)| \leq 1$ implying that condition 2 in Definition 5 is satisfied, the successors n'_λ of n_λ are included in W to visit in the successive iterations. However, if *completed*$(n_\lambda) = false$ (and thus n_λ is not yet visited) and $|\Sigma(m_\lambda)| > 1$ implying that condition 1 in Definition 5 is satisfied, n_λ is included in $Nodes_\phi(C, \Sigma, G^\alpha_\mu)$. Note that if n_λ is included in $Nodes_\phi(C, \Sigma, G^\alpha_\mu)$, path π is not extended beyond n_λ by not including successors of n_λ in W. Thus, Algorithm 3 computes $Nodes_\phi(C, \Sigma, G^\alpha_\mu)$ according to Definition 5. □

Lemma 4. *Let $m_\lambda \in N_\mu \setminus C$ be any node in N_μ. If $|\Sigma(m_\lambda)| \leq 1$, then $\Sigma(m_\lambda)$ will be unchanged by a further application of Algorithm 2 with a different C set, and m_λ cannot be a ϕ node for x.*

Proof. We first prove that $\Sigma(m_\lambda) \neq \emptyset$. The edges in the dependency graph (N_μ, E_μ) are constructed inductively in Definition 3 such that if $(m'_u, m''_\lambda) \in E_\mu$ and there exists $m_\lambda \in \mu(m', x)$ then $(m_\lambda, m'_u) \in E_\mu$. Thus, (m_λ, m'_u) is an edge in (N_μ, E_μ) since $m'_u \in N_u$. If (m_λ, n_λ) is any edge in E_μ and $m_\lambda = m$ (i.e. $m_\lambda \in N$), then m is a CD of x (i.e. $x \in def(x)$) and there is no incoming edge to m_λ in (N_μ, E_μ). Thus, (N_μ, E_μ) is a connected graph. Let $C_0 = \{n : n \in N_\mu, x \in def(x)\}$ be the initial set of CDs of x. Then, any node $n_\lambda \in N_\mu$ is reachable from a node in C_0. So, the first flow analysis in Algorithm 2 with respect to $C = C_0$ will produce Σ such that $\Sigma(m_\lambda) \neq \emptyset$ for any $m_\lambda \in N_\mu$. If m_λ is visited by a further application of Algorithm 2, an element (m_λ, n) must be selected from W and n is included in $\Sigma(m_\lambda)$ during the execution of the `while` loop in Algorithm 2. Thus, $\Sigma(m_\lambda) \neq \emptyset$.

Since $|\Sigma(m_\lambda)| \leq 1$, let $\Sigma(m_\lambda) = \{n\}$ for some $n_\lambda \in N_\mu$. Thus, there exists a path $\pi : n_\lambda \xrightarrow{+} m_\lambda$ in the graph (N_μ, E_μ) and no node $n'_\lambda \in \pi - \{n_\lambda, m_\lambda\}$ is a CD of x according to Definition 4 (i.e. either $|\mu(n', x)| \geq 2$ or $\mu(n', x) \cap N_u \neq \emptyset$ at line 14 in Algorithm 2). Moreover, $|\Sigma(m_\lambda)| \leq 1$ implies $|\Sigma(n'_\lambda)| \leq 1$ since otherwise any element in $\Sigma(n'_\lambda)$ other than n would also be included in $\Sigma(m_\lambda)$ due to the path π.

If there exists $m'_\lambda \in N_\mu$ such that $|\Sigma(m'_\lambda)| > 1$, m'_λ is a potential candidate to become a CD of x. However, we argue that if m'_λ can be resolved to a CD of x such that $m'_\lambda = m'$, then m' cannot be included in $\Sigma(m_\lambda)$. Since all elements of $\Sigma(m'_\lambda)$ are not included in $\Sigma(m_\lambda)$, either there exists no path from m'_λ to m_λ or all paths from m'_λ to m_λ go through n_λ which is a CD of x. In any case, $\Sigma(m_\lambda)$ (and also $\Sigma(n'_\lambda)$) cannot receive any other CD of x in further flow analysis except n according to Definition 4. Thus, $\Sigma(m_\lambda)$ will never be changed and the condition $|\Sigma(m_\lambda)| \leq 1$ will hold in further flow analysis having different CDs $C' \subseteq N_\mu$ and m_λ will never be a ϕ node for x. □

Lemma 5. *No two CDs in C can reach any $m_\lambda \in N_\mu \setminus Nodes_\phi(C, \Sigma, G^\alpha_\mu)$, and m_λ cannot become a ϕ node for x due to C.*

Proof. Let $\hat{C} = \{n : n_\lambda \in C\}$. Algorithm 3 computes the set $Nodes_\phi(C, \Sigma, G^\alpha_\mu)$ of ϕ nodes according to Definition 5 (Lemma 3). Now, suppose there exists a node $m_\lambda \in N_\mu$ such that $m_\lambda \notin Nodes_\phi(C, \Sigma, G^\alpha_\mu)$ but $\Sigma(m_\lambda) \cap \hat{C} \neq \emptyset$. For each CD $n \in \Sigma(m_\lambda) \cap \hat{C}$, there exists a path $\pi : n_\lambda \overset{+}{\to} m_\lambda$ in the graph (N_μ, E_μ) through which n is transferred to $\Sigma(m_\lambda)$. Since $m_\lambda \notin Nodes_\phi(C, \Sigma, G^\alpha_\mu)$, either condition (1) or (2) in Definition 5 does not hold. In the former case, $|\Sigma(m_\lambda)| \leq 1$ and m_λ cannot become a ϕ node for x according to Lemma 4. In the latter case, for all $n \in \Sigma(m_\lambda) \cap \hat{C}$, for all paths $\pi : n_\lambda \overset{+}{\to} m_\lambda$ in the graph (N_μ, E_μ), there exists a node $n'_\lambda \in \pi - \{n_\lambda, m_\lambda\}$ such that $|\Sigma(n'_\lambda)| > 1$ or n'_λ is a CD. Assume that n'_λ is the first reachable from n_λ in case there are multiple nodes in π having multiple CDs. Then, either $n'_\lambda \in Nodes_\phi(C, \Sigma, G^\alpha_\mu)$ is a ϕ node for x according to Definition 5 or n'_λ is the CD n'. In any case, the CD n will not reach m_λ through n'_λ according to Definition 4 in the next flow analysis. Since this happens for all paths $\pi : n_\lambda \overset{+}{\to} m_\lambda$ from $n_\lambda \in C$ for all $n \in \Sigma(m_\lambda) \cap \hat{C}$, no CD of x from C will reach $\Sigma(m_\lambda)$. Thus, for any $m_\lambda \in N_\mu \setminus Nodes_\phi(C, \Sigma, G^\alpha_\mu)$, our assumption $\Sigma(m_\lambda) \cap \hat{C} \neq \emptyset$ leads to either $|\Sigma(m_\lambda)| \leq 1$ or the contradiction that a CD $n \in \Sigma(m_\lambda) \cap \hat{C}$ reaches $\Sigma(m_\lambda)$. So, $Nodes_\phi(C, \Sigma, G^\alpha_\mu)$ is the only set of ϕ nodes receiving multiple CDs from C for x. □

Lemma 6. *If Algorithm 3 computes $Nodes_\phi(C, \Sigma, G^\alpha_\mu) = \emptyset$, then either completed$(n_\lambda) = true$ or $|\Sigma(n_\lambda)| \leq 1$ for any $n_\lambda \in N_\mu$.*

Proof. We obtain $Nodes_\phi(C, \Sigma, G^\alpha_\mu) = \emptyset$ if there exists no $m_\lambda \in N_\mu$ visited by Algorithm 3 such that *completed*$(m_\lambda) = false$ and $|\Sigma(m_\lambda)| > 1$. Thus, the lemma holds for all visited nodes $m_\lambda \in N_\mu$ by the algorithm. However, assume that there exists a node $m_\lambda \in N_\mu$ such that *completed*$(m_\lambda) = false$, $|\Sigma(m_\lambda)| > 1$, and it is not visited by the algorithm. This is possible only when there exists a path $\pi : n_\lambda \overset{+}{\to} m_\lambda$ for any $n \in \Sigma(m_\lambda)$ and there exists $n'_\lambda \in \pi - \{n_\lambda, m_\lambda\}$ such that *completed*$(n'_\lambda) = true$. Note that $n_\lambda \in C$, and if *completed*$(n'_\lambda) = true$, then no successors of n'_λ are visited. The iterative flow analysis in Algorithm 4 calls Algorithm 3 iteratively, and *completed*(n'_λ) can be set to true due to a previous call to Algorithm 3 in one of the following ways:

- n'_λ was included in $Nodes_\phi(C, \Sigma, G^\alpha_\mu)$ in the previous call to Algorithm 3 (at line 17 in Algorithm 4) and *completed*(n'_λ) was set to true in the next iteration (line 16 in Algorithm 4).
- path π was visited in a previous call to Algorithm 3, the conditions *completed*$(m_\lambda) = false$ and $|\Sigma(m_\lambda)| \leq 1$ held for node m_λ, and *completed*(n'_λ) was set to true during visiting π.

In the former case, n' is a CD and n cannot be transferred to $\Sigma(m_\lambda)$ according to Definition 4. In Algorithm 3, we set $\mu(n', x) = n'$ and $\Sigma(m_\lambda)$ cannot include n due to line 13 in Algorithm 2. In the latter case, $|\Sigma(m_\lambda)| \leq 1$ implies that $\Sigma(m_\lambda)$ will not be changed in further flow analysis according to Lemma 4, $n \in \Sigma(m_\lambda)$ implies that $n \in \Sigma(n'_\lambda)$ as n passes through n'_λ, and this implies that $n \in C \cap C'$ where C' is the set of CD in the previous call to Algorithm 3. However, this is impossible since *completed*(n_λ) is set to true for all $n_\lambda \in C'$ (line 16 in Algorithm 4) and another call to Algorithm 3 will not include any $n_\lambda \in C'$ in C since *completed*$(n_\lambda) = true$.

Thus, our only assumption that *completed*$(m_\lambda) = false$ and $|\Sigma(m_\lambda)| > 1$ cannot be true. □

Theorem 1. *Let $n \in N$ be any join node. The iterative flow analysis in Algorithm 4 terminates after computing all ϕ nodes for x from (N_μ, E_μ).*

Proof. Iterative flow analysis is performed by the `while` loop (lines 14-18) in Algorithm 4 after obtaining the initial set of CDs C_0 of x. C_0 contains all nodes $n_\lambda \in N_\mu$ that define the variable x. The flow analysis in Algorithm 2 is applied to compute Σ, all nodes in C_0 are marked as completed, and the analysis in Algorithm 3 then computes the set $C_1 = \mathit{Nodes}_\phi(C_0, \Sigma, G^\alpha_\mu) \subseteq N_\mu$ of ϕ nodes for x. No nodes in C_0 are included in C_1 (i.e. $C_0 \cap C_1 = \emptyset$) as C_1 only considers all nodes that are not marked as completed. According to Lemma 5, no other nodes in $N_\mu \setminus C_1$ can become ϕ nodes for x due to C_0. Then, C_1 becomes the new set of CDs. The iterative flow analysis repeats this process: it generates the set of ϕ nodes $C_2, \ldots, C_L$ in successive order and mark all nodes in C_i as completed before generating C_{i+1} for all $1 \leq i < L$ and $C_i \cap C_j = \emptyset$ for any $1 \leq i, j \leq L$. This process cannot repeat forever since $C_i \cap C_j = \emptyset$ and we eventually get $C_L = \emptyset$. According to Lemma 5, there can be no other ϕ nodes except C_i due to the CDs C_{i-1} for all $1 < i \leq L$. According to Lemma 6, $C_L = \emptyset$ implies that $\mathit{completed}(n_\lambda) = \mathit{true}$ or $|\Sigma(n_\lambda)| \leq 1$ for any $n_\lambda \in N_\mu$. Thus, the iterative flow analysis computes all ϕ nodes for x from (N_μ, E_μ) and no more ϕ nodes can be generated. □

5.2. Computational Complexity

We assume the following for the remainder of this section.

- $\mathbf{N}$ and $\mathbf{E}$ are the number of CFG nodes and edges in the CFG (N, E),
- $\mathbf{N}_\mu$ and $\mathbf{E}_\mu$ are the number of nodes and edges in the dependency graph (N_μ, E_μ),
- $\mathbf{V}$ is the number of variables in $\mathbf{Var}$, and
- K is the maximum in-degree of any join node in the CFG (N, E).

Lemma 7. *The worst-case time complexity of Algorithm 1 is $O(\mathbf{NV}K)$.*

Proof. The first `forall` loop at line 1 and the second `forall` loop at line 4 in Algorithm 1 will iterate $\mathbf{NV}$ and $\mathbf{E}$ times and each operation under these loops will take constant time. Thus, the worst case complexity of these two loops will be $O(\mathbf{NV})$ and $O(\mathbf{E})$. The complexity of Algorithm 1 is dominated by the `while` loop as follows.

- The `while` loop iterates as long as the worklist W is not empty. In each iteration of this loop, an edge is removed from W and marked as visited. An edge is included in W at line 13 if it was not visited before. Thus, this loop iterates at most $\mathbf{E}$ times.
- $|\mathcal{A}_\circ(n)(x)| \leq |\mathit{pred}(n)|$ for any $n \in N$ and any $x \in \mathbf{Var}$ according to Lemma 1 and $|\mathit{pred}(n)| \leq K$. Thus, $|\mathcal{A}_\circ(n)| \leq K\mathbf{V}$. Computing $\mathcal{A}_\bullet(n)$ at line 9 by means of applying the transfer function in Eq. 3 will require at most $K\mathbf{V}$ steps as each element in the set $\mathcal{A}_\circ(n)$ will have to be examined at most once.
- $|\mathcal{A}_\bullet(n)(x)| \leq 1$ for any $n \in N$ and any $x \in \mathbf{Var}$ according to Lemma 1. By choosing suitable data structures such as hash tables for $\mathcal{A}_\bullet$, computing $g(m, \mathcal{A}_\bullet(m))$ in Eq. 2 will require at most $\mathbf{V}$ operations and thus the instruction at line 12 in Algorithm 1 will require $K\mathbf{V}$ operations.
- $|\mathit{succ}(m)| \leq 2$ for any CFG node m and thus the `forall` loop at line 12 will iterate at most twice.

- All other operations in Algorithm 1 will take constant time. Thus the worst-case time complexity of the `while` loop is $O(\mathbf{EV}K)$.

Since any CFG node has at most two successors, $O(\mathbf{E}) = O(\mathbf{N})$, and $O(\mathbf{NV}K)$ is the worst-case time complexity of Algorithm 1. $\square$

Lemma 8. *The number of edges of any dependency graph (N_μ, E_μ) is at most $K\mathbf{N}_\mu$.*

Proof. According to Definition 3, each edge (m_λ, m'_u) in the graph (N_μ, E_μ) is constructed from an element $m_\lambda \in \mu(m', x)$ where $x \in \mathbf{Var}$ is the variable of interest for (N_μ, E_μ) and m'_u is a node in N_μ. $\mu(m', x)$ is $\mathcal{A}_\circ(m')(x)$ which contains RDs of x and each RD of m' is due to one of the incoming edges of m'. Since K is the maximum in-degree of any node, $|\mu(m', x)| \leq K$. Thus, each node $m'_u \in N_\mu$ in the graph (N_μ, E_μ) contributes at most K edges due to $\mu(m', x)$, and thus the graph can have at most $K\mathbf{N}_\mu$ edges (i.e. $\mathbf{E}_\mu \leq K\mathbf{N}_\mu$). $\square$

Lemma 9. *The worst-case time complexity of Algorithm 2 is $O(|C|K\mathbf{N}_\mu)$.*

Proof. We divide the worst-case time complexity of Algorithm 2 as follows:

- The worst-case time complexity of the first `forall` loop is $O(|C|\mathbf{N}_\mu)$ as this loop iterates $\mathbf{N}_\mu$ times, the `forall` loop inside it iterates $|C|$ times and all operations inside these loops take constant time.

- The worst-case time complexity of the `forall` loop at line 5 is $O(|C|K\mathbf{N}_\mu)$ as this loop iterates at most $|C|\mathbf{E}_\mu \leq |C|K\mathbf{N}_\mu$ times (Lemma 8) and inserting an element in the worklist W can be performed in constant time.

- In the `while` loop, an element (m_λ, n) is removed from the worklist W and marked as processed by setting $\mathcal{P}(m_\lambda, n)$ to *true*. Then, each successor p_λ of m_λ is paired with n and included in W if $\mathcal{P}(p_\lambda, n)$ is not processed (lines 15-16). W always contains pairs (m_λ, n) such that $(m_\lambda \in N_\mu$ and $n \in C$. Thus, W can contain at most $|C|\mathbf{N}_\mu$ elements and the `while` loop iterates at most $|C|\mathbf{N}_\mu$ times. The conditions in the `if` instruction at line 14 requires examining at most two elements of $\mu(m, x)$. Except the `forall` loop at line 15, all other operations in the `while` loop can be performed in $O(1)$ time, and thus the worst-case time complexity of the instructions from line 7 to 14 is $O(|C|\mathbf{N}_\mu)$.

- The `forall` loop at line 15 can visit all the outgoing edges of m_λ once the pair (m_λ, n) is removed from W. Each node m_λ can have at most K successors as illustrated in the proof of Lemma 8. Since the `while` loop can iterate at most $|C|\mathbf{N}_\mu$ times, and the instruction at line 16 can be performed in constant time, the worst-case time complexity of the instructions in lines 15-16 is $O(|C|K\mathbf{N}_\mu)$.

Since $O(|C|K\mathbf{N}_\mu)$ dominates other computational time complexity, $O(|C|K\mathbf{N}_\mu)$ is the worst-case time complexity of Algorithm 2. $\square$

Lemma 10. *The worst-case time complexity of Algorithm 3 is $O(K\mathbf{N}_\mu)$.*

Proof. The worst-case complexity of initializing the worklist W at line 2 is $O(|C|K)$ since it requires visiting all the successors of each node in C and any node in C has at most K successors (as illustrated in the proof of Lemma 8). Since $|C| \leq \mathbf{N}_\mu$, $O(K\mathbf{N}_\mu)$ is the worst-case complexity of initializing W.

In each iteration of the `while` loop, a node $n_\lambda \in N_\mu \cap W$ is removed from W. If *completed*(n_λ) is $false$ and $|\Sigma(n_\lambda)| \leq 1$, *completed*$(n_\lambda)$ is set to true and the successors of n_λ are included in W; W is not updated otherwise. Thus, the `while` loop does not visit a node in N_μ more than once and it can iterate at most $\mathbf{N}_\mu$ times. All operations in the `while` loop can be performed in $O(1)$ time except the instruction at line 11 to update W. Updating W at line 11 requires visiting the edges in E_μ and no edge needs to be visited more than once since each node $n_\lambda \in N_\mu$ can be included in W at most once. Thus, during the entire execution of the `while` loop, W can be updated at most $\mathbf{E}_\mu$ times. So, the worst-case time complexity of the `while` loop, which is dominated by the update instruction at line 11, is $O(\mathbf{N}_\mu + \mathbf{E}_\mu)$. Note that $\mathbf{N}_\mu$ is additive to $\mathbf{E}_\mu$ instead of being multiplicative in the complexity order since only a part of the edges in E_μ are visited for each visited node $n_\lambda \in N_\mu$ in the `while` loop which adds up to $\mathbf{E}_\mu$ by the end of the loop. $O(\mathbf{N}_\mu + \mathbf{E}_\mu)$ is effectively $O(K\mathbf{N}_\mu)$ since $\mathbf{E}_\mu \leq K\mathbf{N}_\mu$, and $O(K\mathbf{N}_\mu)$ is the worst-case time complexity of Algorithm 3. □

Theorem 2. *Given the CFG (N, E) and the set* **Var** *of program variables, the worst-case time complexity of Algorithm 4 is $O(K\mathbf{N}^3\mathbf{V} + K\mathbf{N}^2\mathbf{V}^2)$.*

Proof. First, we examine the worst-case time complexity of the `forall` loop at line 6.

- The *createGraph* procedure at line 7 generates the graph (N_μ, E_μ) according to Definition 3. It requires examining at most $\mathbf{N}_\mu$ number of μ sets each containing at most K elements to construct $\mathbf{N}_\mu$ nodes and $\mathbf{E}_\mu$ edges. Thus, the worst-case time complexity of the *createGraph* procedure is $O(\mathbf{N}_\mu + \mathbf{E}_\mu)$ which is equivalent to $O(K\mathbf{N}_\mu)$ since $\mathbf{E}_\mu \leq K\mathbf{N}_\mu$ (Lemma 8).

- The worst-case time complexity of constructing the C_0 set at line 8 is $O(\mathbf{N}_\mu\mathbf{V})$, and the `forall` loop at line 9 and the `forall` loop at line 12 each are $O(\mathbf{N}_\mu)$.

- Consider the `while` loop which spans between lines 14 and 18 in Algorithm 4. The $Flow_C$ procedure at line 15 and the *computeNodes*$_\phi$ procedure at line 17 apply Algorithm 2 and Algorithm 3 having the worst-case time complexity $O(|C|K\mathbf{N}_\mu)$ and $O(K\mathbf{N}_\mu)$ respectively (Lemma 9 and 10). The complexity of the `forall` loop at line 16 is $O(\mathbf{N}_\mu)$ since this loop iterates at most $|C| \leq \mathbf{N}_\mu$ times containing an instruction that can be executed in constant time. The union operation at line 18 will require at most $\mathbf{N}_\mu$ operations since the size of the C set obtained from Algorithm 3 cannot include more than $\mathbf{N}_\mu$ elements. So, $O(|C|K\mathbf{N}_\mu)$ is the dominating cost in each iteration of this loop.

  The C set obtained from Algorithm 3 includes all nodes $m_\lambda \in N_\mu$ such that *completed*$(m_\lambda) = false$ and *completed*(m_λ) is set to *true* afterwards. Thus, if C_1 and C_2 are two C sets in two iterations of this loop, then $C_1 \cap C_2 = \emptyset$. Suppose this loop iterates L times generating the sequence of C sets $C_1, \ldots, C_L$ such that $C_L = \emptyset$ and C_0 is the initial C set. Thus, $\bigcup_{0 \leq i \leq L} C_i \subseteq N_\mu$ and $C_i \cap C_j = \emptyset$ for all $1 \leq i \neq j \leq L$. Thus, we can express the worst-case time complexity of the `while` loop as $O(|C_0|K\mathbf{N}_\mu + \ldots + |C_{L-1}|K\mathbf{N}_\mu)$ which is equivalent to $O(K{\mathbf{N}_\mu}^2)$ since $|C_0| + \ldots + |C_{L-1}| \leq \mathbf{N}_\mu$.

- The `forall` loop at line 19 will execute $\mathbf{N}_\mu$ times. In each iteration of this loop, the *reConstruct*Σ procedure in Algorithm 5 is called. In Algorithm 5, the `forall` loop at line 2 iterates at most K times since the predecessors of any node $n_\lambda \in N_\mu$ is obtained from the $\mu(n, x)$ set in Definition 3 which can contain at most K elements. The conditions in the `if` instruction at line 3 checks if n' is a CD or not. All operations in the `if` block at lines 3-4

#	benchmarks	App. area	KLOC	nCFG	nVars	#Proc
1	544.nab_r	Molecular dynamics	24	104080	36920	6540
2	557.xz_r	data compression	33	20400	3450	3620
3	505.perlbench_r	Perl interpreter	362	636650	126220	18158
4	502.gcc_r	GNU C compiler	1304	3666660	762610	178270
5	505.mcf_r	Route planning	3	8240	2940	400
6	525.x264_r	Video compression	96	149130	33740	14490
7	538.imagick_r	Image manipulation	259	309310	101720	25860

Table 2: SPEC CPU2017 [7] benchmarks containing C code for a diverse range of applications

can be performed in constant time except checking the inclusion operation $x \in def(n)$ which will take at most $\mathbf{V}$ operations.

Since $reConstruct\Sigma$ procedure is called after obtaining an empty set C_L from Algorithm 3 and either $completed(m_\lambda) = true$ or $|\Sigma(m_\lambda)| \leq 1$ for any $m_\lambda \in N_\mu$ according to Lemma 6, if m_λ is not a CD, then we must have $|\Sigma(m_\lambda)| \leq 1$. Thus, $|\Sigma(n'_\lambda)| \leq 1$ in the `else` block at line 6 and hence line 6 can be performed in constant time. Thus, the worst-case time complexity of the $reConstruct\Sigma$ procedure is $O(K\mathbf{V})$ and the worst-case time complexity of the `forall` loop at line 19 is $O(K\mathbf{N}_\mu\mathbf{V})$.

Thus, $O(K{\mathbf{N}_\mu}^2 + K\mathbf{N}_\mu\mathbf{V})$ is the dominating worst-case time complexity of each iteration of the `forall` loop at line 6. This loop iterates at most the number of join nodes in the CFG (N, E). Moreover $\mathbf{N}_\mu \leq \mathbf{N}$. Thus, the worst-case time complexity of this loop is $O(K\mathbf{N}^3 + K\mathbf{N}^2\mathbf{V})$. This is the dominating cost of the `forall` loop at line 3 which iterates $\mathbf{V}$ times. Thus, the worst-case time complexity of the `forall` loop at line 3 is $O(K\mathbf{N}^3\mathbf{V} + K\mathbf{N}^2\mathbf{V}^2)$. Since the worst-case time complexity of Algorithm 1 at line 1 is $O(K\mathbf{N}\mathbf{V})$ and the worst-case cost of the `forall` loop at line 2 can be $O(\mathbf{N}\mathbf{V})$, $O(K\mathbf{N}^3\mathbf{V} + K\mathbf{N}^2\mathbf{V}^2)$ is the worst-case time complexity of Algorithm 4. □

6. Experimental Evaluation

We implemented both ours and the DF-based ϕ-placement approach of Cytron et al. [1] in the Clang/LLVM compiler framework [8]. We performed the experiments on an Intel(R) Core(TM) i7-7567U CPU with 3.50GHz leveraging a number of SPEC CPU2017 [7] benchmarks consisting of approximately 2081 KLOC. SPEC is a set of industry-standardized, CPU intensive suites for measuring and comparing, among others, compute intensive performance and compilers. Table 2 shows the seven benchmarks selected from the SPEC CPU2017 [7], which are written in C code. *nCFG*, *nVars*, and #Proc indicate the number of basic blocks in the CFG representation of the C code, program variables, and procedures in the respective benchmarks, respectively.

Note that we could not analyze some source files as Clang excluded the code in the source files or failed to parse them. If the path of some headers included in a source file could not be resolved, Clang failed to parse the source file. Also, Clang excluded the code in a source file when the code was under macros like `ifdef` and the Clang preprocessor had no definitions of these macros. Thereby, we excluded these files from our analysis. We analyzed each procedure separately. The number of program variables reported in the *nVar* column of Table 2 is the total number of variables of all analyzed procedures in the respective benchmark. Thus, global variables that may be used in multiple procedures are accounted multiple times in the *nVar* column.

	benchmarks	ϕ_{RD}	ϕ_{DF}	ϕ_{RDE}	ϕ_{DFE}	%ϕ-sup	%ϕ-supE
1	544.nab_r	26960	44600	2400	7400	65.43	51.47
2	557.xz_r	820	1300	150	320	58.53	46.27
3	505.perlbench_r	124210	220340	19370	46830	77.39	65.50
4	502.gcc_r	430910	807190	65530	191290	87.32	68.56
5	505.mcf_r	3090	4730	250	710	53.07	41.55
6	525.x264_r	23750	40030	3050	7500	68.55	57.15
7	538.imagick_r	38320	67750	10730	31590	76.80	31.06
8	(Average)					69.59	51.65

Table 3: Comparing the ϕ-functions generated by our ϕ-placement approach (Algorithm 4) and the approach of Cytron et al. [1]

	benchmarks	P_2	P_5	P_{5M}	%P_2	%P_5	%P_{5M}
1	544.nab_r	3594	2279	667	54.95	34.85	10.20
2	557.xz_r	3365	242	13	92.96	6.69	0.36
3	505.perlbench_r	9469	5211	3478	52.15	28.70	19.15
4	502.gcc_r	116257	45567	16446	65.21	25.56	9.23
5	505.mcf_r	143	183	74	35.75	45.75	18.5
6	525.x264_r	10911	2617	962	75.30	18.06	6.64
7	538.imagick_r	18583	5958	1319	71.86	23.04	5.10
8	(Average)				65.63	25.09	9.28

Table 4: Comparing execution time of our approach and the approach of Cytron et al.

We compare the number of ϕ-functions generated by our approach (Algorithm 4) and the approach of Cytron et al., in Table 3. The symbols ϕ_{RD} and ϕ_{DF} in Table 3 indicate the total number of ϕ-functions generated by our RD-based approach and the DF-based approach of Cytron et al. in the respective benchmarks. Moreover, ϕ_{RDE} and ϕ_{DFE} in the table indicate the number of ϕ-functions generated at the exit node of the CFG. Local variables at the exit node are not live variables and hence we can remove ϕ functions generated at this node. If we assume that all program variables are defined at the beginning (i.e. *def*(*entry*) is the set of all program variables of the analyzed procedure), then the DF-based approach and ours generate exactly the same number of ϕ-functions. However, our RD-based approach considers that only global variables and formal parameters are defined at the entry node of the CFG.

As shown in Table 3, ϕ_{DF} is significantly higher than ϕ_{RD}. Since our approach generates ϕ-functions based on witnessing RDs, our results on generating ϕ-functions are more precise. In %ϕ-sup and %ϕ-supE columns in Table 3, we report on the number of superfluous ϕ-functions in percentage as generated by the DF-based approach compared to our RD-based approach due to the assumption that all program variable definitions are at the entry node of the CFG. We calculate %ϕ-sup and %ϕ-supE as follows:

$$\phi\text{-}sup = (\tfrac{\phi_{DF}}{\phi_{RD}} - 1) \cdot 100$$
$$\phi\text{-}supE = (\tfrac{(\phi_{DF}-\phi_{DFE})}{\phi_{RD}-\phi_{RDE}} - 1) \cdot 100$$

As it can be seen in Table 3 and Figure 6(a), the DF-based approach generates more than 50% superfluous ϕ-functions in all benchmarks, up to 87.32% and on average 69.59% superfluous

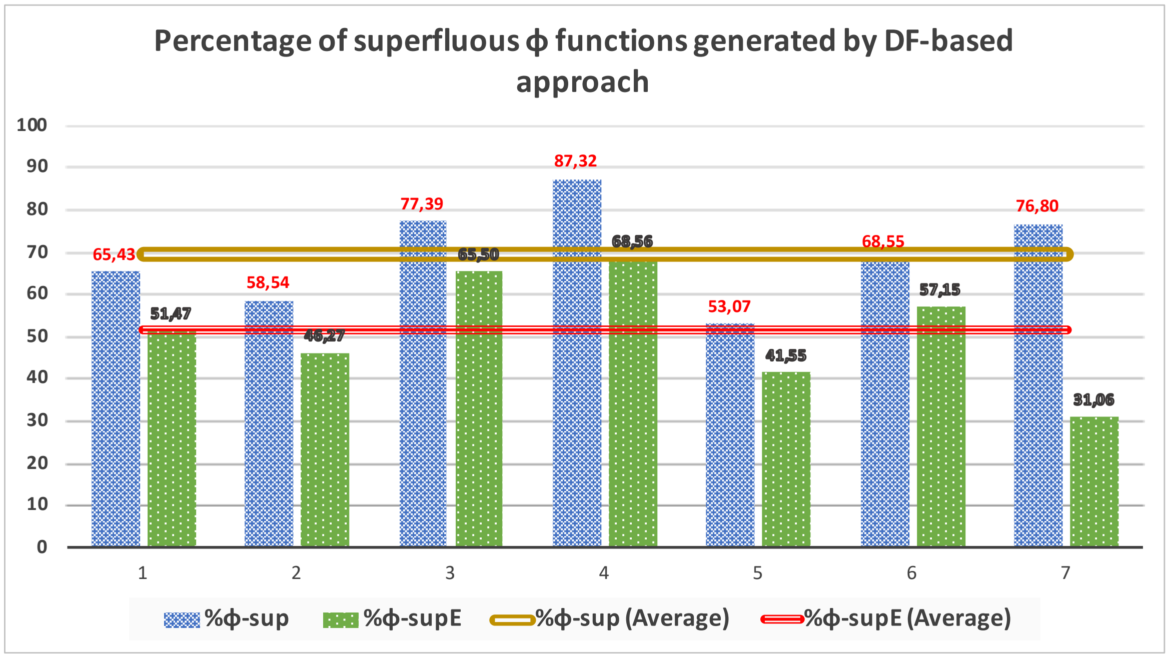
ϕ-functions when comparing the results with our RD-based approach. If we remove all ϕ-functions from the exit node of the CFG, then the DF-based approach generates on average 51.65% and up to 68.56% superfluous ϕ-functions.

Since we are computing more information to be precise in generating ϕ-functions, our approach is expected to be computationally more expensive than DF-based approaches. In Table 4, we compare the average execution time of our approach and the approach of Cytron et al. [1]. In order to make such a comparison, we calculate the mean time of 10 executions of the approaches. P_2, P_5, and P_{5M} columns indicate the number of analyzed procedures where the execution time of our RD-based approach is at most two times, more than two times to at most five times, and more than five times than the DF-based approach respectively. We calculate the percentage of procedures that could be analyzed by at most i times by the RD-based approach compared to the DF-based approach in the $\%P_i$ column for any $i \in \{2, 5, 5M\}$ where $5M$ means "more than 5". We calculate $\%P_i = \frac{P_i}{P} \cdot 100$, where P is the total number of procedures in the respective benchmarks provided in the #Proc column in Table 2. As Table 4 and Figure 6(b) indicate, our approach could finish computing ϕ-functions for:

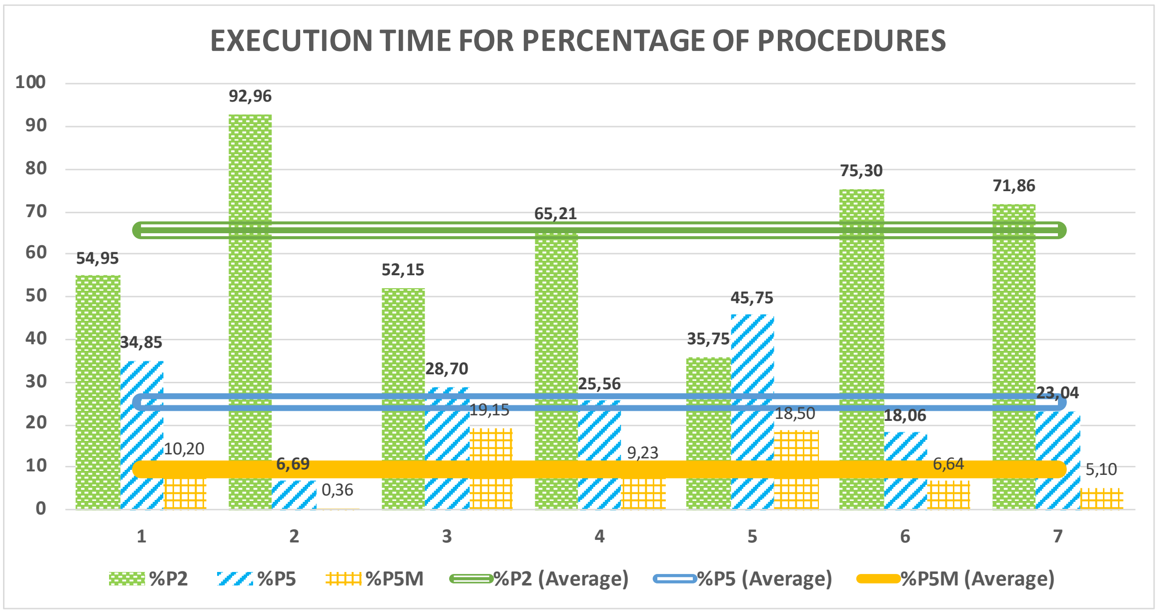
- 65.63% procedures on an average and up to 92.96% procedures with the execution time within twice the execution time of the DF-based approach;
- 25.09% procedures on an average and up to 45.75% procedures with the execution time more than twice and within five times the execution time of the DF-based approach;
- 9.28% procedures on an average and up to 19.15% procedures with the execution time more than five times the execution time of the DF-based approach.

In Table 5 we closely inspect the analysis results obtained from the DF-based and RD-based approaches on some of the source files from the 505.perlbench_r benchmark. T indicates the execution time of 30 executions of the analyses in seconds. T_1 column indicates the execution time of Algorithm 1, T_{Iter} column indicates the average execution time of Algorithm 4 for iterative flow analysis, which excludes the execution time of Algorithm 1, T_{RD} is the sum of T_1 and T_{Iter}, which is the total execution time of our RD-based approach, and T_{DF} indicates the execution time of the DF-based approach. As it can be seen in Figure 7 (a), (b), and in the rightmost column in Table 5, the DF-based approach generates up to 169% and on an average 74% superfluous ϕ-functions when comparing the results with our RD-based approach from the selected source files of 505.perlbench_r benchmark listed in Table 5 due to the limiting assumption that all program variables are defined at the beginning. This average result (i.e. 74% superfluous ϕ-functions) is within the range of the superfluous ϕ-functions generated by the DF-based approach compared to the RD-based approach in our benchmarks, as shown in the $\%\phi$-sup column in Table 3. The peak number (i.e. 169% superfluous ϕ-functions) on the individual source code in the 505.perlbench benchmark is also not uncommon in the separate analysis of individual source code in other benchmarks.

Regarding the execution time, the DF-based approach performs better than ours. As can be seen from the T_{RD} and T_{DF} columns, there is no noticeable execution time differences between the two approaches in most cases. However, there exist few cases, namely *pp_pack.c, regcomp.c, toke.c, regexec.c* (more particularly, some functions in these source codes), that take considerably more time for our RD-based approach. The shape of the execution time graphs in Figure 8(a) and 8(b) reveals that this is due to the iterative flow analysis in Algorithm 4. Since Algorithm 1 visits each edge exactly once per-variable, it is efficient in practice, as well as Algorithm 4, except some exceptional cases. Evidently, our approach produces significantly fewer ϕ-functions and also



(a)

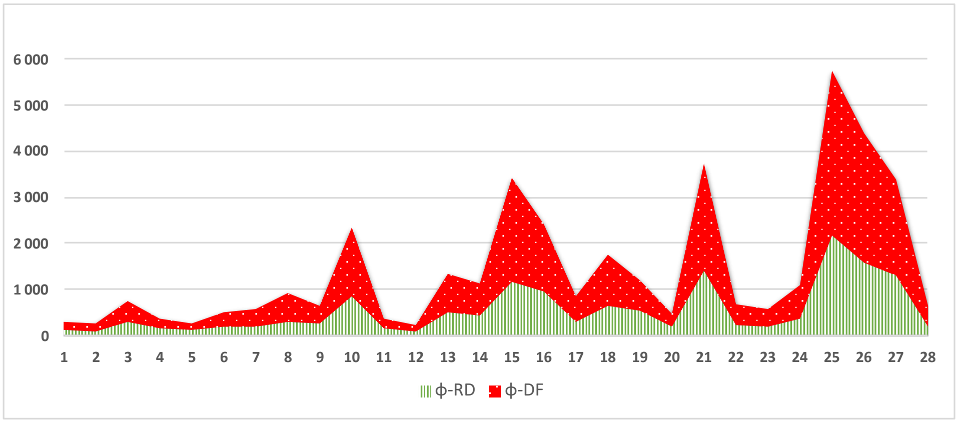


(b)

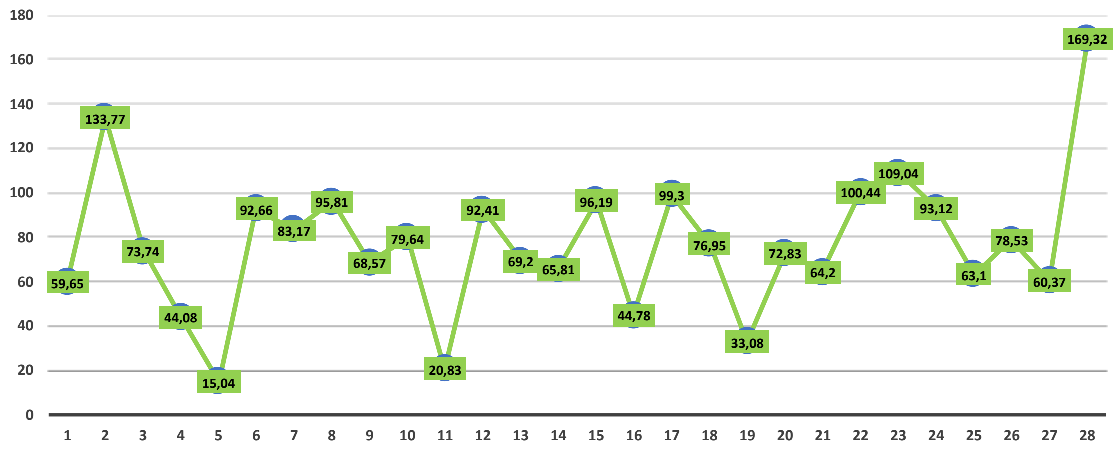
Figure 6: (a) Percentage of superfluous ϕ-functions computed by DF-based approach in comparison with our RD-based approach in all benchmarks, (b) Execution time of RD-based approach for percentage of procedures with respect to DF-based approach. Numbers 1-7 in the horizontal axis indicate the benchmarks in Table 3

#	Code	nCFG	nVars	T_1	T_{Iter}	T_{RD}	T_{DF}	ϕ_{RD}	ϕ_{DF}	%ϕ-sup
1	scope.c	669	241	0.078	0.088	0.166	0.034	114	182	59.65
2	Opcode.c	372	236	0.013	0.021	0.035	0.024	77	180	133.77
3	pp_sort.c	612	228	0.053	0.107	0.160	0.037	278	483	73.74
4	vutil.c	621	89	0.030	0.058	0.088	0.036	152	219	44.08
5	perly.c	454	36	0.818	0.112	0.931	0.028	113	130	15.04
6	pad.c	689	236	0.045	0.083	0.128	0.040	177	341	92.66
7	util.c	1426	402	0.043	0.053	0.097	0.070	202	370	83.17
8	pp_sys.c	2212	575	0.094	0.155	0.249	0.118	310	607	95.81
9	doop.c	781	189	0.052	0.116	0.168	0.049	245	413	68.57
10	pp.c	5470	1048	0.283	0.689	0.972	0.326	835	1500	79.64
11	numeric.c	737	125	0.035	0.048	0.084	0.041	168	203	20.83
12	universal.c	696	284	0.022	0.036	0.058	0.039	79	152	92.41
13	pp_hot.c	2623	456	0.181	0.511	0.692	0.165	500	846	69.20
14	utf8.c	2018	444	0.122	0.262	0.384	0.126	427	708	65.81
15	op.c	6793	1303	0.643	1.479	2.122	0.401	1156	2268	96.19
16	pp_pack.c	2707	270	0.709	7.591	8.300	0.191	956	1442	44.78
17	hv.c	1388	416	0.090	0.202	0.292	0.075	284	566	99.30
18	pp_ctl.c	3100	748	0.274	0.559	0.834	0.202	629	1113	76.95
19	perl.c	1385	673	0.167	0.383	0.550	0.086	522	679	33.08
20	doio.c	1018	223	0.084	0.153	0.238	0.061	173	299	72.83
21	sv.c	5927	1147	0.606	2.737	3.344	0.381	1419	2330	64.20
22	mg.c	1623	393	0.169	0.146	0.315	0.091	219	459	100.44
23	mro_core.c	891	152	0.067	0.248	0.316	0.053	188	393	109.04
24	gv.c	2500	477	0.231	0.799	1.031	0.248	378	730	93.12
25	regcomp.c	8633	1094	1.677	25.396	27.073	2.792	2184	3562	63.10
26	toke.c	5922	757	5.525	27.759	33.284	0.477	1574	2810	78.53
27	regexec.c	4959	445	1.977	11.427	13.405	0.494	1307	2096	60.37
28	dump.c	1482	274	0.178	1.347	1.525	0.110	176	474	169.32

Table 5: Execution time and number of ϕ-functions generated by our approach and the approach of Cytron et al.

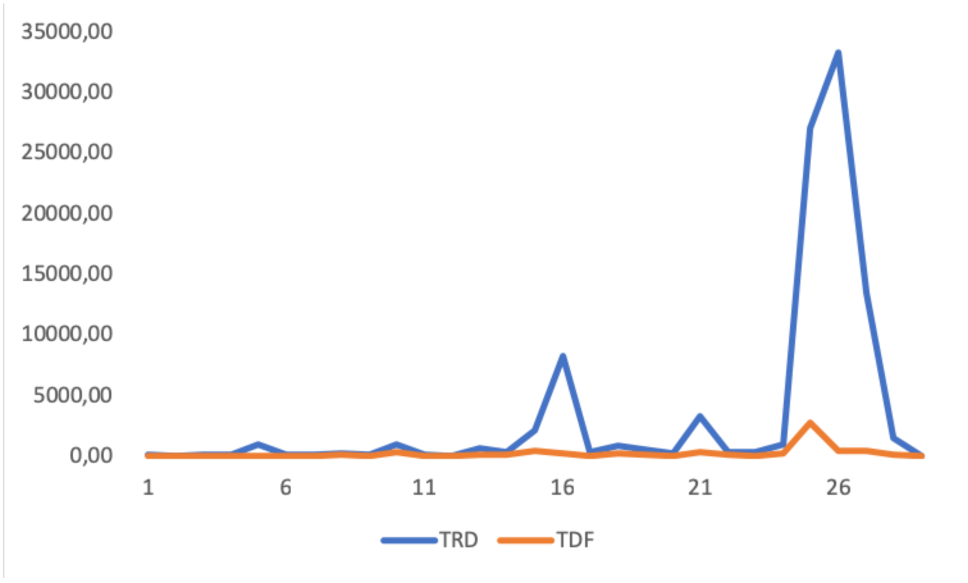


(a)

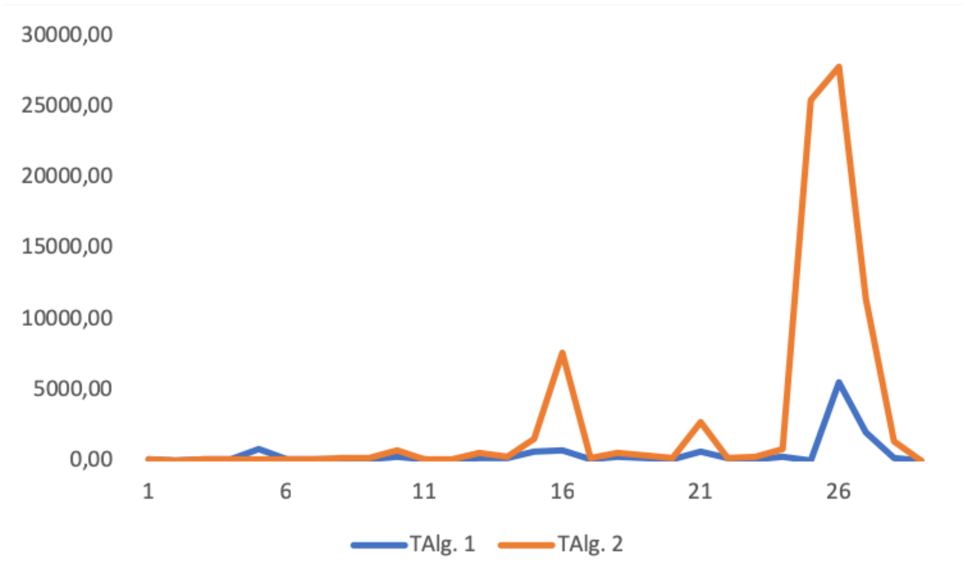


(b)

Figure 7: Comparing the results in Table 5: (a) the number of generated ϕ-functions by DF and RD-based approaches, and (b) percentage of superfluous ϕ functions generated by the DF-based approach in comparison with the RD-based approach. Numbers in the horizontal axis indicate the source code listed in Table 5



(a)



(b)

Figure 8: Comparing the results on execution time in Table 5: (a) execution times by the DF-based and RD-based SSA construction approaches, and (b) execution times of Algorithm 1 (i.e. TAlg.1) versus Algorithm 4-Algorithm 1 (i.e. TAlg.2). Numbers in the horizontal axis indicate the source code listed in Table 5

generates RD information that can reduce the lookup time of reaching variable definitions during the renaming phase of the SSA construction.

7. Related Work

The first approach to generate the set of nodes that require *pseudo assignments*, or ϕ-functions, dates back to the work of Shapiro and Saint [9]. Subsequent contributions include the work of (i) Reif and Tarjan [10] providing a complex ϕ-placement algorithm in a bottom-up walk of the dominator tree, and (ii) Rosen et al. [11] generating SSA form for reducible programs. However, Cytron et al. [1] presented the first practically efficient algorithm based on computing dominance frontiers to generate the SSA intermediate representation of programs. This algorithm behaves linearly in practice in most typical cases that became popular and widely used afterwards. Since then, computing dominance frontiers has become the most common approach to compute the *join sets* in SSA construction algorithms. Even though the algorithm of Cytron et al. [1] is practically efficient, it has the nonlinear computational complexity since the size of the dominance frontiers sets can have quadratic growth in terms of the size of the CFG. Thus, myriad efforts were given to improve the theoretical complexity of dominance frontiers based ϕ placement algorithms.

Algorithms having linear or almost linear complexity for ϕ-placements in the size of the CFG have been proposed, but are practically not as efficient as the original classic algorithm of Cytron et al. The theoretical complexity of these algorithms is expressed per variable in the program code; this requires repeated application of these algorithms for multiple variables and thus performs many redundant computations that lead to reduced efficiency. Sreedhar and Gao [12] proposed an algorithm that can construct DF sets on-the-fly by using so-called DJ graphs and reported a linear time behavior based-on complexity analysis and observations on some practical applications. They claimed that it outperformed the one of Cytron et al. by factors of 5 to 10 on their experimental benchmarks. Bilardi and Pingali [13] presented a systematic study of ϕ-placement algorithms by using the merge relation on CFG nodes and derived all known properties of the SSA form. Their framework led them to present both known and new algorithms for ϕ-placement and the asymptotic complexity of the new algorithms match the known best algorithms in the literature. By using their framework, they evaluated the algorithm of Sreedhar and Gao and discovered that this algorithm is not competitive with the algorithm of Cytron et al. Choi et al. [14] presented an approach to compute a variation of SSA program called *pruned* SSA form that removed dead ϕ-functions. Their approach needs prior computation of the dominator tree and the dominance frontiers. Briggs et al. [15] provided improvements of SSA construction algorithms presented by Cytron et al. in which they computed *semi-pruned* SSA, a smaller SSA form than the one computed by Cytron et al. Even though the semi-pruned SSA is smaller than the *minimal* SSA computed by Cytron et al., it is still based on computing the dominance frontier. All these methods are based on computing dominance frontiers that have the implicit assumption that all program variables are defined at the beginning of the program that leads them to produces superfluous ϕ-functions. On the other hand, our RD-based ϕ-placement algorithm can freely choose the set of program variables defined at the beginning, and thus able to produce more accurate ϕ-functions.

There exists a number of simple SSA construction algorithms that work directly on the *abstract syntax tree* (AST) representation of the programs. Brandis and Mössenböck [16] provided a single-pass analysis of structured programs to construct SSA form that does not require constructing the dominator tree or dominance frontier relation. Click and Paleczny [17, 18] generated ϕ-functions during generating graph-based intermediate program representation, which is neither a pruned nor a minimal SSA form [19]. Aycock and Horspool's [20] SSA construction algorithm produces minimal SSA form for reducible programs. Braun et al. [19] presented a simple SSA construction algorithm from the AST representation of the program that neither computes dominator trees nor dominance frontiers. However, the authors reported that the number of generated SSA instructions produced by their methods is similar to what Cytron et al.'s method computed when implemented in LLVM.

Our SSA construction approach which only deals with scalar variables is orthogonal to the practical methods modeling aliasing, non-scalar variables such as structures or arrays, and indirect memory operations in the SSA form [21, 22, 23].

8. Conclusion and Future work

Most SSA construction algorithms are based on computing dominance frontiers, which is very efficient for reducible programs. However, the correctness and precision condition (i.e., $DF^+(S) = J^+(S)$) of any DF-based method depends on the limiting assumption that all program variables are defined at the beginning (i.e., S contains the *entry* node), which is not always the case for local variables. To understand the impact of this assumption, we have developed a novel RD-based ϕ-placement algorithm that generates the optimal number of ϕ-functions without considering the limiting assumption and is fairly efficient in most cases. We assume that all global variables and formal parameters are defined at the beginning of the program. Our experimental evaluation reveals that the reference DF-based approach generates (i) up to 87% and on average 69% superfluous ϕ-functions on all benchmarks, and (ii) up to 169% and on an average 74% superfluous ϕ-functions on an individual benchmark when comparing the results with our RD-based method. Our RD-based ϕ-placement algorithm can be seen as a reference method computing an optimal number of ϕ-functions. Future work includes improving the theoretical complexity as well as performing practical optimizations of the RD-based method in generating different kinds of SSA programs.

Acknowledgment

This research is supported by the Knowledge Foundation through the MOMENTUM and the HERO projects led by the Mälardalen University.

CRediT author statement

Abu Naser Masud: Conceptualization, Investigation, Methodology, Software, Data curation, Writing- Original draft preparation. Visualization, Validation. **Federico Ciccozzi:** Writing - Review and Editing, Project administration, Funding acquisition, Validation.

References

[1] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, ACM Trans. Program. Lang. Syst. 13 (4) (1991) pp. 451–490.

[2] M. Weiss, The Transitive Closure of Control Dependence: The Iterated Join, ACM Lett. Program. Lang. Syst. 1 (2) (1992) pp. 178–190.

[3] F. Nielson, H. R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[4] A. N. Masud, F. Ciccozzi, Towards constructing the SSA form using reaching definitions over dominance frontiers, in: IEEE International Working Conference on Source Code Analysis and Manipulation, 2019.

[5] M. S. Hecht, J. D. Ullman, Flow Graph Reducibility, in: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, STOC '72, ACM, New York, NY, USA, 1972, pp. 238–250.

[6] F. E. Allen, Control Flow Analysis, in: Proceedings of a Symposium on Compiler Optimization, ACM, New York, NY, USA, 1970, pp. 1–19.

[7] J. Bucek, K.-D. Lange, J. v. Kistowski, SPEC CPU2017: Next-Generation Compute Benchmark, in: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18, ACM, New York, NY, USA, 2018, pp. 41–42.

[8] C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, IEEE Computer Society, Washington, DC, USA, 2004.

[9] R. M. Shapiro, H. Saint, The representation of algorithms, Tech. Rep. Report CA-7002-1432, Massachusetts Computer Associates (02 1970).

[10] J. H. Reif, Symbolic Program Analysis in Almost Linear Time, in: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '78, ACM, New York, NY, USA, 1978, pp. 76–83.

[11] B. K. Rosen, M. N. Wegman, F. K. Zadeck, Global Value Numbers and Redundant Computations, in: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88, ACM, New York, NY, USA, 1988, pp. 12–27.

[12] V. C. Sreedhar, G. R. Gao, A Linear Time Algorithm for Placing Φ-nodes, in: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95, ACM, New York, NY, USA, 1995, pp. 62–73.

[13] G. Bilardi, K. Pingali, Algorithms for Computing the Static Single Assignment Form, J. ACM 50 (3) (2003) pp. 375–425.

[14] J.-D. Choi, R. Cytron, J. Ferrante, Automatic Construction of Sparse Data Flow Evaluation Graphs, in: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '91, ACM, New York, NY, USA, 1991, pp. 55–66.

[15] P. Briggs, K. D. Cooper, T. J. Harvey, L. T. Simpson, Practical Improvements to the Construction and Destruction of Static Single Assignment Form, Softw. Pract. Exper. 28 (8) (1998) pp. 859–881.

[16] M. M. Brandis, H. Mössenböck, Single-pass Generation of Static Single-assignment Form for Structured Languages, ACM Trans. Program. Lang. Syst. 16 (6) (1994) pp. 1684–1698.

[17] C. Click, M. Paleczny, A Simple Graph-based Intermediate Representation, SIGPLAN Not. 30 (3) (1995) pp. 35–49.

[18] C. Click, M. Paleczny, A Simple Graph-based Intermediate Representation, in: Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations, IR '95, ACM, New York, NY, USA, 1995, pp. 35–49.

[19] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, A. Zwinkau, Simple and Efficient Construction of Static Single Assignment Form, in: Proceedings of the 22Nd International Conference on Compiler Construction, CC'13, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 102–122.

[20] J. Aycock, R. N. Horspool, Simple Generation of Static Single-Assignment Form, in: Proceedings of the 9th International Conference on Compiler Construction, CC '00, Springer-Verlag, London, UK, 2000, pp. 110–124.

[21] D. Novillo, Memory SSA - A Unified Approach for Sparsely Representing Memory Operations, in: Proceedings of the GCC Developers' Summit, 2007.

[22] F. C. Chow, S. Chan, S.-M. Liu, R. Lo, M. Streich, Effective Representation of Aliases and Indirect Memory Operations in SSA Form, in: Proceedings of the 6th International Conference on Compiler Construction, CC '96, Springer-Verlag, London, UK, 1996, pp. 253–267.

[23] Y. Sui, H. Yan, Z. Zheng, Y. Zhang, J. Xue, Parallel construction of interprocedural memory SSA form, Journal of Systems and Software 146 (2018) pp. 186–195.