Simple and Efficient Computation of Minimal Weak Control Closure

Abu Naser Masud[0000−0002−4872−1208]

School of Innovation, Design and Engineering, Mälardalen University, Sweden
{masud.abunaser}@mdh.se

Abstract. Control dependency is a fundamental concept in many program analyses, transformation, parallelization, and compiler optimization techniques. An overwhelming number of definitions of control dependency relations are found in the literature that capture various kinds of program control flow structures. Weak and strong control closure (WCC and SCC) relations capture nontermination insensitive and sensitive control dependencies and subsume all previously defined control dependency relations. In this paper, we have shown that static dependency-based program slicing requires the repeated computation of WCC and SCC. The state-of-the-art WCC algorithm provided by Danicic et al. has the cubic worst-case complexity in terms of the size of the control flow graph and is a major obstacle to be used in static program slicing. We have provided a simple yet efficient method to compute the minimal WCC which has the quadratic worst-case complexity and proved the correctness of our algorithms. We implemented ours and the state-of-the-art algorithms in the Clang/LLVM compiler framework and run experiments on a number of SPEC CPU 2017 benchmarks. Our method performs a maximum of 23.8 times and on average 10.6 times faster than the state-of-the-art method. The performance curves of our WCC algorithm for practical applications are closer to the NlogN curve in the microsecond scale. Evidently, we improve the practical performance of WCC computation by an order of magnitude.

Keywords: Control dependency · Weak control closure · Strong control closure · Program slicing · Nontermination (in)sensitive

1 Introduction

Control dependency is a fundamental concept in many program analyses, transformation, parallelization and compiler optimization techniques. It is used to express the relation between two program statements such that one decides whether the other statement can be executed or not. One of the key applications of control dependency is program slicing [**?**] that transforms an original program into a sliced program with respect to a so-called slicing criterion. The slicing criterion specifies the variables at a particular program point that will affect the execution of the sliced program. All program instructions in the original program that does not affect the slicing criterion are discarded from the sliced code.

Control dependency is used to identify the program instructions that indirectly affect the slicing criterion due to the execution of conditional expressions in the loops or conditional instructions.

The standard definition of control dependency provided by Ferrante et al. [**?**] has been widely used for over two decades. This definition is provided at the level of the *control flow graph* (CFG) representation of a program assuming that the CFG has a unique *end* node (i.e. the program has a single exit point). Several recent articles on control dependency illustrate that this definition does not sufficiently capture the intended control dependency of programs having the modern programming language features. For instance, the *exception* or *halt* instructions cause multiple exits of the programs, or reactive systems, web services or distributed real-time systems have nonterminating program instructions without an end node. The standard definition of control dependency did not intend to handle the above systems. The possibility of having nontermination in the program code introduces two different types of control dependency relations: the *weak* and *strong* control dependencies that are nontermination insensitive and nontermination sensitive. One of the distinguishing effects between the two types of control dependencies is that an original nonterminating program remains nonterminating or may be transformed into a terminating program if the slicing method uses *strong* or *weak* control dependence respectively.

Numerous authors provided an overwhelming number of definitions of control dependencies [**?**,**?**,**?**,**?**,**?**] given at the level of CFG and describe computation methods to obtain such dependencies. Danicic et al. [**?**] unified all previously defined control dependence relations by providing the definitions and theoretical insights of *weak* and *strong control-closure* (WCC and SCC) that are most generalized and capture all non-termination insensitive and nontermination sensitive control dependence relations. Thus, WCC and SCC subsume all control dependency relations found in the literature. However, Danicic et al. provided expensive algorithms to compute WCC and SCC. In particular, the algorithms for computing WCC and SCC have the cubic and quartic worst-case asymptotic complexity in terms of the size of the CFG. We have shown that static program slicing requires the repeated computation of WCC and/or SCC. The state-of-the-art WCC and SCC algorithms are not only expensive, but the use of these algorithms in client applications such as program slicing will make these applications underperforming.

In this article, we have provided a simple and efficient method to compute WCC. We have formalized several theorems and lemmas demonstrating the soundness, minimality, and complexity of our algorithm. Our WCC algorithm has the quadratic worst-case time complexity in terms of the size of the CFG. We implemented ours and the WCC algorithm of Danicic et al. in the Clang/LLVM compiler framework [**?**] and performed experiments on a number of benchmarks selected from SPEC CPU 2017 [**?**]. Our algorithm performs a maximum of 23.8 times and on average 10.6 times faster than the WCC algorithm of Danicic et al. Moreover, the practical performance of our WCC algorithm is closer to the NlogN curve, and thus we improve the theoretical as well as the practical performance of WCC computation by an order of magnitude.

Outline The remainder of this paper is organized as follows. Sec. **??** provides some notations and backgrounds on WCC, Sec. **??** illustrates the changes to be performed in static program slicing due to WCC/SCC, Sec. **??** provides the detailed description of our WCC computation method, prove the correctness, and the worst-case time complexity of our method, Sec. **??** compares the performance of ours and the WCC computation method of Danicic et al. on some practical benchmarks, Sec. **??** discusses the related works, and Sec. **??** concludes the paper.

2 Background

We provide the following formal definition of control flow graph (CFG).

Definition 1 (CFG). *A CFG is a directed graph* (N, E) *where*

1. N *is the set of nodes that includes a* Start *node from where the execution starts, at most one* End *node where the execution terminates normally,* Cond *nodes representing boolean conditions, and* nonCond *nodes; and*
2. $E \subseteq N \times N$ *is the relation describing the possible flow of execution in the graph. An* End *node has no successor, a* Cond *node* n *has at most one true successor and at most one false successor, and all other nodes have at most one successor.*

Like Danicic et al. [**?**], we assume the following:

- The CFG is deterministic. So, any *Cond* node n cannot have multiple true successors and/or multiple false successors.
- We allow *Cond* nodes to have either or both of the successors missing. We may also have non-*End* nodes having no successor (i.e. out-degree zero). An execution that reaches these nodes are silently nonterminating as it is not performing any action and does not return control to the operating system.
- If the CFG G has no *End* nodes, then all executions of G are nonterminating.
- Moreover, if a program has multiple terminating exit points, nodes representing those exit points are connected to the *End* node to model those terminations. Thus, the CFG in Def. **??** is sufficiently general to model a wide-range of real-world intraprocedural programs.

The sets of successor and predecessor nodes of any CFG node n in a CFG (N, E) are denoted by $succ(n)$ and $pred(n)$ where $succ(n) = \{m : (n, m) \in E\}$ and $pred(n) = \{m : (m, n) \in E\}$.

Definition 2 (CFG paths). *A path* π *is a sequence* $n_1, n_2, \ldots, n_k$ *of CFG nodes (denoted by* $[n_1..n_k]$*) such that* $k \geq 1$ *and* $n_{i+1} \in succ(n_i)$ *for all* $1 \leq i \leq k-1$.

A path is *non-trivial* if it contains at least two nodes. We write $\pi - S$ to denote the set of all nodes in the path π that are not in the set S. The length of any path $[n_1..n_k]$ is $k - 1$. A trivial path $[n]$ has path length 0.

Definition 3 (Disjoint paths). *Two finite paths* $[n_1..n_k]$ *and* $[m_1..m_l]$ *such that* $k, l \geq 1$ *in any CFG* G *are disjoint paths if and only if no* n_i *is equal to* m_j *for all* $1 \leq i \leq k$ *and* $1 \leq j \leq l$*. In other words, the paths do not meet at a common vertex.*

Sometimes, we shall use the phrase "two disjoint paths from n" to mean that there exist two paths $n_1 = n, \ldots, n_k$ and $n'_1 = n, \ldots, n'_l$ such that $[n_2..n_k]$ and $[n'_2..n'_l]$ are disjoint paths. In other words, the paths are disjoint after the first common vertex.

Ferrante et al. [?] provided the first formal definition of control dependency relation based on *postdominator* [?] relation. Computing postdominator relations on a CFG G requires that G has a single End node n_e and there is a path from each node n in G to n_e. A node n *postdominates* a node m if and only if every path from m to n_e goes through n. Node n *strictly postdominates* m if n postdominates m and $n \neq m$. The standard *postdominator-based control dependency relation* can then be defined as follows:

Definition 4 (Control Dependency [?,?]). *Node* n *is* control dependent *on node* m *(written* $m \stackrel{cd}{\to} n$*) in the CFG* G *if (1) there exists a nontrivial path* π *in* G *from* m *to* n *such that every node* $m' \in \pi - \{m, n\}$ *is postdominated by* n*, and (2)* m *is not strictly postdominated by* n*.*

The relation $m \stackrel{cd}{\to} n$ implies that there must be two branches of m such that n is always executed in one branch and may not execute in the other branch.

Example 1. The CFG in Fig. **??** that we shall use as a running example is obtained from the perlbench in SPEC CPU2017 [?]. The details of the source code and the labeling of *true* and *false* branches of Cond nodes are omitted for simplicity. The control dependency graph (CDG) is computed from the CFG based on computing postdominator relations such that an edge (n, m) in the CDG represents the control dependency relation $n \stackrel{cd}{\to} m$.

Podgurski and Clarke [?] introduced the weak control dependence which is nontermination sensitive. A number of different nontermination sensitive and nontermination insensitive control dependency relations conservatively extending the standard relation above are defined in successive works [?,?,?,?,?]. Danicic et al. [?] unified all previous definitions and presented two generalizations called *weak* and *strong control closure* which are non-termination insensitive and non-termination sensitive. WCC and SCC capture all the existing non-termination (in)sensitive control dependency relations found in the literature. In this paper, we shall focus mostly on the efficient computation of WCC and occasionally mention SCC. We now recall some relevant definitions and terminologies of WCC from Danicic et al. [?].

Definition 5 (N'-Path). *An* N'*-path is a finite path* $[n_1..n_k]$ *in a CFG* G *such that* $n_k \in N'$ *and* $n_i \notin N'$ *for all* $1 < i \leq k-1$*.*

Note that n_1 may be in N' in the above definition. Thus, an N'-path from n ends at a node in N' and no node in this path are in N' except n_1 which may or may not be in N'.

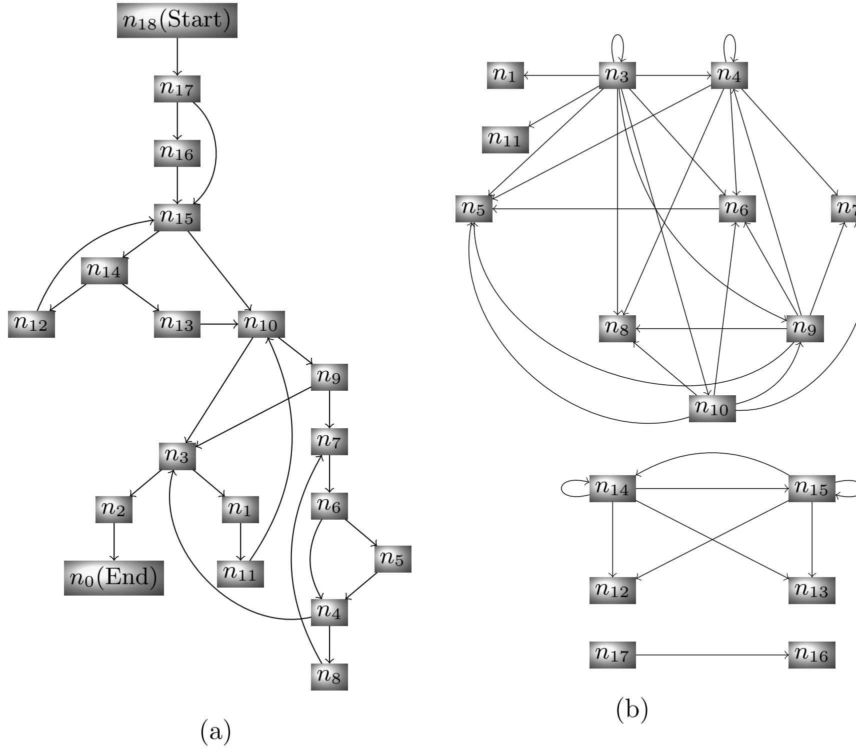

Fig. 1: (a) CFG obtained from a benchmark in SPEC CPU2017 [**?**] (we omit the program instructions for simplicity), (b) control dependency graph computed using postdominator relations.

Definition 6 (N'-weakly committing vertex). *Let $G = (N, E)$ be any CFG. A node $n \in N$ is N'-weakly committing in G if all N'-paths from n have the same endpoint. In other words, there is at most one element of N' that is 'first-reachable' from n.*

Definition 7 (Weak control closure). *Let $G = (N, E)$ be any CFG and let $N' \subseteq N$. N' is weakly control-closed in G if and only if all nodes $n \in N \setminus N'$ that are reachable from N' are N'-weakly committing in G.*

The concept of weakly deciding vertices is introduced to prove that there exists minimal and unique WCC of a set of nodes $N' \subseteq N$. Since program slicing uses control dependence relations to capture all control dependent nodes affecting the slicing criterion, using minimal WCC in program slicing gives us smaller nontermination insensitive slices.

Definition 8 (Weakly deciding vertices). *A node $n \in N$ is N'-weakly deciding in G if and only if there exist two finite proper N'-paths in G that both start at n and have no other common vertices. $WD_G(N')$ denotes the set of all N'-weakly deciding vertices in G.*

Thus, if there exists an N'-weakly deciding vertex n, then n is not N'-weakly committing. The WCC of an arbitrary set $N' \subseteq N$ can be formally defined using weakly deciding vertices as follows:

$$WCC(N') = \{n : n \in WD_G(N'), n \text{ is reachable from } N' \text{ in } G\} \cup N'$$

Example 2. Consider the CFG in Fig. **??**. Let $N' = \{n_5, n_8, n_{10}\}$. The N'-paths in this CFG include $n_9, \ldots, n_5$ and $n_4, \ldots, n_6, n_4, n_8$. The path n_6, n_5, n_4, n_8 is not an N'-path since $n_5 \in N'$. Nodes n_{12}, n_{13}, n_{14} and n_{15} are N'-weakly committing. However, n_9 and n_6 are not N'-weakly committing due to the N'-paths $[n_9..n_{10}]$ and $[n_9..n_5]$, and n_6, n_5 and n_6, n_4, n_8. Nodes n_9 and n_6 are thus N'-weakly deciding and N' is not weakly control closed. However, all N'-weakly deciding vertices n_4, n_6 and n_9 are reachable from N' and thus $N' \cup \{n_4, n_6, n_9\}$ is a weak control-closed set capturing all the relevant control dependencies of N'.

3 Program slicing using WCC and SCC

Program slicing is specified by means of a *slicing criterion* which is usually a set of CFG nodes representing program points of interest. Static backward/forward program slicing then asks to select all program instructions that directly or indirectly affect/ affected by the computation specified in the slicing criterion. Static dependence-based program slicing [**?**,**?**,**?**] is performed by constructing a so-called program dependence graph (PDG) [**?**]. A PDG explicitly represents the data and the control dependence relations in the control flow graph (CFG) of the input program. Any edge $n_1 \rightarrow n_2$ in a PDG represents either the control dependence relation $n_1 \stackrel{cd}{\rightarrow} n_2$ or the data dependence relation $n_1 \stackrel{dd}{\rightarrow} n_2$. The relation $n_1 \stackrel{dd}{\rightarrow} n_2$ holds if n_2 is using the value of a program variable defined at n_1. A PDG is constructed by computing all the data and the control dependence relations in the CFG of a program beforehand, and then include all edges (n, m) in the PDG if $n_1 \stackrel{dd}{\rightarrow} n_2$ or $n_1 \stackrel{cd}{\rightarrow} n_2$ holds. A forward/backward slice includes the set of all reachable nodes in the PDG from the nodes in the slicing criterion in the forward/backward direction.

The existence of the numerous kinds of control dependence in the literature puts us in the dilemma of which control dependence algorithm is to be used to construct PDG. Control dependence computation algorithms such as postdominator-based algorithms exist that cannot compute control dependencies from the following code having no exit point:

if *(p) { L1: x=x+1;* ***goto*** *L2; }* ***else*** *{ L2:* **print**(x)*;* ***goto*** *L1; }*

Building a PDG by using a particular control dependence computation algorithm may miss computing certain kinds of control dependencies, and the program slicing may produce unsound results. With the advent of WCC and SCC, we obtain a more generalized method to compute control closure of a wide-range of programs. However, the above approach of static program slicing is not feasible

Algorithm 1 (Slicing) *Let C be the the slicing criterion, and let $S = C$.*

1. $S' := \bigcup_{n \in S} \{m : m \overset{dd}{\rightarrow}^* n\}$
2. $S := cl(S')$
3. ***if*** $(S = S')$ ***then EXIT***
4. ***else GOTO*** *step 1*

with WCC and SCC. This is due to the fact that even though WCC and SCC capture/compute the weak and strong form of control dependencies that are nontermination (in)sensitive, it is not possible to tell specifically which node is control dependent on which other nodes. Given any set N' of CFG nodes, the weak/strong control closure $cl(N')$ of N' captures all control dependencies $n_1 \overset{cd}{\rightarrow} n_2$ such that $n_2 \in cl(N')$ implies $n_1, n_2 \in cl(N')$. However, by looking into the set $cl(N')$, it is not possible to tell if the relation $n_1 \overset{cd}{\rightarrow} n_2$ holds or not for any $n_1, n_2 \in cl(N')$. Since we cannot compute all individual control dependencies $n_1 \overset{cd}{\rightarrow} n_2$ beforehand, it is not possible to compute a PDG from a CFG using weak or strong control closed sets. However, Alg. **??** can be applied to perform the static program slicing using weak or strong control closures.

The relation $\overset{dd}{\rightarrow}^*$ denotes the transitive-reflexive closure of $\overset{dd}{\rightarrow}$. The above algorithm computes the *slice set* S for backward slicing containing all CFG nodes that affect the computation at the nodes in C. For forward slicing, the relation $\overset{dd}{\rightarrow}^*$ has to be computed in the forward direction. To compute the relation $\overset{dd}{\rightarrow}^*$, we can build a data dependency graph (DDG) capturing only the data dependency relations. Then, step 1 in Algorithm **??** can be accomplished by obtaining the set of all reachable nodes in the DDG from the nodes in S in the forward/backward direction.

Algorithm **??** illustrates that step 2 needs to be performed iteratively until a fixpoint $S = S'$ is reached. Given any CFG (N, E), Danicic et al. provided expensive algorithms to compute weak and strong control closures with worst-case time complexity $O(|N|^3)$ and $O(|N|^4)$ respectively. These algorithms are not only computationally expensive, they cause the static forward/backward program slicing practically inefficient. In the next section, we shall provide an alternative simple yet practically efficient method of computing a minimal weak control closed set.

4 Efficient computation of minimal WCC

The relationship between WCC and weakly deciding vertices are the following (Lemma 51 in [**?**]): the set of CFG nodes $N' \subseteq N$ is weakly control-closed in the CFG $G = (N, E)$ iff all N'-weakly deciding vertices in G that are reachable from N' are in N'. Moreover, $N' \cup WD_G(N')$ is the unique minimal weakly control-closed subset of N that contains N' (Theorem 54 in [**?**]). We perform a simple and efficient two-step process of computing all N'-weakly deciding vertices $WD_G(N')$

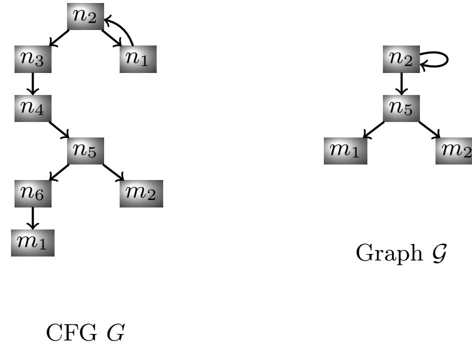

Fig. 2: CFG G used for the informal illustration of our approach. The graph $\mathcal{G}$ is generated by our analysis for the verification of potential N'-weakly deciding vertices.

followed by checking the reachability of these vertices from N' to compute the weakly control-closed subset of N containing N'.

In what follows, let $G = (N, E)$ be a CFG, let $N' \subseteq N$, and let $\mathcal{N}$ be the set of nodes such that $WD_G(N') \cup N' \subseteq \mathcal{N} \subseteq N$. The set of all N'-weakly deciding vertices $WD_G(N')$ are computed in the following two steps:

1. We compute a set of CFG nodes WD which is an overapproximation of the set of all N'-weakly deciding vertices, i.e., $WD_G(N') \subseteq WD$. The WD set includes all CFG nodes n such that n has two disjoint N'-paths. However, WD also contains spurious nodes having overlapping N'-paths or a single N' path which are not N'-weakly deciding. Thus, $\mathcal{N} = WD \cup N'$ is a weakly control-closed subset of N containing N' which is not minimal.
2. For each node $n \in WD$, the above process also indicates all CFG nodes $m \in \mathcal{N}$ such that either $[n..m]$ is an N'-path or there exists an N'-path from n that must go through m. From this information, we build a directed graph $(\mathcal{N}, \mathcal{E})$ such that any edge $(n, m) \in \mathcal{E}$ indicates that n is possibly a weakly deciding vertex, $m \in \mathcal{N}$, and there exists an $\mathcal{N}$-path $[n..m]$ in G. Next, we perform a verification process to check that each node in WD has two disjoint N'-paths using the graph $(\mathcal{N}, \mathcal{E})$ and discard all nodes in WD that do not have two such paths.

4.1 An informal account of our approach

In this section, we give an informal description of our algorithm to compute the N'-weakly deciding vertices. The first step of this algorithm keeps track of all N'-paths (or $\mathcal{N}$-paths to be more specific where $\mathcal{N} = N'$ initially) in the CFG. We traverse the CFG backward from the nodes in N' and record all $\mathcal{N}$-paths at each visited node of the CFG. During this process, we discover all CFG nodes n that have more than one $\mathcal{N}$-paths ending at different CFG nodes, and n is included in WD (and thus $n \in \mathcal{N}$) as it is a potential N'-weakly deciding vertex.

S_1	$T_1 = m_1 \rightarrow n_6 \rightarrow n_5 \rightarrow n_4 \rightarrow n_3 \rightarrow n_2 \rightarrow n_1 \rightarrow n_2$
	$P_1 = [m_1]^0 \quad [n_6, m_1]^1 \quad [n_5..m_1]^2 \quad [n_4..m_1]^3 \quad [n_3..m_1]^4 \quad [n_2..m_1]^5 \quad [n_1..m_1]^6$
	$T_2 = m_2 \rightarrow n_5 \rightarrow n_4 \rightarrow n_3 \rightarrow n_2 \rightarrow n_1 \rightarrow n_2$
	$P_2 = [m_2]^0 \quad [\mathbf{n_5}]^{\mathbf{0}} \quad [n_4, n_5]^1 \quad [n_3..n_5]^2 \quad [\mathbf{n_2}]^{\mathbf{0}} \quad [n_1..n_2]^1$
S_2	$T_3 = m_1 \rightarrow n_6 \rightarrow n_5 \rightarrow n_4$
	$P_3 = [m_1]^0 \quad [n_6, m_1]^1 \quad [n_5..m_1]^2 \quad [n_4..m_1]^3$
	$T_4 = m_2 \rightarrow n_5 \rightarrow n_4 \rightarrow n_3 \rightarrow n_2 \rightarrow n_1 \rightarrow n_2$
	$P_4 = [m_2]^0 \quad [\mathbf{n_5}]^{\mathbf{0}} \quad [n_4, n_5]^1 \quad [n_3..n_5]^2 \quad [n_2..n_5]^3 \quad [n_1..n_5]^4$

Table 1: The $\mathcal{N}$-paths discovered by our algorithm. CFG nodes are visited in two different orders denoted by S_1 and S_2. T_i represents the sequence of visited CFG nodes and P_i represents the sequence of discovered $\mathcal{N}$-paths during the corresponding visits for $1 \leq i \leq 4$. The superscript on a path denotes its length.

In the following, we illustrate this process using the CFG G in Fig. **??** where $m_1, m_2 \in N'$.

We have trivial $\mathcal{N}$-paths $[m_1]$ and $[m_2]$ of lengths zero at CFG nodes m_1 and m_2 respectively. The $\mathcal{N}$-paths from a node are identified from the $\mathcal{N}$-paths of its successor nodes. The trivial $\mathcal{N}$-path $[m_1]$ leads to the $\mathcal{N}$-path $[n_6, m_1]$ of length 1 which in turn leads to $[n_5..m_1]$ of length 2. Similarly, $[m_2]$ leads to the $\mathcal{N}$-path $[n_5, m_2]$ of length 1. Since two $\mathcal{N}$-paths $[n_5..m_1]$ and $[n_5, m_2]$ are identified from n_5, n_5 is included in *WD* and a new trivial $\mathcal{N}$-path $[n_5]$ of length 0 is identified. Different orders of visiting CFG nodes may produce different $\mathcal{N}$-paths.

Table **??** presents two possible orders of visiting the CFG nodes. The sequence of $\mathcal{N}$-paths denoted by P_1 is produced due to visiting the node sequence T_1. Note that an earlier visit to n_2 has produced the $\mathcal{N}$-path $[n_2..m_1]$ of length 5, and the last visit to n_2 from n_1 (via the backward edge) in T_1 does not produce any new $\mathcal{N}$-path at n_2 as it could generate the $\mathcal{N}$-path $[n_2..m_1]$ of length 7 which is not preferred over $[n_2..m_1]$ of length 5 by our algorithm. While visiting the sequence of nodes in T_2, our algorithm identifies two $\mathcal{N}$-paths $[n_5..m_1]$ and $[n_5..m_2]$, and thus it includes n_5 in *WD*. Moreover, a new trivial $\mathcal{N}$-path $[n_5]$ is generated, and the successive visits to the remaining sequence of nodes replace the old $\mathcal{N}$-paths by the newly generated paths of smaller lengths. From the $\mathcal{N}$-paths $[n_3..n_5]$ and $[n_1..m_1]$ at the successor nodes of n_2, our algorithm infers that there exist two $\mathcal{N}$-paths $[n_2..m_1]$ and $[n_2..n_5]$ from n_2, and thus it includes n_2 in *WD* even though no two disjoint $\mathcal{N}$-paths exist in G. Thus, *WD* is an overapproximation of $WD_G(N')$. When CFG nodes are visited according to the order specified in S_2, our algorithm does not infer two $\mathcal{N}$-paths at n_2, and thus it becomes more precise by not including n_2 in *WD*. Note that this order of visiting CFG nodes does not affect the soundness (as we prove it later in this section), but the precision and performance of the first step our analysis, which is a well-known phenomenon in static program analysis. Note that our algorithm does not compute path lengths explicitly in generating $\mathcal{N}$-paths; it is accounted implicitly by our analysis.

The second step of our algorithm generates a graph $\mathcal{G}$ consisting of the set of nodes $N' \cup WD$ and the edges (n, m) such that $n \in WD$, $n' \in succ(n)$, and $[n'..m]$ is the $\mathcal{N}$-path discovered in the first step of the analysis. Thus, $[n..m]$ is an $\mathcal{N}$-path in the CFG. The graph $\mathcal{G}$ in Fig. **??** is generated from the WD set and the $\mathcal{N}$-paths generated due to visiting node sequences T_1 and T_2 in Table **??**. Next, we traverse the graph $\mathcal{G}$ from N' backward; if a node $n \in WD$ is reached, we immediately know one of the $\mathcal{N}$-paths from n and explore the other unvisited branches of n to look for a second disjoint $\mathcal{N}$-path. For the graph $\mathcal{G}$ in Fig. **??**, if $n_5 \in WD$ is reached from m_1, it ensures that $[n_5..m_1]$ is an $\mathcal{N}$-path in the CFG. Next, we look for a second $\mathcal{N}$-path in the other branch of n_5. In this particular case, the immediate successor of n_5 that is not yet visited is $m_2 \in N'$ such that $[n_5..m_2]$ is the second $\mathcal{N}$-path disjoint from $[n_5..m_1]$, which verifies that n_5 is an N'-weakly deciding vertex. We could have that $m_2 \notin N'$, and in that case, we traverse the graph $\mathcal{G}$ from m_2 in the forward direction to look for an $\mathcal{N}$-path different from $[n_5..m_1]$, include n_5 in $WD_G(N')$ if such a path is found, and excluded it from $WD_G(N')$ otherwise. Similarly, we discover the $\mathcal{N}$-path $[n_2..n_5]$ by reaching n_2 from n_5. However, since any $\mathcal{N}$-path from n_2 through the other branch of n_2 overlaps with $[n_2..n_5]$, n_2 is discarded to be a N'-weakly deciding vertex. When all nodes in WD are verified, we obtain the set $WD_G(N') \subseteq WD$ and the algorithm terminates.

4.2 An overapproximation of the weakly deciding vertices

We perform a backward traversal of the CFG from the nodes in N'. Initially, $\mathcal{N} = N'$. We maintain a function $A(n)$ for each CFG node $n \in N$. This function serves the following purposes:

1. If the backward traversal of the CFG visits only one $\mathcal{N}$-path $[n..m]$, then we set $A(n) = m$.
2. If two disjoint $\mathcal{N}$-paths $[n..m_1]$ and $[n..m_2]$ are visited during the backward traversal of the CFG, then we set $A(n) = n$.

We initialize the function $A(n)$ as follows:

$$A(n) = \begin{cases} \bot & n \in N \setminus N' \\ n & n \in N' \end{cases} \tag{1}$$

The valuation $A(n) = \bot$ indicates that no $\mathcal{N}$-path from n is visited yet. If we visit a CFG node $n \in N \setminus N'$ with two N'-paths (which may possibly be not disjoint due to overapproximation), then n is a potential N'-weakly deciding vertex. In this case, we set $A(n) = n$, n is included in WD (and hence $n \in \mathcal{N}$), and the function $A(n)$ will not be changed further.

If $A(n) \neq n$, then $A(n)$ may be modified multiple times during the walk of the CFG. If $A(n) = m_1$ is modified to $A(n) = m_2$ such that $n \neq m_1 \neq m_2$, then there exists a path $n, \ldots, m_2, \ldots, m_1$ in G such that $m_1, m_2 \in \mathcal{N}$ and $[n..m_2]$ is an $\mathcal{N}$-path in G. This may happen when (i) visiting the CFG discovers the $\mathcal{N}$-path $[n..m_1]$ such that $m_2 \notin \mathcal{N}$, and (ii) in a later visit to m_2, m_2 is included

Algorithm 2 (OverapproxWD) ***Input***: $G = (N, E), N', A$, ***Output***: A, WD

1. *Initialization:*
   (a) *Set* $WD = \emptyset$
   (b) *Set the worklist* $W = N'$
2. *Remove an element* n *from* W. ***Forall*** $m \in pred(n)$ ***do*** *the following:*
   (a) *Compute* $S_m = \{A(m') : m' \in succ(m), A(m') \neq \bot\}$
   if $(|S_m| > 1)$ ***then GOTO*** *(b)* ***else GOTO*** *(c)*
   (b) ***if*** $(A(m) \neq m)$ ***then*** *insert* m *into* W, *update* $A(m) = m$, *and add* m *to* WD. ***GOTO*** *(3).*
   (c) ***if*** $(A(m) \neq m$ *and* $x \in S_m)$ ***then*** *(i) obtain* $y = A(m)$, *(ii) update* $A(m) = x$, *and (iii)* ***if*** $(y \neq x)$ ***then*** *insert* m *into* W.
   GOTO *(3).*
3. ***if*** *(*W *is empty)* ***then EXIT else GOTO*** *(2)*

in WD (and in $\mathcal{N}$) that invalidates the path $[n..m_1]$ as an $\mathcal{N}$-path and obtains a new $\mathcal{N}$-path $[n..m_2]$. Note that if $[n..m]$ is an $\mathcal{N}$-path and $m \notin N'$, then there exists an N'-path from n that go through m which we prove later in this section.

Alg. **??** computes the set WD which is an overapproximation of weakly control-closed subset of N containing N'. It uses a worklist W to keep track of which CFG nodes to visit next. Note the following observations for Alg. **??**.

- For any node n in W, $A(n) \neq \bot$ due to the initializations in Eq. **??** and steps 2(b) and 2(c).
- The set S_m in step 2(a) is never empty due to the fact that n is a successor of m and $A(n) \neq \bot$.
- If $A(m) = m$, then m will never be included in W in 2(b) and 2(c) as further processing of node m will not give us any new information.
- Since m can only be included in WD in step (2b) if $A(m) \neq m$, and $A(m) = m$ for any $m \in N'$ due to Eq. **??**, we must have $WD \cap N' = \emptyset$.
- Node m can only be included in W in step 2(c) if $A(m) = x$ is updated to $A(m) = y$ such that $y \neq x$.
- If any path $[n..m]$ is traversed such that $A(m) = m$ and no node in $[n..m] - \{m\}$ is in WD, then m is transferred such that $A(n') = m$ for all $n' \in [n..m] - \{m\}$ due to step (2c). Also, note that if $A(n) = m$, then we must have $A(m) = m$.
- The functions A are both the input and the output of the algorithm. This facilitates computing WD incrementally. This incremental WD computation will improve the performance of client applications of WCC such as program slicing (see Alg. **??**). We leave the study on the impact of incremental WD computation on program slicing as a future work.

Theorems **??** and **??** below state the correctness of Alg. **??** which we prove using an auxiliary lemma.

Lemma 1. *If* $A(n) = m$ *and* $n \neq m$, *then there exists an* N'*-path from* n *and all* N'*-paths from* n *must include* m.

Proof. Since $A(n) = m$, there exists a path $\pi = [n..m]$ visited in Alg. **??** from m backward. The transfer of m to $A(n)$ is only possible if we have $S_x = \{m\}$ for all $x \in \pi - \{m\}$ and $A(x) = m$ is set in step (2c). Since $A(x) \neq x$, no node $x \in \pi - \{m\}$ is in $WD \cup N'$. Also, there exists a predecessor y of m such that $S_y = \{m\}$ which is only possible if $A(m) = m$. Thus, we must have $m \in N' \cup WD$ and π is a $(WD \cup N')$-path.

If $m \in N'$, then the lemma trivially holds. Suppose $m = m_1 \notin N'$. Then, we must have $m_1 \in WD$, and there exists a successor n_1 of m_1 such that $A(n_1) = m_2$. If $m_2 \notin N'$, then $m_2 \in WD$ and there exists a successor n_2 of m_2 such that $A(n_2) = m_3$. Thus, we obtain a subsequence of nodes $n_1, \ldots, n_k$ such that $A(n_i) = m_{i+1}$ for all $1 \leq i \leq k$ and eventually we have $m_{k+1} \in N'$ since the CFG is finite and it is traversed from the nodes in N' backward. Thus, $[n..m_{k+1}]$ is an N'-path which go through m. □

Corollary 1. *If $A(n) = m$, then $m \in N' \cup WD$.*

Proof. The proof follows from the first part of the proof of Lemma **??**.

Theorem 1. *For any WD computed in Alg.* **??**, *$WD_G(N') \subseteq WD$.*

Proof. Suppose the lemma does not hold. So, there exists an N'-weakly deciding vertex $n \in WD_G(N')$ such that $n \notin WD$. Thus, there are two disjoint N'-paths from n. Let $n_1 = n, \ldots, n_k$ and $m_1 = n, \ldots, m_l$ be two N'-paths. Since $n_k, m_l \in N'$, $A(n_k) = n_k$ and $A(m_l) = m_l$ due to Eq. **??**. Alg. **??** traverses these paths and update $A(n_i)$ and $A(m_j)$ in step (2c) such that

$$A(n_i) \neq \bot \text{ and } A(m_j) \neq \bot \text{ for all } 1 \leq i < k \text{ and } 1 \leq j < l.$$

Since $n \notin WD$, $|S_n| \leq 1$ in (2a). Node n has at most two successors according to the definition of CFG (Def. **??**). Since $A(n_2) \neq \bot$ and $A(m_2) \neq \bot$, $|S_n| \leq 1$ is only possible if $A(n_2) = A(m_2)$. Let $A(n_2) = m$. Then, we must have $A(n) = m$ and all N'-paths must include m according to Lemma **??**. Thus, we conclude that n is not an N'-weakly deciding vertex since the N'-paths from n are not disjoint, and we obtain the contradiction. □

Theorem 2. *Alg.* **??** *eventually terminates.*

Proof. Alg. **??** iterates as long as there exist elements in W. For all $n \in N$ such that $A(n) = n$, n is included in WD and it never gets included in W again. If the value of $A(n)$ remains $\bot$, then n is never reached and included in W during the walk of the CFG. Thus the algorithm can only be nonterminating for some node n such that $A(n) \neq n \neq \bot$. According to step (2c) in the algorithm, n can only be included in W if the new value of $A(n)$ is different from the old one. Thus, in order for the algorithm to be nonterminating, there exists an infinite update to $A(n)$ by the sequence of values $m_1, \ldots, m_k, \ldots$ such that no two consecutive values are the same, i.e., $m_i \neq m_{i+1}$ for all $i \geq 1$.

According to Lemma **??**, $A(n) = m_i$ implies that there exists an N'-path from n and all N'-paths from n must include m_i. Thus, there exists a path $[n..m_i]$ in

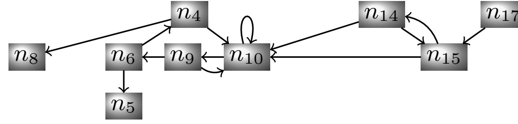

Fig. 3: Graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ constructed according to Def. **??** from the CFG in Fig. **??**, and A and WD in Example **??**

the CFG. If $A(n)$ is updated by m_{i+1}, then $m_{i+1} \in WD$ and $A(m_{i+1}) = m_{i+1}$. Node m_{i+1} must be in the path $[n..m_i]$ as otherwise we eventually have $S_n = \{m_i, m_{i+1}\}$ in (2a) which will lead to $A(n) = n$. So, $A(n)$ will never become m_i again in (2c) as all N'-paths from n must go through $m_{i+1} \in WD$. Similarly, if $A(n)$ is updated by m_{i+2}, m_{i+2} must be in the path $[n..m_{i+1}]$ and $A(n)$ will never be updated by m_{i+1} again. Since the path $[n..m_{i+1}]$ has a finite number of nodes, $A(n)$ cannot be updated infinitely, and the algorithm eventually terminates. □

Example 3. Let $N' = \{n_5, n_8\}$ for the CFG in Fig **??**. Alg. **??** computes A and WD as follows:

- $A(n) = \bot$ for $n \in \{n_0, n_2\}$
- $A(n_i) = n_i$ for $i \in \{4, \ldots, 6, 8, \ldots, 10, 14, 15, 17\}$
- $A(n_i) = n_{10}$ for $i \in \{1, 3, 11, 13\}$
- $A(n_7) = n_6, A(n_{12}) = n_{15}, A(n_{16}) = n_{15}, A(n_{18}) = n_{17}$
- $WD = \{n_4, n_6, n_9, n_{10}, n_{14}, n_{15}, n_{17}\}$

Note that CFG nodes $n_9, n_{10}, n_{14}, n_{15}$, and n_{17} have no disjoint N'-paths as all N'-paths from these nodes must go through n_{10}. Thus, these nodes do not belong to $WD_G(N')$. However, we have $WD_G(N') = \{n_4, n_6\}$ and $WD_G(N') \subseteq WD$ holds.

4.3 Generating minimal weakly deciding vertices

Alg. **??** is sound according to Theorem **??**. However, as illustrated in Sec. **??**, the WD set computed in this algorithm contains spurious nodes that are not N'-weakly deciding. In what follows, we provide a general and efficient algorithm to verify the results obtained from Alg. **??** and discard all incorrectly identified N'-weakly deciding vertices. Thus, both algorithms together provide minimal and sound N'-weakly deciding vertices. We first represent the solutions generated by Alg. **??** as a directed graph $\mathcal{G}$ as follows:

Definition 9. *$\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is a directed graph, where*

- *$\mathcal{N} = N' \cup WD$, and*
- *$\mathcal{E} = \{(n, A(m)) : n \in WD, m \in succ(n), A(m) \neq \bot\}$.*

Note that $succ(n)$ is the set of successors of n in the CFG. In Fig. **??**, $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is constructed from A and WD in Example **??** and the CFG in Fig. **??**. Any graph $\mathcal{G}$ constructed according to Def. **??** has the following properties:

- If there exists an edge (n, m) in $\mathcal{E}$ such that $m \in N'$, then there exists an N'-path $[n..m]$ in the CFG G.
- An edge (n, m) in $\mathcal{E}$ such that $m \in WD$ implies that there exists an N'-path from n going through m (from Lemma **??**).
- There exist no successors of a node in N' since $WD \cap N' = \emptyset$.
- Graph $\mathcal{G}$ may be an edge-disjoint graph since there may exist N'-weakly deciding vertices and their N'-paths do not overlap.
- Since our CFG has at most two successors according to Def. **??**, any node in $\mathcal{G}$ has at most two successors. However, some nodes in $\mathcal{G}$ may have self-loop or only one successor due to the spurious nodes generated in WD. Moreover, $|\mathcal{N}| \leq |N|$, $|\mathcal{E}| \leq |E|$.

The intuitive idea of the verification process is the following. For any $n \in N'$, we consider a predecessor m of n in $\mathcal{G}$. Thus, we know that $[m..n]$ is an N'-path in the CFG G. If there exist another successor $n' \in N'$ of m such that $n \neq n'$, then $[m..n']$ is another N'-path disjoint from $[m..n]$ and m is an N'-weakly deciding vertex. However, all other successors of m might be from WD instead of N'. Let $succ_{\mathcal{G}}(m)$ and $pred_{\mathcal{G}}(m)$ be the sets of successors and predecessors of m in $\mathcal{G}$. Then, we traverse $\mathcal{G}$ from the nodes in $succ_{\mathcal{G}}(m) \setminus N'$ in the forward direction to find an N'-path from m which is disjoint from $[m..n]$. If it visits a node in N' different from n, then m is an N'-weakly deciding vertex due to having two disjoint N'-paths. Otherwise, we exclude m from WD. Most nodes in WD can be immediately verified by looking into their immediate successors without traversing the whole graph $\mathcal{G}$. Also, the graph $\mathcal{G}$ is usually much smaller than the CFG. Thus, the whole verification process is practically very efficient.

Given the graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, Alg. **??** generates WD_{min} which is the set of minimal N'-weakly deciding vertices. Like Alg. **??**, we use a function $\bar{A}(n)$ to keep track of N'-paths visited from n. Initially, $\bar{A}(n) = \perp$ for all $n \in \mathcal{N} \setminus N'$ and $\bar{A}(n) = n$ otherwise. A boolean function $T(n)$ is set to true if $n \in N'$, and $T(n) = false$ otherwise. Another boolean function $V(n)$, which is initially *false*, is set to *true* if n is already verified. The procedure $noDisjointNPath(m, \mathcal{G}, R_n, WD_{min}, N')$ used in the algorithm traverses the graph $\mathcal{G}$ from the nodes in R_n in the forward direction visiting each node at most once. If a node in $N' \cup WD_{min}$ different from m is visited, then it returns *true*, otherwise *false*. During this traversal, no successors of a node in $N' \cup WD_{min}$ are visited as an N'-path must end at a node in N'. We skip providing the details of this procedure since it is a simple graph traversal algorithm. Note that $S_n \neq \emptyset$ in step (3). This is because there exists a successor m of n from which n is reached during the backward traversal of the graph $\mathcal{G}$ and $\bar{A}(m) \neq \perp$.

Theorem **??** below proves that WD_{min} is the minimal weakly control-closed subset of N containing N'.

Theorem 3. *For any WD_{min} computed in Alg. ??, $WD_G(N') = WD_{min}$.*

Proof. "$\subseteq$": Let $n \in WD_G(N')$. According to Theorem **??**, $WD_G(N') \subseteq WD$ and thus $n \in WD$. Suppose $m_1, m_2 \in succ(n)$ since there exist two disjoint N'-paths from n, and also assume that $A(m_i) = n_1^i$ for $i = 1, 2$. Thus, (n, n_1^i) is an

Algorithm 3 (VerifyWDV) ***Input***: $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and N', ***Output***: WD_{min}

1. *Initialization :*
   (a) ***Forall*** $(n \in \mathcal{N} \setminus N')$ ***do***
   $\bar{A}(n) = \bot$, $V(n) = false$, and $T(n) = false$
   (b) ***Forall*** $(n \in N')$ ***do***
   $\bar{A}(n) = n$, $V(n) = true$, and $T(n) = true$
   (c) *Set the worklist* $W = \bigcup_{n \in N'} pred_{\mathcal{G}}(n)$, *and set* $WD_{min} = \emptyset$
2. ***if*** *(W is empty)* ***then EXIT else*** *remove* n *from* W *and set* $V(n) = true$
3. *Compute the following sets:*

$$S_n = \{\bar{A}(m) : m \in succ_{\mathcal{G}}(n), \bar{A}(m) \neq \bot\}$$
$$R_n = \{m : m \in succ_{\mathcal{G}}(n), \bar{A}(m) = \bot\}$$

   Let $m \in S_n$. ***if*** $(|S_n| > 1)$ ***then GOTO*** *(a)* ***else GOTO*** *(b)*
   (a) *Set* $\bar{A}(n) = n$. ***if*** $(T(n) = false)$ ***then*** $WD_{min} = WD_{min} \cup \{n\}$. ***GOTO*** *(4)*
   (b) ***if*** $(noDisjointNPath(m, \mathcal{G}, R_n, WD_{min}, N'))$ ***then*** *set* $\bar{A}(n) = \bar{A}(m)$ *and* ***GOTO*** *(4)* ***else GOTO*** *(a)*
4. ***Forall*** $(n' \in pred_{\mathcal{G}}(n)$ *such that* $V(n') = false)$ ***do***
   $W = W \cup \{n'\}$
   GOTO *(2)*

edge in $\mathcal{G}$ for $i = 1, 2$. According to Corollary **??**, $n_1^i \in N' \cup WD$. If $n_1^i \notin N'$, we can show similarly that there exists a node n_2^i such that (n_1^i, n_2^i) is an edge in $\mathcal{G}$ for some $1 \leq i \leq 2$ and $n_2^i \in N' \cup WD$. Since graph $\mathcal{G}$ and the CFG G are finite, eventually we have the following sequence of edges

$$(n, n_1^1), (n_1^1, n_2^1), \ldots, (n_{k-1}^1, n_k^1) \text{ and } (n, n_1^2), (n_1^2, n_2^2), \ldots, (n_{l-1}^2, n_l^2)$$

such that $n_k^1, n_l^2 \in N'$ for some $k, l \geq 1$. The graph $\mathcal{G}$ is traversed backward from $n_k^i \in N'$ and n will be reached in successive iterations in Alg. **??**. Thus, n is reached by traversing an N'-path $[n..n_k^1]$ backward. Either another N'-path $[n..n_l^2]$ will be discovered immediately during the construction of S_n in step (3) or it will be discovered by calling the procedure *noDisjointNPath* and we eventually have $n \in WD_{min}$.

"$\supseteq$": Let $n \in WD_{min}$. Thus, there exists a node $m \in N'$ such that n is reached during traversing the graph $\mathcal{G}$ backward and thus $[n..m]$ is an N'-path. Also, there exists a successor $m' \neq m$ of n such that either $m' \in N'$ or *noDisjointNPath* procedure traverses an N'-path from m_2 which is disjoint from $[n..m]$.Thus, $n \in WD_G(N')$ due to having two disjoint N'-paths.

4.4 Computing minimal WCC

After obtaining the WD_{min} set containing minimal N'-weakly deciding vertices, computing minimal WCC requires checking the reachability of these nodes from the nodes in N'. Alg. **??** below provides the complete picture of computing minimal WCC.

Algorithm 4 (minimalWCC) ***Input***: $G = (N, E)$ and N', ***Output***: WCC

1. *Apply Eq.* **??** *to initialize* A *and set* $WCC = N'$
2. $(A, WD) = OverapproxWD(G, N')$
3. *Construct* $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ *according to Def.* **??**
4. $WD_{min} = VerifyWDV(\mathcal{G}, N')$
5. *Traverse* G *forward from the nodes in* N' *visiting each node* $n \in N$ *at most once and include* n *to* WCC **if** $n \in WD_{min}$

Example 4. For the graph $\mathcal{G}$ in Fig. **??**(b) and $N' = \{n_5, n_8\}$, Alg. **??** generates $WD_{min} = \{n_4, n_6\}$. Alg. **??** computes $WCC = \{n_4, n_5, n_6, n_8\}$ for the CFG in Fig. **??** and N' as above.

4.5 Worst-case time complexity

Lemma 2. *The worst-case time complexity of Alg.* **??** *is* $O(|N|^2)$.

Proof. The worst-case time complexity is dominated by the costs in step (2) of Alg. **??**. Since $|succ(n)| \leq 2$ for any CFG node n, all the operations in steps (2a)-(2c) have constant complexity. However, after removing a node n from W, all the predecessors of n are visited. If the CFG G has no N'-weakly deciding vertices, then Alg. **??** visits at most $|N|$ nodes and $|E|$ edges after which the operation $y \neq x$ in (2c) is always false, no node will be inserted in W, and thus the cost will be $O(|N|+|E|)$. In order to obtain a vertex in *WD*, it needs to visit at most $|N|$ nodes and $|E|$ edges and the maximum cost will be $O(|N| + |E|)$. If a node n is included in *WD*, then we set $A(n) = n$ and n will never be included in W afterwards due to the first conditional instruction in step (2c). Since we can have at most $|N|$ N'-weakly deciding vertices, the total worst-case cost will be $O((|N|+|E|)*|N|)$. Since any CFG node has at most two successors, $O(|E|) = O(|N|)$, and thus the worst-case time complexity is $O(|N|^2)$.

Lemma 3. *The worst-case time complexity of Alg.* **??** *is* $O(|N|^2)$.

Proof. The initialization steps in Alg. **??** have the worst-case cost $O(|\mathcal{N}|)$. The worst-case cost of Alg. **??** is dominated by the main loop in steps (2)-(4). This main loop iterates at most $|\mathcal{N}|$ times since (i) once an element is removed from W, it is marked as visited and never inserted into W again, and (ii) the loop iterates as long as there are elements in W. Computing the sets S_n and R_n have constant costs since $|succ_{\mathcal{G}}(n)| \leq 2$. The costs of all other operations in step (3) are also constant except the *noDisjointNPath* procedure which has the worst-case cost of $O(|\mathcal{N}|+|\mathcal{E}|)$ as it is a simple forward graph traversal algorithm visiting each node and edge at most once and other operations have constant cost. Step (4) visits the edges in $\mathcal{E}$ to insert elements in W and cannot visit more than $|\mathcal{E}|$ edges. Thus, the dominating cost of Alg. **??** is $O((|\mathcal{N}| + |\mathcal{E}|) \times |\mathcal{N}|)$. Since $|\mathcal{N}| \leq N$, $|\mathcal{E}| \leq E$, and $O(|N|) = O(|E|)$, $O(|N|^2)$ is the worst-case time complexity of this algorithm.

Theorem 4. *The worst-case time complexity of Alg.* **??** *is* $O(|N|^2)$.

Proof. The worst-case time complexity of Alg. **??** is dominated by the $VerifyWDV$ and *OverapproxWD* procedures which have the worst-case time complexity $O(|N|^2)$ according to Lemma **??** and **??**.

#	benchmarks	KLOC	#Proc	T_{wcc}	T_{wccD}	Speedup
1	Mcf	3	40	9.6	56.7	5.9
2	Nab	24	327	55.1	418.6	7.6
3	Xz	33	465	40.5	116.5	2.9
4	X264	96	1449	155.7	896.0	5.8
5	Imagick	259	2586	334.8	2268.9	6.8
6	Perlbench	362	2460	1523.3	32134.8	21.1
7	GCC	1304	17827	26658.1	634413.9	23.8
	Average Speedup = 10.6					

Table 2: Experimental results on selected benchmarks from SPEC CPU 2017 [?]

5 Experimental evaluation

We implemented ours and the weak control closure algorithms of Danicic et al. [?] in the Clang/LLVM compiler framework [?] and run experiments in an Intel(R) Core(TM) i7-7567U CPU with 3.50GHz. The experiments are performed on a number of benchmarks consisting of approximately 2081 KLOC written in C language.

Table **??** shows experimental results performed on seven benchmarks selected from the SPEC CPU 2017 [?]. The #Proc column indicates the number of procedures analyzed in the respective benchmarks, T_{wcc} and T_{wccD} columns show total runtime of the algorithms of ours and Danicic et al., and the Speedup column indicates the speedup of our approach over Danicic et al. which is calculated as T_{wccD}/T_{wcc}. Each procedure is analyzed 10 times and the N'-sets are chosen randomly for each run. All times are recorded in microseconds which are converted to milliseconds and the analysis times reported in Table **??** are the average of 10 runs.

Regarding the correctness, both algorithms compute the same weakly control closed sets. As shown in Table **??**, we obtain the highest and the lowest speedup of 23.8 and 2.9 from the *GCC* and the *Xz* benchmarks, and an average speedup from all benchmarks is 10.6. The *Xz* benchmark provides the lowest speedup due to the fact that it has fewer procedures than *GCC* and the sizes of the CFGs for most procedures in this benchmark are very small; the average size of a CFG (i.e. number of CFG nodes) is only 8 per procedure. On the other hand, *GCC* has 38 times more procedures than *Xz* and the average size of a CFG per procedure is 20. Also, the greater speedups are obtained in larger CFGs. There are 171 and 55 procedures in *GCC* with the size of the CFGs greater than 200 and 500 respectively and the maximum CFG size is 15912, whereas the maximum CFG size in *Xz* is 87. For benchmarks like *Mcf* and *Nab*, even though they have fewer procedures than *Xz*, the average CFG size per procedure in these benchmarks are 21 and 16.

Since Alg. **??** and **??** dominates the computational complexity of computing WCC, we compare the execution times of these algorithms in Fig. **??**. We also plotted the functions $NlogN$ and N^2 to compare the execution curves of the algorithms with these functions. All times are measured in microseconds and an average of 10 runs. If there exist several CFGs with the same size, we keep the

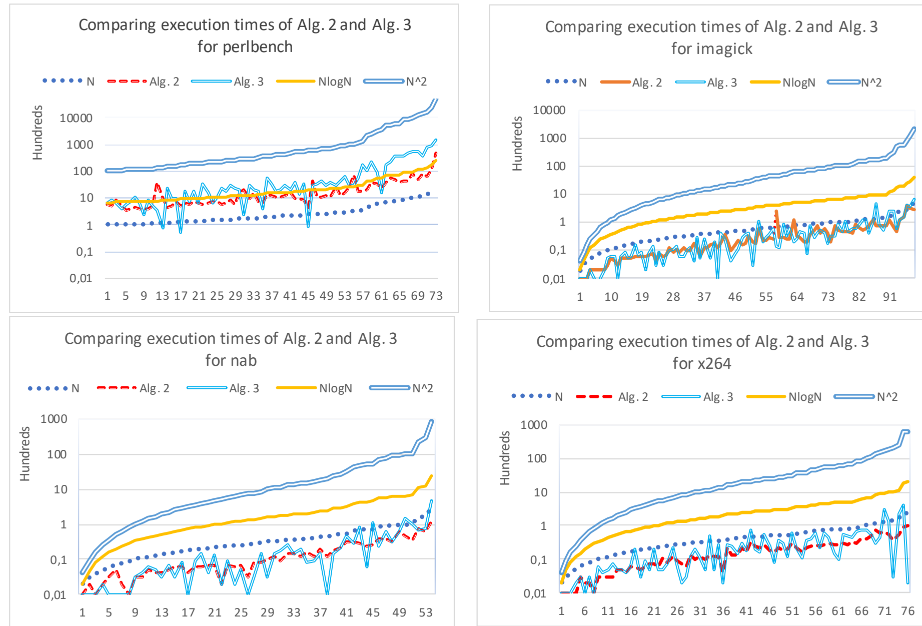

Fig. 4: Comparing execution times of Alg. **??** and **??**. Execution time curves are also compared with the $NlogN$ and N^2 functions where N represents the number of nodes in the CFG. X-axis represents selected CFGs from the respective benchmarks. Y-axis represents either the execution times of the algorithms measured in microseconds or the value of $NlogN$ and N^2. All charts are displayed in the logarithmic scale.

execution time of only one of them. As illustrated in the figure, Alg. **??** performs consistently. However, the performance of Alg. **??** varies above or below the performance of Alg. **??**. This due to the fact that it shows optimal performance when *noDisjointNPath* procedure is called minimally. The performance curves of both algorithms are closer to the $NlogN$ curve for perlbench benchmark and closer to the linear curve for other benchmarks depicted in Fig. **??** when the times are measured in microseconds. In the appendix, we provide execution curves of other benchmarks.

We also have evaluated our algorithms by performing the same experiments on a virtual machine (VM) running on the real machine as specified above. The virtual machine uses a 64-bit Ubuntu OS with 10 GB RAM having 2 cores and the real machine runs Mac OS Version 10.15.4 with 16GB RAM. Due to randomization, the experiments have different N' sets. We obtain a maximum speedup of 12 for *Perlbench* and an average speedup of 5.7 on all benchmarks from the experiments on the VM. Even though we obtain a smaller speedup compared to the speedup on the real machine, our algorithm is still several times faster than the WCC computation of Danicic et al., and we obtain similar perfor-

mance curves for all benchmarks on VM. Evidently, our algorithm improves the state-of-the-art computation of weak control closure by an order of magnitude.

6 Related Work

Denning and Denning [?] are the pioneers to use dominator-based approach to identify program instructions influenced by the conditional instructions in the context of information-flow security. Weiser [?], the pioneer and prominent author in program slicing, used their approach in program slicing. However, the first formal definition of control dependence is provided by Ferrante et al. [?] in developing the program dependence graph (PDG) which is being used for program slicing and program optimization. This definition became standard afterward and is being used for over two decades.

Podgurski and Clarke [?] provided two control dependence relations called weak and strong syntactic dependence. The strong control dependence corresponds to the standard control dependence relation. The weak control dependence subsumes the strong control dependence relation in the sense that any strong control dependence relation is also a weak control dependence. Moreover, the weak control dependence relation is nontermination sensitive. Bilardi and Pingali [?] provided a generalized framework for the standard and the weak control dependence relation of Podgurski and Clarke by means of the dominance relation parameterized with respect to a set of CFG paths. Different classes of CFG path set provides different control dependence relations.

Ranganath et al. [?,?] considered CFGs possibly having multiple end nodes or no end node. These kinds of CFGs originate from programs containing modern program instructions like exceptions or nonterminating constructs often found in web services or distributed systems. They also considered low-level code such as JVM producing irreducible CFGs, and defined a number of control dependency relations that are nontermination (in)sensitive and conservatively extend the standard control dependency relation. The worst-case time complexity of the algorithms for computing their control dependences is $O(|N|^4 log|N|)$ where $|N|$ is the number of vertices of the CFG.

The control dependence relations defined later are progressively generalized than the earlier definitions, but one may be baffled by the overwhelming number of such definitions, e.g. in [?], to choose the right one. Danicic et al. [?] unified all previously defined control dependence relations and provided the most generalized non-termination insensitive and nontermination sensitive control dependence called weak and strong control-closure. These definitions are based on the weak and strong projections which are the underlying semantics for control dependence developed by the authors. These semantics are opposite to that of Podgurski and Clark in the sense that Danicic et al.'s weak (resp. strong) relation is similar to Podgurski and Clark's strong (resp. weak) relation. The worst-case time complexity of their weak and strong control closure algorithms are $O(|N|^3)$ and $O(|N|^4)$ where $|N|$ is the number of vertices of the CFG. Léchenet et al. [?] provided automated proof of correctness in the Coq proof assistant for the weak

control closure algorithm of Danicic et al. and presented an efficient algorithm to compute such control closure. The efficiency of their method is demonstrated by experimental evaluation. However, no complexity analysis of their algorithm is provided.

Khanfar et al. [?] developed an algorithm to compute all direct control dependencies to a particular program statement for using it in demand-driven slicing. Their method only works for programs that must have a unique exit point. Neither the computational complexity nor the practical performance benefits of their algorithm are stated. On the other hand, we compute minimal weak control closure for programs that do not have such restrictions. Our method improves the theoretical computational complexity of computing weak control closure than the state-of-the-art methods, and it is also practically efficient.

7 Conclusion and future work

Danicic et al. provided two generalizations called weak and strong control closure (WCC and SCC) that subsume all existing nontermination insensitive and nontermination sensitive control dependency relations. However, their algorithms to compute these relations have cubic and quartic worst-case complexity in terms of the size of the CFG which is not acceptable for client applications of WCC and/or SCC such as program slicing. In this paper, we have developed an efficient and easy to understand method of computing minimal WCC. We provided the theoretical correctness of our method. Our WCC computation method has the quadratic worst-case time complexity in terms of the size of the CFG. We experimentally evaluated the algorithms for computing WCC of ours and Danicic et al. on practical benchmarks and obtained the highest 23.8 and on average 10.6 speedups compared to the state-of-the-art method. The performance of our WCC algorithm for practical applications is closer to either NlogN or linear curve in most cases when time is measured in microseconds. Thus we improve the practical performance of WCC computation by an order of magnitude.

We have applied our algorithms of computing minimal weakly deciding vertices in the computation of strongly control closed sets, implemented ours and the state-of-the-art SCC method in the Clang/LLVM framework, and evaluated these algorithms on practical benchmarks. We also obtained similar speedups in computing SCC. However, we have not included our SCC computation method in this paper due to space limitations.

As regards future work, our algorithm to compute minimal weakly deciding vertices can be applied to compute minimal SSA programs. Recently, Masud and Ciccozzi [?,?] showed that the standard dominance frontier-based SSA construction method increases the size of the SSA program by computing a significant amount of unnecessary ϕ functions. However, they provided complex and expensive algorithms that can generate minimal SSA programs. Our algorithm can be adapted to get an efficient alternative method in computing minimal SSA programs. Another future direction would be to compute WCC and SCC for interprocedural programs.

Acknowledgment

This research is supported by the Knowledge Foundation through the HERO project.

References

1. Amtoft, T.: Correctness of practical slicing for modern program structures. Tech. rep., Department of Computing and Information Sciences, Kansas State University (2007)
2. Bilardi, G., Pingali, K.: A framework for generalized control dependence. SIGPLAN Not. **31**(5), 291–300 (May 1996). https://doi.org/10.1145/249069.231435
3. Bucek, J., Lange, K.D., v. Kistowski, J.: Spec cpu2017: Next-generation compute benchmark. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering. pp. 41–42. ICPE '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3185768.3185771
4. Danicic, S., Barraclough, R., Harman, M., Howroyd, J.D., Kiss, Á., Laurence, M.: A unifying theory of control dependence and its application to arbitrary program structures. Theoretical Computer Science **412**(49), 6809–6842 (2011)
5. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (Jul 1977). https://doi.org/10.1145/359636.359712
6. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (Jul 1987). https://doi.org/10.1145/24039.24041
7. Khanfar, H., Lisper, B., Masud, A.N.: Static backward program slicing for safety-critical systems. In: de la Puente, J.A., Vardanega, T. (eds.) Reliable Software Technologies - Ada-Europe 2015 - 20th Ada-Europe International Conference on Reliable Software Technologies, Madrid Spain, June 22-26, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9111, pp. 50–65. Springer (2015). https://doi.org/10.1007/978-3-319-19584-1_4
8. Khanfar, H., Lisper, B., Mubeen, S.: Demand-driven static backward slicing for unstructured programs. Tech. rep. (May 2019), http://www.es.mdh.se/publications/5511-
9. Lattner, C., Adve, V.: The LLVM Compiler Framework and Infrastructure Tutorial. In: LCPC'04 Mini Workshop on Compiler Research Infrastructures. West Lafayette, Indiana (Sep 2004)
10. Léchenet, J.C., Kosmatov, N., Le Gall, P.: Fast computation of arbitrary control dependencies. In: Russo, A., Schürr, A. (eds.) Fundamental Approaches to Software Engineering. pp. 207–224. Springer International Publishing, Cham (2018)
11. Lisper, B., Masud, A.N., Khanfar, H.: Static backward demand-driven slicing. In: Asai, K., Sagonas, K. (eds.) Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015. pp. 115–126. ACM (2015). https://doi.org/10.1145/2678015.2682538
12. Masud, A.N., Ciccozzi, F.: Towards constructing the SSA form using reaching definitions over dominance frontiers. In: 19th International Working Conference on Source Code Analysis and Manipulation, SCAM 2019, Cleveland, OH, USA, September 30 - October 1, 2019. pp. 23–33. IEEE (2019). https://doi.org/10.1109/SCAM.2019.00012

13. Masud, A.N., Ciccozzi, F.: More precise construction of static single assignment programs using reaching definitions. Journal of Systems and Software **166**, 110590 (2020). https://doi.org/10.1016/j.jss.2020.110590
14. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. SIGSOFT Softw. Eng. Notes **9**(3), 177–184 (Apr 1984). https://doi.org/10.1145/390010.808263
15. Pingali, K., Bilardi, G.: Optimal control dependence computation and the roman chariots problem. ACM Trans. Program. Lang. Syst. **19**(3), 462–491 (May 1997)
16. Podgurski, A., Clarke, L.A.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Softw. Eng. **16**(9), 965–979 (Sep 1990)
17. Prosser, R.T.: Applications of boolean matrices to the analysis of flow diagrams. In: Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference. pp. 133–138. IRE-AIEE-ACM '59 (Eastern), ACM, New York, NY, USA (1959)
18. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. In: European Symposium on Programming. LNCS, vol. 3444, pp. 77–93. Springer-Verlag (2005)
19. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst. **29**(5) (Aug 2007)
20. Weiser, M.: Program slicing. In: Proc. 5th International Conference on Software Engineering. pp. 439–449. ICSE '81, IEEE Press, Piscataway, NJ, USA (1981), http://dl.acm.org/citation.cfm?id=800078.802557

A Appendix

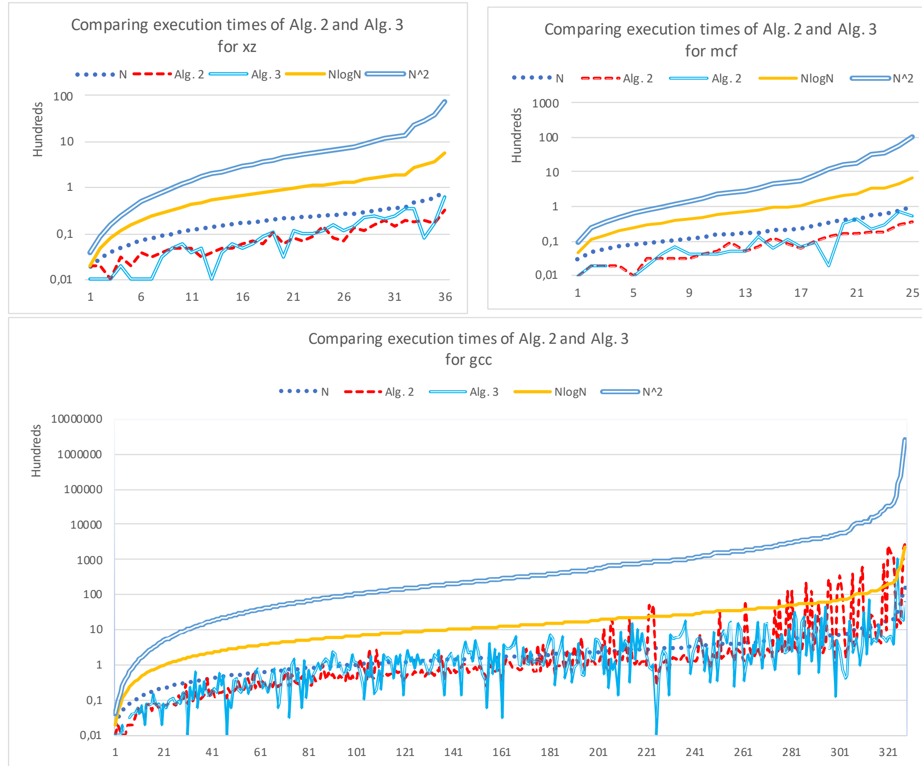

Fig. 5: Comparing execution times of Alg. **??** and **??**. Execution time curves are also compared with the $NlogN$ and N^2 functions where N represents the number of nodes in the CFG. X-axis represents selected CFGs from the respective benchmarks. Y-axis represents either the execution times of the algorithms measured in microseconds or the value of $NlogN$ and N^2. All charts are displayed in the logarithmic scale.