Memory-Efficient Fixpoint Computation

Sung Kook Kim[1](✉), Arnaud J. Venet[2], and Aditya V. Thakur[1]

[1] University of California, Davis, CA 95616, USA
{sklkim,avthakur}@ucdavis.edu
[2] Facebook, Inc., Menlo Park, CA 94025, USA
ajv@fb.com

Abstract. Practical adoption of static analysis often requires trading precision for performance. This paper focuses on improving the memory efficiency of abstract interpretation without sacrificing precision or time efficiency. Computationally, abstract interpretation reduces the problem of inferring program invariants to computing a fixpoint of a set of equations. This paper presents a method to minimize the memory footprint in Bourdoncle's iteration strategy, a widely-used technique for fixpoint computation. Our technique is agnostic to the abstract domain used. We prove that our technique is optimal (i.e., it results in minimum memory footprint) for Bourdoncle's iteration strategy while computing the same result. We evaluate the efficacy of our technique by implementing it in a tool called Mikos, which extends the state-of-the-art abstract interpreter IKOS. When verifying user-provided assertions, Mikos shows a decrease in peak-memory usage to 4.07% (24.57×) on average compared to IKOS. When performing interprocedural buffer-overflow analysis, Mikos shows a decrease in peak-memory usage to 43.7% (2.29×) on average compared to IKOS.

1 Introduction

Abstract interpretation [14] is a general framework for expressing static analysis of programs. Program invariants inferred by an abstract interpreter are used in client applications such as program verifiers, program optimizers, and bug finders. To extract the invariants, an abstract interpreter computes a fixpoint of an equation system approximating the program semantics. The efficiency and precision of the abstract interpreter depends on the *iteration strategy*, which specifies the order in which the equations are applied during fixpoint computation.

The *recursive iteration strategy* developed by Bourdoncle [10] is widely used for fixpoint computation in academic and industrial abstract interpreters such as NASA IKOS [11], Crab [32], Facebook SPARTA [16], Kestrel Technology CodeHawk [48], and Facebook Infer [12]. Extensions to Bourdoncle's approach that improve precision [1] and time efficiency [26] have also been proposed.

This paper focuses on improving the memory efficiency of abstract interpretation. This is an important problem in practice because large memory requirements can prevent clients such as compilers and developer tools from using

D. Pichardie and M. Sighireanu (Eds.): SAS 2020, LNCS 12389, pp. 35–64, 2020.
https://doi.org/10.1007/978-3-030-65474-0_3

sophisticated analyses. This has motivated approaches for efficient implementations of abstract domains [4,25,44], including techniques that trade precision for efficiency [5,17,24].

This paper presents a technique for memory-efficient fixpoint computation. Our technique minimizes the memory footprint in Bourdoncle's recursive iteration strategy. Our approach is agnostic to the abstract domain and does not sacrifice time efficiency. We prove that our technique exhibits optimal peak-memory usage for the recursive iteration strategy while computing the same fixpoint (Sect. 3). Specifically, our approach does not change the iteration order but provides a mechanism for early deallocation of abstract values. Thus, there is no loss of precision when improving memory performance. Furthermore, such "backward compatibility" ensures that existing implementations of Bourdoncle's approach can be replaced without impacting clients of the abstract interpreter, an important requirement in practice.

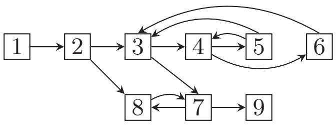


Fig. 1. Control-flow graph G_1

Suppose we are tasked with proving assertions at program points 4 and 9 of the control-flow graph $G_1(V, \rightarrow)$ in Fig. 1. Current approaches (Sect. 2.1) allocate abstract values for each program point during fixpoint computation, check the assertions at 4 and 9 after fixpoint computation, and then deallocate all abstract values. In contrast, our approach deallocates abstract values and checks the assertions during fixpoint computation while guaranteeing that the results of the checks remain the same and that the peak-memory usage is optimal.

We prove that our approach deallocates abstract values as soon as they are no longer needed during fixpoint computation. Providing this theoretical guarantee is challenging for arbitrary irreducible graphs such as G_1. For example, assuming that node 8 is analyzed after 3, one might think that the fixpoint iterator can deallocate the abstract value at 2 once it analyzes 8. However, 8 is part of the strongly-connected component $\{7, 8\}$, and the fixpoint iterator might need to iterate over node 8 multiple times. Thus, deallocating the abstract value at 2 when node 8 is first analyzed will lead to incorrect results. In this case, the earliest that the abstract value at 2 can be deallocated is after the stabilization of component $\{7, 8\}$.

Furthermore, we prove that our approach performs the assertion checks as early as possible during fixpoint computation. Once the assertions are checked, the associated abstract values are deallocated. For example, consider the assertion check at node 4. Notice that 4 is part of the strongly-connected components $\{4, 5\}$ and $\{3, 4, 5, 6\}$. Checking the assertion the first time node 4 is analyzed could lead to an incorrect result because the abstract value at 4 has not converged. The earliest that the check at node 4 can be executed is after the convergence of the component $\{3, 4, 5, 6\}$. Apart from being able to deallocate abstract values earlier, early assertion checks provide partial results on timeout.

The key theoretical result (Theorem 1) is that our iteration strategy is memory-optimal (i.e., it results in minimum memory footprint) while computing the same result as Bourdoncle's approach. Furthermore, we present an almost-linear time algorithm to compute this optimal iteration strategy (Sect. 4).

We have implemented this memory-optimal fixpoint computation in a tool called Mikos (Sect. 5), which extends the state-of-the-art abstract interpreter for C/C++, IKOS [11]. We compared the memory efficiency of Mikos and IKOS on the following tasks:

T1 Verifying user-provided assertions. Task T1 represents the program-verification client of a fixpoint computation. We performed interprocedural analysis of 784 SV-COMP 2019 benchmarks [6] using reduced product of Difference Bound Matrix with variable packing [17] and congruence [20] domains.

T2 Proving absence of buffer overflows. Task T2 represents the bug-finding and compiler-optimization client of fixpoint computation. In the context of bug finding, a potential buffer overflow can be reported to the user as a potential bug. In the context of compiler optimization, code to check buffer-access safety can be elided if the buffer access is verified to be safe. We performed interprocedural buffer overflow analysis of 426 open-source programs using the interval abstract domain.

On Task T1, Mikos shows a decrease in peak-memory usage to 4.07% (24.57×) on average compared to IKOS. For instance, peak-memory required to analyze the SV-COMP 2019 benchmark `ldv-3.16-rc1/205_9a-net-rtl8187` decreased from 46 GB to 56 MB. Also, while `ldv-3.14/usb-mxl111sf` spaced out in IKOS with 64 GB memory limit, peak-memory usage was 21 GB for Mikos. On Task T2, Mikos shows a decrease in peak-memory usage to 43.7% (2.29×) on average compared to IKOS. For instance, peak-memory required to analyze a benchmark `ssh-keygen` decreased from 30 GB to 1 GB.

The contributions of the paper are as follows:

- A memory-optimal technique for Bourdoncle's recursive iteration strategy that does not sacrifice precision or time efficiency (Sect. 3).
- An almost-linear time algorithm to construct our memory-efficient iteration strategy (Sect. 4).
- Mikos, an interprocedural implementation of our approach (Sect. 5).
- An empirical evaluation of the efficacy of Mikos using a large set of C benchmarks (Sect. 6).

Sect. 2 presents necessary background on fixpoint computation, including Bourdoncle's approach; Sect. 7 presents related work; Sect. 8 concludes.

2 Fixpoint Computation Preliminaries

This section presents background on fixpoint computation that will allow us to clearly state the problem addressed in this paper (Sect. 2.3). This section is not meant to capture all possible approaches to implementing abstract interpretation. However, it does capture the relevant high-level structure of abstract-interpretation implementations such as IKOS [11].

Consider an equation system Φ whose dependency graph is $G(V, \rightarrow)$. The graph G typically reflects the control-flow graph of the program, though this is not always true. The aim is to find the fixpoint of the equation system Φ:

$$\begin{aligned} \textsc{Pre}[v] &= \bigsqcup \{\textsc{Post}[p] \mid p \rightarrow v\} && v \in V \\ \textsc{Post}[v] &= \tau_v(\textsc{Pre}[v]) && v \in V \end{aligned} \tag{1}$$

The maps $\textsc{Pre}\colon V \rightarrow \mathcal{A}$ and $\textsc{Post}\colon V \rightarrow \mathcal{A}$ maintain the abstract values at the beginning and end of each program point, where $\mathcal{A}$ is an abstract domain. The abstract transformer $\tau_v\colon \mathcal{A} \rightarrow \mathcal{A}$ overapproximates the semantics of program point $v \in V$. After fixpoint computation, $\textsc{Pre}[v]$ is an invariant for $v \in V$.

Client applications of the abstract interpreter typically query these fixpoint values to perform assertion checks, program optimizations, or report bugs. Let $V_C \subseteq V$ be the set of program points where such checks are performed, and let $\varphi_v\colon \mathcal{A} \rightarrow \mathit{bool}$ represent the corresponding functions that performs the check for each $v \in V_C$. To simplify presentation, we assume that the check function merely returns `true` or `false`. Thus, after fixpoint computation, the client application computes $\varphi_v(\textsc{Pre}[v])$ for each $v \in V_C$.

The exact least solution of the system Eq. 1 can be computed using Kleene iteration provided $\mathcal{A}$ is Noetherian. However, most interesting abstract domains require the use of *widening* ($\triangledown$) to ensure termination followed by *narrowing* to improve the post solution. In this paper, we use "fixpoint" to refer to such an approximation of the least fixpoint. Furthermore, for simplicity of presentation, we restrict our description to a simple widening strategy. However, our implementation (Sect. 5) uses more sophisticated widening and narrowing strategies implemented in state-of-the-art abstract interpreters [1,11].

An *iteration strategy* specifies the order in which the individual equations are applied, where widening is used, and how convergence of the equation system is checked. For clarity of exposition, we introduce a *Fixpoint Machine (`FM`)* consisting of an imperative set of instructions. An `FM` program represents a particular iteration strategy used for fixpoint computation. The syntax of Fixpoint Machine programs is defined by the following grammar:

$$\mathit{Prog} ::= \texttt{exec}\ v \mid \texttt{repeat}\ v\ \texttt{[}\mathit{Prog}\texttt{]} \mid \mathit{Prog} \fatsemi \mathit{Prog}\ , v \in V \tag{2}$$

Informally, the instruction `exec` v applies τ_v for $v \in V$; the instruction `repeat` v `[`P_1`]` repeatedly executes the `FM` program P_1 until convergence and performs widening at v; and the instruction $P_1 \fatsemi P_2$ executes `FM` programs P_1 and P_2 in sequence.

The syntax (Eq. 2) and semantics (Fig. 2) of the Fixpoint Machine are sufficient to express Bourdoncle's recursive iteration strategy (Sect. 2.1), a widely-used approach for fixpoint computation [10]. We also extend the notion of iteration strategy to perform memory management of the abstract values as well as perform checks during fixpoint computation (Sect. 2.2).

2.1 Bourdoncle's Recursive Iteration Strategy

In this section, we review Bourdoncle's recursive iteration strategy [10] and show how to generate the corresponding `FM` program.

Bourdoncle's iteration strategy relies on the notion of *weak topological ordering (WTO)* of a directed graph $G(V, \rightarrow)$. A WTO is defined using the notion of a *hierarchical total ordering (HTO)* of a set.

Definition 1. *A* hierarchical total ordering $\mathcal{H}$ *of a set* S *is a well parenthesized permutation of* S *without two consecutive "(".* ∎

An HTO $\mathcal{H}$ is a string over the alphabet S augmented with left and right parenthesis. Alternatively, we can denote an HTO $\mathcal{H}$ by the tuple $(S, \preceq, \omega)$, where $\preceq$ is the total order induced by $\mathcal{H}$ over the elements of S and $\omega \colon V \rightarrow 2^V$. The elements between two matching parentheses are called a *component*, and the first element of a component is called the *head*. Given $l \in S$, $\omega(l)$ is the set of heads of the components containing l. We use $\mathcal{C} \colon V \rightarrow 2^V$ to denote the mapping from a head to its component.

Example 1. Let $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. An example HTO $\mathcal{H}_1(V, \preceq, \omega)$ is `1 2 (3 (4 5) 6) (7 8) 9`. $\omega(3) = \{3\}$, $\omega(5) = \{3, 4\}$, and $\omega(1) = \emptyset$. It has components $\mathcal{C}(4) = \{4, 5\}$, $\mathcal{C}(7) = \{7, 8\}$ and $\mathcal{C}(3) = \{3, 6\} \cup \mathcal{C}(4)$. ∎

A weak topological ordering (WTO) $\mathcal{W}$ of a directed graph $G(V, \rightarrow)$ is an HTO $\mathcal{H}(V, \preceq, \omega)$ satisfying certain constraints listed below:

Definition 2. *A* weak topological ordering $\mathcal{W}(V, \preceq, \omega)$ *of a directed graph* $G(V, \rightarrow)$ *is an HTO* $\mathcal{H}(V, \preceq, \omega)$ *such that for every edge* $u \rightarrow v$*, either (i)* $u \prec v$*, or (ii)* $v \preceq u$ *and* $v \in \omega(u)$*.* ∎

Example 2. HTO $\mathcal{H}_1$ in Example 1 is a WTO $\mathcal{W}_1$ of the graph G_1 (Fig. 1). ∎

Given a directed graph $G(V, \rightarrow)$ that represents the dependency graph of the equation system, Bourdoncle's approach uses a WTO $\mathcal{W}(V, \preceq, \omega)$ of G to derive the following *recursive iteration strategy*:

- The total order $\preceq$ determines the order in which the equations are applied. The equation after a component is applied only after the component stabilizes.
- The stabilization of a component $\mathcal{C}(h)$ is determined by checking the stabilization of the head h.
- Widening is performed at each of the heads.

We now show how the WTO can be represented using the syntax of our Fixpoint Machine (`FM`) defined in Eq. 2. The following function `genProg`: WTO $\rightarrow$ *Prog* maps a given WTO $\mathcal{W}$ to an `FM` program:

$$\texttt{genProg}(\mathcal{W}) := \begin{cases} \texttt{repeat}\ v\ \texttt{[genProg}(\mathcal{W}')\texttt{]} & \text{if } \mathcal{W} = (v\ \mathcal{W}') \\ \texttt{genProg}(\mathcal{W}_1) \fatsemi \texttt{genProg}(\mathcal{W}_2) & \text{if } \mathcal{W} = \mathcal{W}_1\ \mathcal{W}_2 \\ \texttt{exec}\ v & \text{if } \mathcal{W} = v \end{cases} \tag{3}$$

Each node $v \in V$ is mapped to a single `FM` instruction by `genProg`; we use `Inst`$[v]$ to refer to this `FM` instruction corresponding to v. Note that if $v \in V$ is a head, then `Inst`$[v]$ is an instruction of the form `repeat` v `[...]`, else `Inst`$[v]$ is `exec` v.

Example 3. The WTO $\mathcal{W}_1$ of graph G_1 (Fig. 1) is `1 2 (3 (4 5) 6) (7 8) 9`. The corresponding `FM` program is P_1 = `genProg`$(\mathcal{W}_1)$ = `exec` 1 ⨾ `exec` 2 ⨾ `repeat` 3 `[repeat` 4 `[exec` 5`]` ⨾ `exec` 6`]` ⨾ `repeat` 7 `[exec` 8`]` ⨾ `exec` 9. The colors used for brackets and parentheses are to more clearly indicate the correspondence between the WTO and the `FM` program. Note that `Inst`$[1]$ = `exec` 1, and `Inst`$[4]$ = `repeat` 4 `[exec` 5`]`. ■

Ignoring the text in gray, the semantics of the `FM` instructions shown in Fig. 2 capture Bourdoncle's recursive iteration strategy. The semantics are parameterized by the graph $G(V, \rightarrow)$ and a WTO $\mathcal{W}(V, \preceq, \omega)$.

2.2 Memory Management During Fixpoint Computation

In this paper, we extend the notion of iteration strategy to indicate when abstract values are deallocated and when checks are executed. The gray text in Fig. 2 shows the semantics of the `FM` instructions that handle these issues. The right-hand side of $\Rightarrow$ is executed if the left-hand side evaluates to true. Recall that the set $V_C \subseteq V$ is the set of program points that have assertion checks. The map $\textsc{Ck}\colon V_C \rightarrow$ `bool` records the result of executing the check $\varphi_u(\textsc{Pre}[u])$ for each $u \in V_C$. Thus, the *output of the `FM` program* is the map $\textsc{Ck}$. In practice, the functions φ_u are expensive to compute. Furthermore, they often write the result to a database or report the output to a user. Consequently, we assume that only the first execution of φ_u is recorded in $\textsc{Ck}$.

The *memory configuration* $\mathcal{M}$ is a tuple $(\textsc{Dpost}, \textsc{Achk}, \textsc{Dpost}^\ell, \textsc{Dpre}^\ell)$ where

- The map $\textsc{Dpost}\colon V \rightarrow V$ controls the deallocation of values in $\textsc{Post}$ that have no further use. If $v = \textsc{Dpost}[u]$, $\textsc{Post}[u]$ is deallocated after the execution of `Inst`$[v]$.
- The map $\textsc{Achk}\colon V_C \rightarrow V$ controls when the check function φ_u corresponding to $u \in V_C$ is executed, after which the corresponding $\textsc{Pre}$ value is deallocated. If $\textsc{Achk}[u] = v$, assertions in u are checked and $\textsc{Pre}[u]$ is subsequently deallocated after the execution of `Inst`$[v]$.

```
$G(V, \rightarrow)$, WTO $\mathcal{W}(V, \preceq, \omega)$,
$V_C \subseteq V$, memory configuration $\mathcal{M}(\textsc{Dpost}, \textsc{Achk}, \textsc{Dpost}^{\ell}, \textsc{Dpre}^{\ell})$

$[\![\texttt{exec } v]\!]_{\mathcal{M}} \overset{\text{def}}{=}$ $\textsc{Pre}[v] \leftarrow \bigsqcup\{\textsc{Post}[p] \mid p \rightarrow v\}$
        $\textbf{foreach } u \in V : v = \textsc{Dpost}[u] \Rightarrow \textbf{free } \textsc{Post}[u]$
        $\textsc{Post}[v] \leftarrow \tau_v(\textsc{Pre}[v])$
        $v \notin V_C \Rightarrow \textbf{free } \textsc{Pre}[v]$
        $\textbf{foreach } u \in V_C : v = \textsc{Achk}[u] \Rightarrow \textsc{Ck}[u] \leftarrow \varphi_u(\textsc{Pre}[u]);$
                $\textbf{free } \textsc{Pre}[u]$

$[\![\texttt{repeat } v \texttt{ [}P\texttt{]}]\!]_{\mathcal{M}} \overset{\text{def}}{=}$ $tpre \leftarrow \bigsqcup\{\textsc{Post}[p] \mid p \rightarrow v \wedge v \notin \omega(p)\}$    } Preamble
        $\textbf{do } \{$
            $\textbf{foreach } u \in V : v \in \textsc{Dpost}^{\ell}[u] \Rightarrow \textbf{free } \textsc{Post}[u]$
            $\textbf{foreach } u \in V_C : v \in \textsc{Dpre}^{\ell}[u] \Rightarrow \textbf{free } \textsc{Pre}[u]$
            $\textsc{Pre}[v], \textsc{Post}[v] \leftarrow tpre, \tau_v(tpre)$    } Loop
            $[\![P]\!]_{\mathcal{M}}$
            $tpre \leftarrow \textsc{Pre}[v] \triangledown \bigsqcup\{\textsc{Post}[p] \mid p \rightarrow v\}$
        $\}$ $\textbf{while}(tpre \not\sqsubseteq \textsc{Pre}[v])$
        $\textbf{foreach } u \in V : v = \textsc{Dpost}[u] \Rightarrow \textbf{free } \textsc{Post}[u]$
        $v \notin V_C \Rightarrow \textbf{free } \textsc{Pre}[v]$    } Postamble
        $\textbf{foreach } u \in V_C : v = \textsc{Achk}[u] \Rightarrow \textsc{Ck}[u] \leftarrow \varphi_u(\textsc{Pre}[u]);$
                $\textbf{free } \textsc{Pre}[u]$

$[\![P_1 \fatsemi P_2]\!]_{\mathcal{M}} \overset{\text{def}}{=}$ $[\![P_1]\!]_{\mathcal{M}}$
        $[\![P_2]\!]_{\mathcal{M}}$
```

Fig. 2. The semantics of the Fixpoint Machine (`FM`) instructions of Eq. 2.

- The map $\textsc{Dpost}^{\ell} : V \rightarrow 2^V$ control deallocation of $\textsc{Post}$ values that are recomputed and overwritten in the loop of a `repeat` instruction before its next use. If $v \in \textsc{Dpost}^{\ell}[u]$, $\textsc{Post}[u]$ is deallocated in the loop of $\texttt{Inst}[v]$.
- The map $\textsc{Dpre}^{\ell} : V_C \rightarrow 2^V$ control deallocation of $\textsc{Pre}$ values that recomputed and overwritten in the loop of a `repeat` instruction before its next use. If $v \in \textsc{Dpre}^{\ell}[u]$, $\textsc{Pre}[u]$ is deallocated in the loop of $\texttt{Inst}[v]$.

To simplify presentation, the semantics in Fig. 2 does not make explicit the allocations of abstract values: if a $\textsc{Post}$ or $\textsc{Pre}$ value that has been deallocated is accessed, then it is allocated and initialized to $\bot$.

2.3 Problem Statement

Two memory configurations are *equivalent* if they result in the same values for each check in the program:

Definition 3. *Given an* FM *program P, memory configuration $\mathcal{M}_1$ is* equivalent *to $\mathcal{M}_2$, denoted by $[\![P]\!]_{\mathcal{M}_1} = [\![P]\!]_{\mathcal{M}_2}$, iff for all $u \in V_C$, we have $\textsc{Ck}_1[u] = \textsc{Ck}_2[u]$, where $\textsc{Ck}_1$ and $\textsc{Ck}_2$ are the check maps corresponding to execution of P using $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively.* ■

The *default memory configuration* $\mathcal{M}_{dflt}$ performs checks and deallocations at the end of the FM program after fixpoint has been computed.

Definition 4. *Given an* FM *program P, the* default memory configuration *$\mathcal{M}_{dflt}(\textsc{Dpost}_{dflt}, \textsc{Achk}_{dflt}, \textsc{Dpost}^{\ell}{}_{dflt}, \textsc{Dpre}^{\ell}{}_{dflt})$ is $\textsc{Dpost}_{dflt}[v] = z$ for all $v \in V$, $\textsc{Achk}_{dflt}[c] = z$ for all $c \in V_C$, and $\textsc{Dpost}^{\ell}{}_{dflt} = \textsc{Dpre}^{\ell}{}_{dflt} = \emptyset$, where z is the last instruction in P.* ■

Example 4. Consider the FM program P_1 from Example 3. Let $V_C = \{4, 9\}$. $\textsc{Dpost}_{\text{dflt}}[v] = 9$ for all $v \in V$. That is, all $\textsc{Post}$ values are deallocated at the end of the fixpoint computation. Also, $\textsc{Achk}_{\text{dflt}}[4] = \textsc{Achk}_{\text{dflt}}[9] = 9$, meaning that assertion checks also happen at the end. $\textsc{Dpost}^{\ell}{}_{\text{dflt}} = \textsc{Dpre}^{\ell}{}_{\text{dflt}} = \emptyset$, so the FM program does not clear abstract values whose values will be recomputed and overwritten in a loop of `repeat` instruction. ■

Given an FM program P, a memory configuration $\mathcal{M}$ is *valid* for P iff it is equivalent to the default configuration; i.e., $[\![P]\!]_{\mathcal{M}} = [\![P]\!]_{\mathcal{M}_{\text{dflt}}}$.

Furthermore, a valid memory configuration $\mathcal{M}$ is *optimal* for a given FM program iff memory footprint of $[\![P]\!]_{\mathcal{M}}$ is smaller than or equal to that of $[\![P]\!]_{\mathcal{M}'}$ for all valid memory configuration $\mathcal{M}'$. The problem addressed in this paper can be stated as:

Given an FM program P, find an optimal memory configuration $\mathcal{M}$.

An optimal configuration should deallocate abstract values during fixpoint computation as soon as they are no longer needed. The challenge is ensuring that the memory configuration remains valid even without knowing the number of loop iterations for `repeat` instructions. Sect. 3 gives the optimal memory configuration for the FM program P_1 from Example 3.

3 Declarative Specification of Optimal Memory Configuration $\mathcal{M}_{\text{opt}}$

This section provides a declarative specification of an optimal memory configuration $\mathcal{M}_{\text{opt}}(\textsc{Dpost}_{\text{opt}}, \textsc{Achk}_{\text{opt}}, \textsc{Dpost}^{\ell}{}_{\text{opt}}, \textsc{Dpre}^{\ell}{}_{\text{opt}})$. The proofs of the theorems in this section can be found in Appendix A. Sect. 4 presents an efficient algorithm for computing $\mathcal{M}_{\text{opt}}$.

Definition 5. *Given a WTO $\mathcal{W}(V, \preceq, \omega)$ of a graph $G(V, \rightarrow)$, the* nesting relation *N is a tuple $(V, \preceq_{\mathsf{N}})$ where $x \preceq_{\mathsf{N}} y$ iff $x = y$ or $y \in \omega(x)$ for $x, y \in V$.* ■

Let $\llbracket v \rceil_{\preceq_{\mathsf{N}}} \overset{\text{def}}{=} \{w \in V \mid v \preceq_{\mathsf{N}} w\}$; that is, $\llbracket v \rceil_{\preceq_{\mathsf{N}}}$ equals the set containing v and the heads of components in the WTO that contain v. The nesting relation $\mathsf{N}(V, \preceq_{\mathsf{N}})$ is a *forest*; i.e. a partial order such that for all $v \in V$, $(\llbracket v \rceil_{\preceq_{\mathsf{N}}}, \preceq_{\mathsf{N}})$ is a chain (Theorem 4, Appendix A.1).

Example 5. For the WTO $\mathcal{W}_1$ of G_1 in Example 2, $\mathsf{N}_1(V, \preceq_{\mathsf{N}})$ is:

$$\begin{tikzcd}
1 & 2 & 3 \arrow[d, dash] \arrow[dr, dash] & & 7 \arrow[d, dash] & 9 \\
& & 4 \arrow[d, dash] & 6 & 8 & \\
& & 5 & & &
\end{tikzcd}.$$

Note that $\llbracket 5 \rceil_{\preceq_{\mathsf{N}}} = \{5, 4, 3\}$, forming a chain $5 \preceq_{\mathsf{N}} 4 \preceq_{\mathsf{N}} 3$. ■

3.1 Declarative Specification of $\textsc{Dpost}_{\text{opt}}$

$\textsc{Dpost}_{\text{opt}}[u] = v$ implies that v is the earliest instruction at which $\textsc{Post}[u]$ can be deallocated while ensuring that there are no subsequents reads of $\textsc{Post}[u]$ during fixpoint computation. We cannot conclude $\textsc{Dpost}_{\text{opt}}[u] = v$ from a dependency $u \rightarrow v$ as illustrated in the following example.

Example 6. Consider the `FM` program P_1 from Example 3, whose graph $G_1(V, \rightarrow)$ is in Fig. 1. Although $2 \rightarrow 8$, memory configuration with $\textsc{Dpost}[2] = 8$ is not valid: $\textsc{Post}[2]$ is read by `Inst`[8], which is executed repeatedly as part of `Inst`[7]; if $\textsc{Dpost}[2] = 8$, $\textsc{Post}[2]$ is deallocated the first time `Inst`[8] is executed, and subsequent executions of `Inst`[8] will read $\perp$ as the value of $\textsc{Post}[2]$. ■

In general, for a dependency $u \rightarrow v$, we must find the head of maximal component that contains v but not u as the candidate for $\textsc{Dpost}_{\text{opt}}[u]$. By choosing the head of *maximal* component, we remove the possibility of having a larger component whose head's `repeat` instruction can execute `Inst`[v] after deallocating $\textsc{Post}[u]$. If there is no component that contains v but not u, we simply use v as the candidate. The following `Lift` operator gives us the candidate of $\textsc{Dpost}_{\text{opt}}[u]$ for $u \rightarrow v$:

$$\texttt{Lift}(u, v) \overset{\text{def}}{=} \max_{\preceq_{\mathsf{N}}}((\llbracket v \rceil_{\preceq_{\mathsf{N}}} \setminus \llbracket u \rceil_{\preceq_{\mathsf{N}}}) \cup \{v\}) \tag{4}$$

$\llbracket v \rceil_{\preceq_{\mathsf{N}}}$ gives us v and the heads of components that contain v. Subtracting $\llbracket u \rceil_{\preceq_{\mathsf{N}}}$ removes the heads of components that also contain u. We put back v to account for the case when there is no component containing v but not u and $\llbracket v \rceil_{\preceq_{\mathsf{N}}} \setminus \llbracket u \rceil_{\preceq_{\mathsf{N}}}$ is empty. Because $\mathsf{N}(V, \preceq_{\mathsf{N}})$ is a forest, $\llbracket v \rceil_{\preceq_{\mathsf{N}}}$ and $\llbracket u \rceil_{\preceq_{\mathsf{N}}}$ are chains, and hence, $\llbracket v \rceil_{\preceq_{\mathsf{N}}} \setminus \llbracket u \rceil_{\preceq_{\mathsf{N}}}$ is also a chain. Therefore, maximum is well-defined.

Example 7. Consider the nesting relation $\mathsf{N}_1(V, \preceq_{\mathsf{N}})$ from Example 5. $\texttt{Lift}(2, 8) = \max_{\preceq_{\mathsf{N}}}((\{8, 7\} \setminus \{2\}) \cup \{8\}) = 7$. We see that 7 is the head of the maximal component containing 8 but not 2. Also, $\texttt{Lift}(5, 4) = \max_{\preceq_{\mathsf{N}}}((\{4, 3\} \setminus \{5, 4, 3\}) \cup \{4\}) = 4$. There is no component that contains 4 but not 5. ■

For each instruction u, we now need to find the last instruction from among the candidates computed using `Lift`. Notice that deallocations of $\textsc{Post}$ values are at a postamble of `repeat` instructions in Fig. 2. Therefore, we cannot use the total order $\preceq$ of a WTO to find the last instruction: $\preceq$ is the order in which the instruction begin executing, or the order in which *preamble*s are executed.

Example 8. Let $\textsc{Dpost}_{to}[u] \overset{\text{def}}{=} \max_{\preceq}\{\texttt{Lift}(u,v) \mid u \rightarrow v\}, u \in V$, an incorrect variant of $\textsc{Dpost}_{\text{opt}}$ that uses the total order $\preceq$. Consider the FM program P_1 from Example 3, whose graph $G_1(V, \rightarrow)$ is in Fig. 1 and nesting relation $\mathsf{N}_1(V, \preceq_{\mathsf{N}})$ is in Example 5. $\textsc{Post}[5]$ has dependencies $5 \rightarrow 4$ and $5 \rightarrow 3$. $\texttt{Lift}(5,4) = 4$, $\texttt{Lift}(5,3) = 3$. Now, $\textsc{Dpost}_{to}[5] = 4$ because $3 \preceq 4$. However, a memory configuration with $\textsc{Dpost}[5] = 4$ is not valid: $\texttt{Inst}[4]$ is nested in $\texttt{Inst}[3]$. Due to the deletion of $\textsc{Post}[5]$ in $\texttt{Inst}[4]$, $\texttt{Inst}[3]$ will read $\perp$ as the value of $\textsc{Post}[5]$. ■

To find the order in which the instructions finish executing, or the order in which *postambles* are executed, we define the relation $(V, \leq)$, using the total order $(V, \preceq)$ and the nesting relation $(V, \preceq_{\mathsf{N}})$:

$$x \leq y \overset{\text{def}}{=} x \preceq_{\mathsf{N}} y \vee (y \not\preceq_{\mathsf{N}} x \wedge x \preceq y) \tag{5}$$

In the definition of $\leq$, the nesting relation $\preceq_{\mathsf{N}}$ takes precedence over $\preceq$. $(V, \leq)$ is a total order (Theorem 5, Appendix A.1). Intuitively, the total order $\leq$ moves the heads in the WTO to their corresponding closing parentheses ')'.

Example 9. For G_1 (Fig. 1) and its WTO $\mathcal{W}_1$, `1 2 (3 (4 5) 6) (7 8) 9`, we have $1 \leq 2 \leq 5 \leq 4 \leq 6 \leq 3 \leq 8 \leq 7 \leq 9$. Note that $3 \preceq 6$ while $6 \leq 3$. Postamble of `repeat` 3 `[...]` is executed after $\texttt{Inst}[6]$, while preamble of `repeat` 3 `[...]` is executed before $\texttt{Inst}[6]$. ■

We can now define $\textsc{Dpost}_{\text{opt}}$. Given a nesting relation $\mathsf{N}(V, \preceq_{\mathsf{N}})$ for the graph $G(V, \rightarrow)$, $\textsc{Dpost}_{\text{opt}}$ is defined as:

$$\textsc{Dpost}_{\text{opt}}[u] \overset{\text{def}}{=} \max_{\leq}\{\texttt{Lift}(u,v) \mid u \rightarrow v\}, u \in V \tag{6}$$

Example 10. Consider the FM program P_1 from Example 3, whose graph $G_1(V, \rightarrow)$ is in Fig. 1 and nesting relation $\mathsf{N}_1(V, \preceq_{\mathsf{N}})$ is in Example 5. An optimal memory configuration $\mathcal{M}_{\text{opt}}$ defined by Eq. 6 is:

$$\textsc{Dpost}_{\text{opt}}[1] = 2,\ \textsc{Dpost}_{\text{opt}}[2] = \textsc{Dpost}_{\text{opt}}[3] = \textsc{Dpost}_{\text{opt}}[8] = 7,\ \textsc{Dpost}_{\text{opt}}[4] = 6,$$
$$\textsc{Dpost}_{\text{opt}}[5] = \textsc{Dpost}_{\text{opt}}[6] = 3,\ \textsc{Dpost}_{\text{opt}}[7] = \textsc{Dpost}_{\text{opt}}[9] = 9.$$

Successors of u are first lifted to compute $\textsc{Dpost}_{\text{opt}}[u]$. For example, to compute $\textsc{Dpost}_{\text{opt}}[2]$, 2's successors, 3 and 8, are lifted to $\texttt{Lift}(2,3) = 3$ and $\texttt{Lift}(2,8) = 7$. To compute $\textsc{Dpost}_{\text{opt}}[5]$, 5's successors, 3 and 4, are lifted to $\texttt{Lift}(5,3) = 3$ and $\texttt{Lift}(5,4) = 4$. Then, the maximum (as per the total order $\leq$) of the lifted successors is chosen as $\textsc{Dpost}_{\text{opt}}[u]$. Because $3 \leq 7$, $\textsc{Dpost}_{\text{opt}}[2] = 7$. Thus, $\textsc{Post}[2]$ is deleted in $\texttt{Inst}[7]$. Also, because $4 \leq 3$, $\textsc{Dpost}_{\text{opt}}[5] = 3$, and $\textsc{Post}[5]$ is deleted in $\texttt{Inst}[3]$. ■

3.2 Declarative Specification of $\textsc{Achk}_{\textbf{opt}}$

$\textsc{Achk}_{\mathrm{opt}}[u] = v$ implies that v is the earliest instruction at which the assertion check at $u \in V_C$ can be executed so that the invariant passed to the assertion check function φ_u is the same as when using $\mathcal{M}_{\mathrm{dflt}}$. Thus, guaranteeing the same check result $\textsc{Ck}$.

Because an instruction can be executed multiple times in a loop, we cannot simply execute the assertion checks right after the instruction, as illustrated by the following example.

Example 11. Consider the `FM` program P_1 from Example 3. Let $V_C = \{4, 9\}$. A memory configuration with $\textsc{Achk}[4] = 4$ is not valid: $\texttt{Inst}[4]$ is executed repeatedly as part of $\texttt{Inst}[3]$, and the first value of $\textsc{Pre}[4]$ may not be the final invariant. Consequently, executing $\varphi_4(\textsc{Pre}[4])$ in $\texttt{Inst}[4]$ may not give the same result as executing it in $\texttt{Inst}[9]$ ($\textsc{Achk}_{\mathrm{dflt}}[4] = 9$). ■

In general, because we cannot know the number of iterations of the loop in a `repeat` instruction, we must wait for the convergence of the maximal component that contains the assertion check. After the maximal component converges, the `FM` program never visits the component again, making $\textsc{Pre}$ values of the elements inside the component final. Only if the element is not in any component can its assertion check be executed right after its instruction.

Given a nesting relation $\mathsf{N}(V, \preceq_{\mathsf{N}})$ for the graph $G(V, \rightarrow)$, $\textsc{Achk}_{\mathrm{opt}}$ is defined as:

$$\textsc{Achk}_{\mathrm{opt}}[u] \overset{\mathrm{def}}{=} \max\nolimits_{\preceq_{\mathsf{N}}} \llbracket u \rceil_{\preceq_{\mathsf{N}}}, u \in V_C \tag{7}$$

Because $\mathsf{N}(V, \preceq_{\mathsf{N}})$ is a forest, $(\llbracket u \rceil_{\preceq_{\mathsf{N}}}, \preceq_{\mathsf{N}})$ is a chain. Hence, $\max_{\preceq_{\mathsf{N}}}$ is well-defined.

Example 12. Consider the `FM` program P_1 from Example 3, whose graph $G_1(V, \rightarrow)$ is in Fig. 1 and nesting relation $\mathsf{N}_1(V, \preceq_{\mathsf{N}})$ is in Example 5. Suppose that $V_C = \{4, 9\}$. $\textsc{Achk}_{\mathrm{opt}}[4] = \max_{\preceq_{\mathsf{N}}}\{4, 3\} = 3$ and $\textsc{Achk}_{\mathrm{opt}}[9] = \max_{\preceq_{\mathsf{N}}}\{9\} = 9$. ■

3.3 Declarative Specification of $\textsc{Dpost}^{\ell}{}_{\textbf{opt}}$

$v \in \textsc{Dpost}^{\ell}[u]$ implies that $\textsc{Post}[u]$ can be deallocated at v because it is recomputed and overwritten in the loop of a `repeat` instruction before a subsequent use of $\textsc{Post}[u]$.

$\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[u]$ must be a subset of $\llbracket u \rceil_{\preceq_{\mathsf{N}}}$: only the instructions of the heads of components that contain v recompute $\textsc{Post}[u]$. We can further rule out the instruction of the heads of components that contain $\textsc{Dpost}_{\mathrm{opt}}[u]$, because $\texttt{Inst}[\textsc{Dpost}_{\mathrm{opt}}[u]]$ deletes $\textsc{Post}[u]$. We add back $\textsc{Dpost}_{\mathrm{opt}}[u]$ to $\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}$ when u is contained in $\textsc{Dpost}_{\mathrm{opt}}[u]$, because deallocation by $\textsc{Dpost}_{\mathrm{opt}}$ happens after the deallocation by $\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}$.

Given a nesting relation $\mathsf{N}(V, \preceq_{\mathsf{N}})$ for the graph $G(V, \rightarrow)$, $\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}$ is defined as:

$$\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[u] \overset{\mathrm{def}}{=} (\llbracket u \rceil_{\preceq_{\mathsf{N}}} \setminus \llbracket d \rceil_{\preceq_{\mathsf{N}}}) \cup \llparenthesis u \preceq_{\mathsf{N}} d \mathrel{?} \{d\} \mathrel{\text{⸬}} \emptyset \rrparenthesis , u \in V \tag{8}$$

where $d = \textsc{Dpost}_{\mathrm{opt}}[u]$ as defined in Eq. 6, and $(\!\!\mid \mathtt{b} \mathbin{?} \mathtt{x} \mathbin{:} \mathtt{y} \mid\!\!)$ is the ternary conditional choice operator.

Example 13. Consider the FM program P_1 from Example 3, whose graph $G_1(V, \rightarrow)$ is in Fig. 1, nesting relation $\mathsf{N}_1(V, \preceq_{\mathsf{N}})$ is in Example 5, and $\textsc{Dpost}_{\mathrm{opt}}$ is in Example 10.

$$\begin{aligned}
&\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[1] = \{1\},\ \textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[2] = \{2\},\ \textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[3] = \{3\},\\
&\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[4] = \{4\},\ \textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[5] = \{3, 4, 5\},\ \textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[6] = \{3, 6\},\\
&\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[7] = \{7\},\ \textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[8] = \{7, 8\},\ \textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[9] = \{9\}.
\end{aligned}$$

For 7, $\textsc{Dpost}_{\mathrm{opt}}[7] = 9$. Because $7 \not\preceq_{\mathsf{N}} 9$, $\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[7] = \llbracket 7 \rceil_{\preceq_{\mathsf{N}}} \setminus \llbracket 9 \rceil_{\preceq_{\mathsf{N}}} = \{7\}$. Therefore, $\textsc{Post}[7]$ is deleted in each iteration of the loop of $\mathtt{Inst}[7]$. While $\mathtt{Inst}[9]$ reads $\textsc{Post}[7]$ in the future, the particular values of $\textsc{Post}[7]$ that are deleted by $\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[7]$ are not used in $\mathtt{Inst}[9]$. For 5, $\textsc{Dpost}_{\mathrm{opt}}[5] = 3$. Because $5 \preceq_{\mathsf{N}} 3$, $\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}[5] = \llbracket 5 \rceil_{\preceq_{\mathsf{N}}} \setminus \llbracket 3 \rceil_{\preceq_{\mathsf{N}}} \cup \{3\} = \{5, 4, 3\}$. ■

3.4 Declarative Specification of $\textsc{Dpre}^{\ell}{}_{\mathrm{opt}}$

$v \in \textsc{Dpre}^{\ell}[u]$ implies that $\textsc{Pre}[u]$ can be deallocated at v because it is recomputed and overwritten in the loop of a `repeat` instruction before a subsequent use of $\textsc{Pre}[u]$.

$\textsc{Dpre}^{\ell}{}_{\mathrm{opt}}[u]$ must be a subset of $\llbracket u \rceil_{\preceq_{\mathsf{N}}}$: only the instructions of the heads of components that contain v recompute $\textsc{Pre}[u]$. If $\mathtt{Inst}[u]$ is a `repeat` instruction, $\textsc{Pre}[u]$ is required to perform widening. Therefore, u must not be contained in $\textsc{Dpre}^{\ell}{}_{\mathrm{opt}}[u]$.

Example 14. Consider the FM program P_1 from Example 3. Let $V_C = \{4, 9\}$. A memory configuration with $\textsc{Dpre}^{\ell}[4] = \{3, 4\}$ is not valid, because $\mathtt{Inst}[4]$ would read $\perp$ as the value of $\textsc{Post}[4]$ when performing widening. ■

Given a nesting relation $\mathsf{N}(V, \preceq_{\mathsf{N}})$ for the graph $G(V, \rightarrow)$, $\textsc{Dpre}^{\ell}{}_{\mathrm{opt}}$ is defined as:

$$\textsc{Dpre}^{\ell}{}_{\mathrm{opt}}[u] \overset{\mathrm{def}}{=} \llbracket u \rceil_{\preceq_{\mathsf{N}}} \setminus \{u\}\ , u \in V_C \tag{9}$$

Example 15. Consider the FM program P_1 from Example 3, whose graph $G_1(V, \rightarrow)$ is in Fig. 1 and nesting relation $\mathsf{N}_1(V, \preceq_{\mathsf{N}})$ is in Example 5. Let $V_C = \{4, 9\}$. $\textsc{Dpre}^{\ell}{}_{\mathrm{opt}}[4] = \{4, 3\} \setminus \{4\} = \{3\}$ and $\textsc{Dpre}^{\ell}{}_{\mathrm{opt}}[9] = \{9\} \setminus \{9\} = \emptyset$. Therefore, $\textsc{Pre}[4]$ is deleted in each loop iteration of $\mathtt{Inst}[3]$. ■

The following theorem is proved in Appendix A.2:

Theorem 1. *The memory configuration* $\mathcal{M}_{\mathrm{opt}}(\textsc{Dpost}_{\mathrm{opt}}, \textsc{Achk}_{\mathrm{opt}}, \textsc{Dpost}^{\ell}{}_{\mathrm{opt}}, \textsc{Dpre}^{\ell}{}_{\mathrm{opt}})$ *is optimal.*

```
Algorithm 1: GenerateFMProgram($G$)
Input: Directed graph $G(V, \rightarrow)$
Output: FM program $pgm$, $\mathcal{M}_{\text{opt}}(\textsc{Dpost}_{\text{opt}}, \textsc{Achk}_{\text{opt}}, \textsc{Dpost}^{\ell}_{\text{opt}}, \textsc{Dpre}^{\ell}_{\text{opt}})$
1   D := DepthFirstForest($G$)
2   $\rightarrow_{\mathtt{B}}$ := back edges in D
3   $\rightarrow_{\mathtt{CF}}$ := cross & forward edges in D
4   $\rightarrow' := \rightarrow \setminus \rightarrow_{\mathtt{B}}$
5   for $v \in V$ do rep($v$) := $v$; R[$v$] := $\emptyset$
6   P := $\emptyset$
7   removeAllCrossFwdEdges()
8   for $h \in V$ in descending $DFN_{\mathtt{D}}$ do
9       restoreCrossFwdEdges($h$)
10      generateFMInstruction($h$)
11  $pgm$ := connectFMInstructions()
12  return $pgm$, $\mathcal{M}_{opt}$
13  def removeAllCrossFwdEdges():
14      for $(u, v) \in \rightarrow_{\mathtt{CF}}$ do
15          $\rightarrow' := \rightarrow' \setminus \{(u, v)\}$
            ▷ Lowest common ancestor.
16          R[lca$_{\mathtt{D}}(u, v)$] := R[lca$_{\mathtt{D}}(u, v)$] $\cup \{(u, v)\}$
17  def restoreCrossFwdEdges($h$):
18      $\rightarrow' := \rightarrow' \cup \{(u, \mathtt{rep}(v)) \mid (u, v) \in \mathtt{R}[h]\}$
19  def findNestedSCCs($h$):
20      $B_h := \{\mathtt{rep}(p) \mid (p, h) \in \mathtt{B}\}$
21      $N_h := \emptyset$          ▷ Nested SCCs except $h$.
22      $W := B_h \setminus \{h\}$          ▷ Worklist.
23      while there exists $v \in W$ do
24          $W,\ N_h := W \setminus \{v\},\ N_h \cup [v]$
25          for $u$ s.t. $u \rightarrow' v$ do
26              if $\mathtt{rep}(u) \notin N_h \cup \{h\} \cup W$ then
27                  $W := W \cup \{\mathtt{rep}(u)\}$
28      return $N_h, B_h$
29  def generateFMInstruction($h$):
30      $N_h, B_h$ := findNestedSCCs($h$)
31      if $B_h = \emptyset$ then
32          Inst[$h$] := exec $h$
33          return
34      for $v \in N_h$ in desc. $postDFN_{\mathtt{D}}$ do
35          Inst[$h$] := Inst[$h$] $\fatsemi$ Inst[$v$]
*36         for $u$ s.t. $u \rightarrow' v$ do
*37             $\textsc{Dpost}_{\text{opt}}[u] := v$
*38             T[$u$] := rep($u$)
39      Inst[$h$] := repeat $h$ [Inst[$h$]]
*40     for $u$ s.t. $u \rightarrow_{\mathtt{B}} h$ do
*41         $\textsc{Dpost}_{\text{opt}}[u]$ := T[$u$] := $h$
42      for $v \in N_h$ do
43          merge($v, h$); P := P $\cup \{(v, h)\}$
44  def connectFMInstructions():
45      $pgm := \epsilon$          ▷ Empty program.
46      for $v \in V$ in desc. $postDFN_{\mathtt{D}}$ do
47          if rep($v$) = $v$ then
48              $pgm := pgm$ $\fatsemi$ Inst[$v$]
*49             for $u$ s.t. $u \rightarrow' v$ do
*50                 $\textsc{Dpost}_{\text{opt}}[u] := v$
*51                 T[$u$] := rep($u$)
*52         if $v \in V_C$ then
*53             $\textsc{Achk}_{\text{opt}}[v]$ := rep($v$)
*54             $\textsc{Dpre}^{\ell}_{\text{opt}}[v] := \llbracket v, \mathtt{rep}(v) \rrbracket_{\mathtt{P}*} \setminus \{v\}$
*55     for $v \in V$ do
*56         $\textsc{Dpost}^{\ell}_{\text{opt}}[v] := \llbracket v, T[v] \rrbracket_{\mathtt{P}*}$
57      return $pgm$
```

4 Efficient Algorithm to Compute $\mathcal{M}_{\text{opt}}$

Algorithm GenerateFMProgram (Algorithm 1) is an almost-linear time algorithm for computing an FM program P and optimal memory configuration $\mathcal{M}_{\text{opt}}$ for a given directed graph $G(V, \rightarrow)$. Algorithm 1 adapts the bottom-up WTO construction algorithm presented in Kim et al. [26]. In particular, Algorithm 1 applies the genProg rules (Eq. 3) to generate the FM program from a WTO. Line 32 generates exec instructions for non-heads. Line 39 generates repeat instructions for heads, with their bodies ([ ]) generated on Line 35. Finally, instructions are merged on Line 48 to construct the final output P.

Algorithm GenerateFMProgram utilizes a disjoint-set data structure. Operation rep(v) returns the representative of the set that contains v. In Line 5, the sets are initialized to be rep(v) = v for all $v \in V$. Operation merge(v, h) on Line 43 merges the sets containing v and h, and assigns h to be the representative for the combined set. lca$_{\mathtt{D}}(u, v)$ is the lowest common ancestor of u, v in the depth-first forest D [47]. Cross and forward edges are initially removed from $\rightarrow'$

on Line 7, making the graph $(V, \rightarrow' \cup \rightarrow_{\mathtt{B}})$ reducible. Restoring it on Line 9 when $h = \mathtt{lca}_{\mathtt{D}}(u, v)$ restores some reachability while keeping $(V, \rightarrow' \cup \rightarrow_{\mathtt{B}})$ reducible.

Lines indicated by $\star$ in Algorithm 1 compute $\mathcal{M}_{\text{opt}}$. Lines 37, 41, and 50 compute $\text{D}\textsc{post}_{\text{opt}}$. Due to the specific order in which the algorithm traverses G, $\text{D}\textsc{post}_{\text{opt}}[u]$ is overwritten with greater values (as per the total order $\leq$) on these lines, making the final value to be the maximum among the successors. `Lift` is implicitly applied when restoring the edges in `restoreCrossFwdEdges`: edge $u \rightarrow v$ whose $\mathtt{Lift}(u, v) = h$ is replaced to $u \rightarrow' h$ on Line 9.

$\text{D}\textsc{post}^{\ell}{}_{\text{opt}}$ is computed using an auxiliary map $\mathtt{T}\colon V \rightarrow V$ and a relation $\mathtt{P}\colon V \times V$. At the end of the algorithm, $\mathtt{T}[u]$ will be the maximum element (as per $\preceq_{\mathsf{N}}$) in $\text{D}\textsc{post}^{\ell}{}_{\text{opt}}[u]$. That is, $\mathtt{T}[u] = \max_{\preceq_{\mathsf{N}}}((\llbracket u \rrbracket_{\preceq_{\mathsf{N}}} \setminus \llbracket d \rrbracket_{\preceq_{\mathsf{N}}}) \cup \llparenthesis u \preceq_{\mathsf{N}} d \; ? \; \{d\} \; ⁏ \; \emptyset \rrparenthesis)$, where $d = \text{D}\textsc{post}_{\text{opt}}[u]$. Once $\mathtt{T}[u]$ is computed by lines 38, 41, and 51, the transitive reduction of $\preceq_{\mathsf{N}}$, $\mathtt{P}$, is used to find all elements of $\text{D}\textsc{post}^{\ell}{}_{\text{opt}}[u]$ on Line 56. $\mathtt{P}$ is computed on Line 43. Note that $\mathtt{P}^* = \preceq_{\mathsf{N}}$ and $\llbracket x, y \rrbracket_{\mathtt{P}^*} \overset{\text{def}}{=} \{v \mid x \; \mathtt{P}^* v \wedge v \; \mathtt{P}^* y\}$. $\text{A}\textsc{chk}$ and $\text{D}\textsc{pre}^{\ell}$ are computed on Lines 53 and 54, respectively. An example run of the algorithm on graph G_1 can be found in the extended version of this paper [27].

The proofs of the following theorems are in Appendix A.3:

Theorem 2. *`GenerateFMProgram` correctly computes $\mathcal{M}_{opt}$, defined in Sect. 3.*

Theorem 3. *Running time of `GenerateFMProgram` is almost-linear.*

5 Implementation

We have implemented our approach in a tool called Mikos, which extends NASA's IKOS [11], a WTO-based abstract-interpreter for C/C++. Mikos inherits all abstract domains and widening-narrowing strategies from IKOS. It includes the localized narrowing strategy [1] that intertwines the increasing and decreasing sequences.

Abstract Domains in IKOS. IKOS uses the state-of-the-art implementations of abstract domains comparable to those used in industrial abstract interpreters such as Astrée. In particular, IKOS implements the interval abstract domain [14] using functional data-structures based on Patricia Trees [35]. Astrée implements intervals using OCaml's map data structure that uses balanced trees [8, Section 6.2]. As shown in [35, Section 5], the Patricia Trees used by IKOS are more efficient when you have to merge data structures, which is required often during abstract interpretation. Also, IKOS uses memory-efficient variable packing Difference Bound Matrix (DBM) relational abstract domain [17], similar to the variable packing relational domains employed by Astrée [5, Section 3.3.2].

Interprocedural Analysis in IKOS. IKOS implements context-sensitive interprocedural analysis by means of dynamic inlining, much like the semantic expansion of function bodies in Astrée [15, Section 5]: at a function call, formal and actual parameters are matched, the callee is analyzed, and the return value at the call site is updated after the callee returns; a function pointer is resolved to a set of callees and the results for each call are joined; IKOS returns top for a callee when a cycle is found in this dynamic call chain. To prevent running the entire interprocedural analysis again at the assertion checking phase, invariants at exits of the callees are additionally cached during the fixpoint computation.

***Interprocedural Extension of* MIKOS.** Although the description of our iteration strategy focused on intraprocedural analysis, it can be extended to interprocedural analysis as follows. Suppose there is a call to function `f1` from a basic block contained in component C. Any checks in this call to `f1` must be deferred until we know that the component C has stabilized. Furthermore, if function `f1` calls the function `f2`, then the checks in `f2` must also be deferred until C converges. In general, checks corresponding to a function call f must be deferred until the maximal component containing the call is stabilized.

When the analysis of callee returns in MIKOS, only PRE values for the deferred checks remain. They are deallocated when the checks are performed or when the component containing the call is reiterated.

6 Experimental Evaluation

The experiments in this section were designed to answer the following questions:

RQ0 [Accuracy] Does MIKOS (Sect. 5) have the same analysis results as IKOS?
RQ1 [Memory footprint] How does the memory footprint of MIKOS compare to that of IKOS?
RQ2 [Runtime] How does the runtime of MIKOS compare to that of IKOS?

Experimental Setup. All experiments were run on Amazon EC2 r5.2 $\times$ large instances (64 GiB memory, 8 vCPUs, 4 physical cores), which use Intel Xeon Platinum 8175M processors. Processors have L1, L2, and L3 caches of sizes 1.5 MiB (data: 0.75 MiB, instruction: 0.75 MiB), 24 MiB, and 33 MiB, respectively. Linux kernel version 4.15.0-1051-aws was used, and gcc 7.4.0 was used to compile both MIKOS and IKOS. Dedicated EC2 instances and BenchExec [7] were used to improve reliability of the results. Time and space limit were set to an hour and 64 GB, respectively. The experiments can be reproduced using https://github.com/95616ARG/mikos_sas2020. Further experimental data can be found in the extended version of this paper [27].

Benchmarks. We evaluated MIKOS on two tasks that represent different client applications of abstract interpretation, each using different benchmarks described in Sects. 6.1 and 6.2. In both tasks, we excluded benchmarks that did not complete in *both* IKOS and MIKOS given the time and space budget. There were no benchmarks for which IKOS succeeded but MIKOS failed to complete. Benchmarks for which IKOS took less than 5 s were also excluded. Measurements for benchmarks that took less than 5 s are summarized in Appendix B of our extended paper [27].

Metrics. To answer RQ1, we define and use *memory reduction ratio (MRR)*:

$$\text{MRR} \overset{\text{def}}{=} \text{Memory footprint of Mikos} \,/\, \text{Memory footprint of IKOS} \qquad (10)$$

The smaller the MRR, the greater reduction in peak-memory usage in Mikos. If MRR is less than 1, Mikos has smaller memory footprint than IKOS.

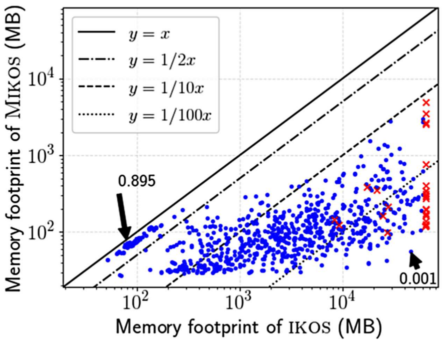
For RQ2, we report the *speedup*, which is defined as below:

$$\text{Speedup} \overset{\text{def}}{=} \text{Runtime of IKOS} \,/\, \text{Runtime of Mikos} \qquad (11)$$

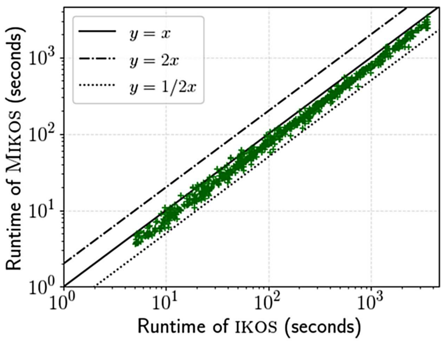
The larger the speedup, the greater reduction in runtime in Mikos. If speedup is greater than 1, Mikos is faster than IKOS.

RQ0: Accuracy of Mikos. As a sanity check for our theoretical results, we experimentally validated Theorem 1 by comparing the analysis results reported by IKOS and Mikos. Mikos used a valid memory configuration, reporting the same analysis results as IKOS. Recall that Theorem 1 also proves that the fixpoint computation in Mikos is memory-optimal (, it results in minimum memory footprint).

6.1 Task T1: Verifying User-Provided Assertions



(a) Min MRR: 0.895. Max MRR: 0.001. Geometric means: (i) 0.044 (when ×s are ignored), (ii) 0.041 (when measurements until timeout/spaceout are used for ×s). 29 non-completions in IKOS.



(b) Min speedup: 0.87×. Max speedup: 1.80×. Geometric mean: 1.29×. Note that ×s are ignored as they space out fast in IKOS compared to in Mikos where they complete.

Fig. 3. Task T1. Log-log scatter plots of (a) memory footprint and (b) runtime of IKOS and Mikos, with an hour timeout and 64 GB spaceout. Benchmarks that did not complete in IKOS are marked ×. All ×s completed in Mikos. Benchmarks below $y = x$ required less memory or runtime in Mikos.

Benchmarks. For Task T1, we selected all 2928 benchmarks from DeviceDriversLinux64, ControlFlow, and Loops categories of SV-COMP 2019 [6]. These categories are well suited for numerical analysis, and have been used in recent works [26,45,46]. From these benchmarks, we removed 435 benchmarks that timed out in both Mikos and IKOS, and 1709 benchmarks that took less than 5 s in IKOS. That left us with **784** SV-COMP 2019 benchmarks.

Abstract Domain. Task T1 used the reduced product of Difference Bound Matrix (DBM) with variable packing [17] and congruence [20]. This domain is much richer and more expressive than the interval domain used in task T2.

Task. Task T1 consists of using the results of interprocedural fixpoint computation to prove user-provided assertions in the SV-COMP benchmarks. Each benchmark typically has one assertion to prove.

RQ1: Memory footprint of* Mikos *compared to IKOS. Figure 3(a) shows the measured memory footprints in a log-log scatter plot. For Task T1, the MRR (Eq. 10) ranged from 0.895 to 0.001. That is, the memory footprint decreased to 0.1% in the best case. For all benchmarks, Mikos had smaller memory footprint than IKOS: MRR was less than 1 for all benchmarks, with all points below the $y = x$ line in Fig. 3(a). On average, Mikos required only 4.1% of the memory required by IKOS, with an MRR 0.041 as the geometric mean.

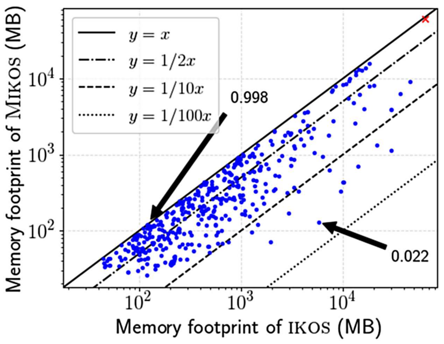
As Fig. 3(a) shows, reduction in memory tended to be greater as the memory footprint in the baseline IKOS grew. For the top 25% benchmarks with largest memory footprint in IKOS, the geometric mean of MRRs was 0.009. While a similar trend was observed in task T2, the trend was significantly stronger in task T1. Our extended paper has more detailed numbers [27].

RQ2: Runtime of* Mikos *compared to IKOS. Figure 3(b) shows the measured runtime in a log-log scatter plot. We measured both the speedup (Eq. 11) and the difference in the runtimes. For fair comparison, we excluded 29 benchmarks that did not complete in IKOS. This left us with 755 SV-COMP 2019 benchmarks. Out of these 755 benchmarks, 740 benchmarks had speedup > 1. The speedup ranged from 0.87× to 1.80×, with geometric mean of 1.29×. The difference in runtimes (runtime of IKOS − runtime of Mikos) ranged from −7.47 s to 1160.04 s, with arithmetic mean of 96.90 s. Our extended paper has more detailed numbers [27].

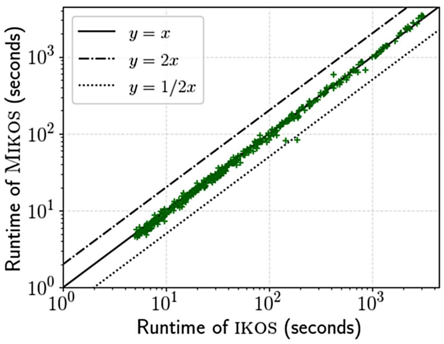
6.2 Task T2: Proving Absence of Buffer Overflows

Benchmarks. For Task T2, we selected all 1503 programs from the official Arch Linux core packages that are primarily written in C and whose LLVM bitcodes are obtainable by gllvm [19]. These include, but are not limited to, `coreutils`, `dhcp`, `gnupg`, `inetutils`, `iproute`, `nmap`, `openssh`, `vim`, etc. From these benchmarks, we removed 76 benchmarks that timed out and 8 benchmarks that spaced out in both Mikos and IKOS. Also, 994 benchmarks that took less than 5 s in IKOS were removed. That left us with **426** open-source benchmarks.

Abstract Domain. Task T2 used the interval abstract domain [14]. Using a richer domain like DBM caused IKOS and Mikos to timeout on most benchmarks.



(a) Min MRR: 0.998. Max MRR: 0.022. Geometric means: (i) 0.436 (when ×s are ignored), (ii) 0.437 (when measurements until timeout/spaceout are used for ×s). 1 non-completions in IKOS.



(b) Min speedup: 0.88×. Max speedup: 2.83×. Geometric mean: 1.08×. Note that ×s are ignored as they space out fast in IKOS compared to in MIKOS where they complete.

Fig. 4. Task T2. Log-log scatter plots of (a) memory footprint and (b) runtime of IKOS and MIKOS, with an hour timeout and 64 GB spaceout. Benchmarks that did not complete in IKOS are marked ×. All ×s completed in MIKOS. Benchmarks below $y = x$ required less memory or runtime in MIKOS.

Task. Task T2 consists of using the results of interprocedural fixpoint computation to prove the safety of buffer accesses. In this task, most program points had checks.

RQ1: Memory footprint of* MIKOS *compared to IKOS. Figure 4(a) shows the measured memory footprints in a log-log scatter plot. For Task T2, MRR (Eq. 10) ranged from 0.998 to 0.022. That is, the memory footprint decreased to 2.2% in the best case. For all benchmarks, MIKOS had smaller memory footprint than IKOS: MRR was less than 1 for all benchmarks, with all points below the $y = x$ line in Fig. 4(a). On average, MIKOS's memory footprint was less than half of that of IKOS, with an MRR 0.437 as the geometric mean. Our extended paper has more detailed numbers [27].

RQ2: Runtime of* MIKOS *compared to IKOS. Figure 4(b) shows the measured runtime in a log-log scatter plot. We measured both the speedup (Eq. 11) and the difference in the runtimes. For fair comparison, we excluded 1 benchmark that did not complete in IKOS. This left us with 425 open-source benchmarks. Out of these 425 benchmarks, 331 benchmarks had speedup > 1. The speedup ranged from 0.88× to 2.83×, with geometric mean of 1.08×. The difference in runtimes (runtime of IKOS − runtime of MIKOS) ranged from −409.74 s to 198.39 s, with arithmetic mean of 1.29 s. Our extended paper has more detailed numbers [27].

7 Related Work

Abstract interpretation has a long history of designing time and memory efficient algorithms for specific abstract domains, which exploit variable packing and clustering and sparse constraints [13,18,22,24,43–46]. Often these techniques represent a trade-off between precision and performance of the analysis. Nonetheless, such techniques are orthogonal to the abstract-domain agnostic approach discussed in this paper. Approaches for improving precision via sophisticated widening and narrowing strategies [2,3,21] are also orthogonal to our memory-efficient iteration strategy. Mikos inherits the interleaved widening-narrowing strategy implemented in the baseline IKOS abstract interpreter.

As noted in Sect. 1, Bourdoncle's approach [10] is used in many industrial and academic abstract interpreters [11,12,16,32,48]. Thus, improving memory efficiency of WTO-based exploration is of great applicability to real-world static analysis. Astrée is one of the few, if not only, industrial abstract interpreters that does not use WTO exploration, because it assumes that programs do not have gotos and recursion [8, Section 2.1], and is targeted towards a specific class of embedded C code [5, Section 3.2]. Such restrictions makes is easier to compute when an abstract value will not be used anymore by naturally following the abstract syntax tree [29, Section 3.4.3]. In contrast, Mikos works for general programs with goto and recursion, which requires the use of WTO-based exploration.

Generic fixpoint-computation approaches for improving running time of abstract interpretation have also been explored [26,30,52]. Most recently, Kim et al. [26] present the notion of weak partial order (WPO), which generalizes the notion of WTO that is used in this paper. Kim et al. describe a parallel fixpoint algorithm that exploits maximal parallelism while computing the same fixpoint as the WTO-based algorithm. Reasoning about correctness of concurrent algorithms is complex; hence, we decided to investigate an optimal memory management scheme in the sequential setting first. However, we believe it would be possible to extend our WTO-based result to one that uses WPO.

The nesting relation described in Sect. 3 is closely related to the notion of Loop Nesting Forest [36,37], as observed in Kim et al. [26]. The almost-linear time algorithm `GenerateFMProgram` is an adaptation of LNF construction algorithm by Ramalingam [36]. The `Lift` operation in Sect. 3 is similar to the outermost-loop-excluding (OLE) operator introduced by Rastello [38, Section 2.4.4].

Seidl et al. [42] present time and space improvements to a generic fixpoint solver, which is closest in spirit to the problem discussed in this paper. For improving space efficiency, their approach recomputes values during fixpoint computation, and does not prove optimality, unlike our approach. However, the setting discussed in their work is also more generic compared to ours; we assume a static dependency graph for the equation system.

Abstract interpreters such as Astrée [8] and CodeHawk [48] are implemented in OCaml, which provides a garbage collector. However, merely using a reference counting garbage collector will not reduce peak memory usage of fixpoint computation. For instance, the reference count of Pre$[u]$ can be decreased to zero only after the final check/assert that uses Pre$[u]$. If the checks are all conducted at the end of the analysis (as is currently done in prior tools), then using a reference counting garbage collector will not reduce peak memory usage. In contrast, our approach lifts the checks as early as possible enabling the analysis to free the abstract values as early as possible.

Symbolic approaches for applying abstract transformers during fixpoint computation [23,28,40,41,49–51] allow the entire loop body to be encoded as a single formula. This might appear to obviate the need for Pre and Post values for individual basic blocks within the loop; by storing the Pre value only at the header, such a symbolic approach might appear to reduce the memory footprint. First, this scenario does not account for the fact that Pre values need to be computed and stored if basic blocks in the loop have checks. Note that if there are no checks within the loop body, then our approach would also only store the Pre value at the loop header. Second, such symbolic approaches only perform intraprocedural analysis [23]; additional abstract values would need to be stored depending on how function calls are handled in interprocedural analysis. Third, due to the use of SMT solvers in such symbolic approaches, the memory footprint might not necessarily reduce, but might increase if one takes into account the memory used by the SMT solver.

Sparse analysis [33,34] and database-backed analysis [54] improve the memory cost of static analysis. For specific classes of static analysis such as the IFDS framework [39], there have been approaches for improving the time and memory efficiency [9,31,53,55].

8 Conclusion

This paper presented an approach for memory-efficient abstract interpretation that is agnostic to the abstract domain used. Our approach is memory-optimal and produces the same result as Bourdoncle's approach without sacrificing time efficiency. We extended the notion of iteration strategy to intelligently deallocate abstract values and perform assertion checks during fixpoint computation. We provided an almost-linear time algorithm that constructs this iteration strategy. We implemented our approach in a tool called Mikos, which extended the abstract interpreter IKOS. Despite the use of state-of-the-art implementation of abstract domains, IKOS had a large memory footprint on two analysis tasks. Mikos was shown to effectively reduce it. When verifying user-provided assertions in SV-COMP 2019 benchmarks, Mikos showed a decrease in peak-memory usage to 4.07% (24.57×) on average compared to IKOS. When performing interprocedural buffer-overflow analysis of open-source programs, Mikos showed a decrease in peak-memory usage to 43.7% (2.29×) on average compared to IKOS.

A Proofs

This section provides proofs of theorems presented in the paper.

A.1 Nesting forest $(V, \preceq_{\mathsf{N}})$ and total order $(V, \leq)$ in Sect. 3

This section presents the theorems and proofs about $\preceq_{\mathsf{N}}$ and $\leq$ defined in Sect. 3.

A partial order (S, R) is a forest if for all $x \in S$, $(\llbracket x \rceil_{\mathsf{R}}, R)$ is a chain, where $\llbracket x \rceil_{\mathsf{R}} \overset{\text{def}}{=} \{y \in S \mid x \mathrel{\mathsf{R}} y\}$.

Theorem 4. *$(V, \preceq_{\mathsf{N}})$ is a forest.*

Proof. First, we show that $(V, \preceq_{\mathsf{N}})$ is a partial order. Let x, y, z be a vertex in V.

- Reflexivity: $x \preceq_{\mathsf{N}} x$. This is true by the definition of $\preceq_{\mathsf{N}}$.
- Transitivity: $x \preceq_{\mathsf{N}} y$ and $y \preceq_{\mathsf{N}} z$ implies $x \preceq_{\mathsf{N}} z$. (i) If $x = y$, $x \preceq_{\mathsf{N}} z$. (ii) Otherwise, by definition of $\preceq_{\mathsf{N}}$, $y \in \omega(x)$. Furthermore, (ii-1) if $y = z$, $z \in \omega(x)$; and hence, $x \preceq_{\mathsf{N}} z$. (ii-2) Otherwise, $z \in \omega(y)$, and by definition of HTO, $z \in \omega(x)$.
- Anti-symmetry: $x \preceq_{\mathsf{N}} y$ and $y \preceq_{\mathsf{N}} x$ implies $x = y$. Suppose $x \neq y$. By definition of $\preceq_{\mathsf{N}}$ and premises, $y \in \omega(x)$ and $x \in \omega(y)$. Then, by definition of HTO, $x \prec y$ and $y \prec x$. This contradicts that $\preceq$ is a total order.

Next, we show that the partial order is a forest. Suppose there exists $v \in V$ such that $(\llbracket v \rceil_{\preceq_{\mathsf{N}}}, \preceq_{\mathsf{N}})$ is not a chain. That is, there exists $x, y \in \llbracket v \rceil_{\preceq_{\mathsf{N}}}$ such that $x \not\preceq_{\mathsf{N}} y$ and $y \not\preceq_{\mathsf{N}} x$. Then, by definition of HTO, $\mathcal{C}(x) \cap \mathcal{C}(y) = \emptyset$. However, this contradicts that $v \in \mathcal{C}(x)$ and $v \in \mathcal{C}(y)$. □

Theorem 5. *$(V, \leq)$ is a total order.*

Proof. We prove the properties of a total order. Let x, y, z be a vertex in V.

- Connexity: $x \leq y$ or $y \leq x$. This follows from the connexity of the total order $\preceq$.
- Transitivity: $x \leq y$ and $y \leq z$ implies $x \leq z$. (i) Suppose $x \preceq_{\mathsf{N}} y$. (i-1) If $y \preceq_{\mathsf{N}} z$, by transitivity of $\preceq_{\mathsf{N}}$, $x \preceq_{\mathsf{N}} z$. (ii-2) Otherwise, $z \not\preceq_{\mathsf{N}} y$ and $y \preceq z$. It cannot be $z \preceq_{\mathsf{N}} x$ because transitivity of $\preceq_{\mathsf{N}}$ implies $z \preceq_{\mathsf{N}} y$, which is a contradiction. Furthermore, it cannot be $z \prec x$ because $y \preceq z \prec x$ and $x \preceq_{\mathsf{N}} y$ implies $y \in \omega(z)$ by the definition of HTO. By connexity of $\preceq$, $x \preceq z$. (ii) Otherwise $y \not\preceq_{\mathsf{N}} x$ and $x \preceq y$. (ii-1) If $y \preceq_{\mathsf{N}} z$, $z \not\preceq_{\mathsf{N}} x$ because, otherwise, transitivity of $\preceq_{\mathsf{N}}$ will imply $y \preceq_{\mathsf{N}} x$. By connexity of $\preceq$, it is either $x \preceq z$ or $z \prec x$. If $x \preceq z$, $x \leq z$. If $z \prec x$, by definition of HTO, $z \in \omega(z)$.
- Anti-symmetry: $x \leq y$ and $y \leq x$ implies $x = y$. (i) If $x \preceq_{\mathsf{N}} y$, it should be $y \preceq_{\mathsf{N}} x$ for $y \leq x$ to be true. By anti-symmetry of $\preceq_{\mathsf{N}}$, $x = y$. (ii) Otherwise, $y \not\preceq_{\mathsf{N}} x$ and $x \preceq y$. For $y \leq x$ to be true, $x \not\preceq_{\mathsf{N}} y$ and $x \preceq y$. By anti-symmetry of $\preceq$, $x = y$.

□

Theorem 6. *For $u, v \in V$, if* `Inst`$[v]$ *reads* $\textsc{Post}[u]$*, then* $u \leq v$.

Proof. By the definition of the mapping `Inst`, there must exists $v' \in V$ such that $u \rightarrow v'$ and $v' \preceq_{\mathsf{N}} v$ for `Inst`$[v]$ to read $\textsc{Post}[u]$. By the definition of WTO, it is either $u \prec v'$ and $v' \notin \omega(u)$, or $v' \preceq u$ and $v' \in \omega(u)$. In both cases, $u \leq v'$. Because $v' \preceq_{\mathsf{N}} v$, and hence $v' \leq v$, $u \leq v$. □

A.2 Optimality of $\mathcal{M}_{\text{opt}}$ in Sect. 3

This section presents the theorems and proofs about the optimality of $\mathcal{M}_{\text{opt}}$ described in Sect. 3. The theorem is divided into optimality theorems of the maps that constitute $\mathcal{M}_{\text{opt}}$.

Given $\mathcal{M}(\textsc{Dpost}, \textsc{Achk}, \textsc{Dpost}^{\ell}, \textsc{Dpre}^{\ell})$ and a map $\textsc{Dpost}_0$, we use $\mathcal{M} \lightning \textsc{Dpost}_0$ to denote the memory configuration ($\textsc{Dpost}_0$, $\textsc{Achk}$, $\textsc{Dpost}^{\ell}$, $\textsc{Dpre}^{\ell}$). Similarly, $\mathcal{M} \lightning \textsc{Achk}_0$ means ($\textsc{Dpost}$, $\textsc{Achk}_0$, $\textsc{Dpost}^{\ell}$, $\textsc{Dpre}^{\ell}$), and so on. For a given `FM` program P, each map X that constitutes a memory configuration is valid for P iff $\mathcal{M} \lightning X$ is valid for every valid memory configuration $\mathcal{M}$. Also, X is optimal for P iff $\mathcal{M} \lightning X$ is optimal for an optimal memory configuration $\mathcal{M}$.

Theorem 7. $\textsc{Dpost}_{opt}$ *is valid. That is, given an* `FM` *program P and a valid memory configuration $\mathcal{M}$, $[\![P]\!]_{\mathcal{M} \lightning \textsc{Dpost}_{opt}} = [\![P]\!]_{\mathcal{M}}$.*

Proof. Our approach does not change the iteration order and only changes where the deallocations are performed. Therefore, it is sufficient to show that for all $u \rightarrow v$, $\textsc{Post}[u]$ is available whenever `Inst`$[v]$ is executed.

Suppose that this is false: there exists an edge $u \rightarrow v$ that violates it. Let d be $\textsc{Dpost}_{\text{opt}}[u]$ computed by our approach. Then, the execution trace of P has execution of `Inst`$[v]$ after the deallocation of $\textsc{Post}[u]$ in `Inst`$[d]$, with no execution of `Inst`$[u]$ in between.

Because $\leq$ is a total order, it is either $d < v$ or $v \leq d$. It must be $v \leq d$, because $d < v$ implies $d < v \leq$ `Lift`(u, v), which contradicts the definition of $\textsc{Dpost}_{\text{opt}}[u]$. Then, by definition of $\leq$, it is either $v \preceq_{\mathsf{N}} d$ or $(d \not\preceq_{\mathsf{N}} v) \wedge (v \preceq d)$. In both cases, the only way `Inst`$[v]$ can be executed after `Inst`$[d]$ is to have another head h whose `repeat` instruction includes both `Inst`$[d]$ and `Inst`$[v]$. That is, when $d \prec_{\mathsf{N}} h$ and $v \prec_{\mathsf{N}} h$. By definition of WTO and $u \rightarrow v$, it is either $u \prec v$, or $u \preceq_{\mathsf{N}} v$. It must be $u \prec v$, because if $u \preceq_{\mathsf{N}} v$, `Inst`$[u]$ is part of `Inst`$[v]$, making `Inst`$[u]$ to be executed before reading $\textsc{Post}[u]$ in `Inst`$[v]$. Furthermore, it must be $u \prec h$, because if $h \preceq u$, `Inst`$[u]$ is executed before `Inst`$[v]$ in each iteration over $\mathcal{C}(h)$. However, that implies $h \in (\llbracket v \rceil_{\preceq_{\mathsf{N}}} \setminus \llbracket u \rceil_{\preceq_{\mathsf{N}}})$, which combined with $d \prec_{\mathsf{N}} h$, contradicts the definition of $\textsc{Dpost}_{\text{opt}}[u]$. Therefore, no such edge $u \rightarrow v$ can exist and the theorem is true. □

Theorem 8. $\textsc{Dpost}_{opt}$ *is optimal. That is, given an* `FM` *program P, memory footprint of $[\![P]\!]_{\mathcal{M} \lightning \textsc{Dpost}_{opt}}$ is smaller than or equal to that of $[\![P]\!]_{\mathcal{M}}$ for all valid memory configuration $\mathcal{M}$.*

Proof. For $\textsc{Dpost}_{\text{opt}}$ to be optimal, deallocation of $\textsc{Post}$ values must be determined at earliest positions as possible with a valid memory configuration $\mathcal{M} \lightning \textsc{Dpost}_{\text{opt}}$. That is, there should not exists $u, b \in V$ such that if $d = \textsc{Dpost}_{\text{opt}}[u]$, $b \neq d$, $\mathcal{M} \lightning (\textsc{Dpost}_{\text{opt}}[u \leftarrow b])$ is valid, and $\texttt{Inst}[b]$ deletes $\textsc{Post}[u]$ earlier than $\texttt{Inst}[d]$.

Suppose that this is false: such u, b exists. Let d be $\textsc{Dpost}_{\text{opt}}[u]$, computed by our approach. Then it must be $b < d$ for $\texttt{Inst}[b]$ to be able to delete $\textsc{Post}[u]$ earlier than $\texttt{Inst}[d]$. Also, for all $u \rightarrow v$, it must be $v \leq b$ for $\texttt{Inst}[v]$ to be executed before deleting $\textsc{Post}[u]$ in $\texttt{Inst}[b]$.

By definition of $\textsc{Dpost}_{\text{opt}}$, $v \leq d$ for all $u \rightarrow v$. Also, by Theorem 6, $u \leq v$. Hence, $u \leq d$, making it either $u \preceq_{\mathsf{N}} d$, or $(d \not\preceq_{\mathsf{N}} u) \land (u \preceq d)$. If $u \preceq_{\mathsf{N}} d$, by definition of $\texttt{Lift}$, it must be $u \rightarrow d$. Therefore, it must be $d \leq b$, which contradicts that $b < d$. Alternative, if $(d \not\preceq_{\mathsf{N}} u) \land (u \preceq d)$, there must exist $v \in V$ such that $u \rightarrow v$ and $\texttt{Lift}(u, v) = d$. To satisfy $v \leq b$, $v \preceq_{\mathsf{N}} d$, and $b < d$, it must be $b \preceq_{\mathsf{N}} d$. However, this makes the analysis incorrect because when stabilization check fails for $\mathcal{C}(d)$, $\texttt{Inst}[v]$ gets executed again, attempting to read $\textsc{Post}[u]$ that is already deleted by $\texttt{Inst}[b]$. Therefore, no such u, b can exist, and the theorem is true. □

Theorem 9. $\textsc{Achk}_{opt}$ *is valid. That is, given an* $\texttt{FM}$ *program* P *and a valid memory configuration* $\mathcal{M}$*,* $[\![P]\!]_{\mathcal{M} \lightning \textsc{Achk}_{opt}} = [\![P]\!]_{\mathcal{M}}$

Proof. Let $v = \textsc{Achk}_{\text{opt}}[u]$. If v is a head, by definition of $\textsc{Achk}_{\text{opt}}$, $\mathcal{C}(v)$ is the largest component that contains u. Therefore, once $\mathcal{C}(v)$ is stabilized, $\texttt{Inst}[u]$ can no longer be executed, and $\textsc{Pre}[u]$ remains the same. If v is not a head, then $v = u$. That is, there is no component that contains u. Therefore, $\textsc{Pre}[u]$ remains the same after the execution of $\texttt{Inst}[u]$. In both cases, the value passed to $\textsc{Ck}_u$ are the same as when using $\textsc{Achk}_{\text{dflt}}$. □

Theorem 10. $\textsc{Achk}_{opt}$ *is optimal. That is, given an* $\texttt{FM}$ *program* P*, memory footprint of* $[\![P]\!]_{\mathcal{M} \lightning \textsc{Achk}_{opt}}$ *is smaller than or equal to that of* $[\![P]\!]_{\mathcal{M}}$ *for all valid memory configuration* $\mathcal{M}$*.*

Proof. Because $\textsc{Pre}$ value is deleted right after its corresponding assertions are checked, it is sufficient to show that assertion checks are placed at the earliest positions with $\textsc{Achk}_{\text{opt}}$.

Let $v = \textsc{Achk}_{\text{opt}}[u]$. By definition of $\textsc{Achk}_{\text{opt}}$, $u \preceq_{\mathsf{N}} v$. For some b to perform assertion checks of u earlier than v, it must satisfy $b \prec_{\mathsf{N}} v$. However, because one cannot know in advance when a component of v would stabilize and when $\textsc{Pre}[u]$ would converge, the assertion checks of u cannot be performed in $\texttt{Inst}[b]$. Therefore, our approach puts the assertion checks at the earliest positions, and it leads to the minimum memory footprint. □

Theorem 11. ${\textsc{Dpost}^{\ell}}_{opt}$ *is valid. That is, given an* $\texttt{FM}$ *program* P *and a valid memory configuration* $\mathcal{M}$*,* $[\![P]\!]_{\mathcal{M} \lightning {\textsc{Dpost}^{\ell}}_{opt}} = [\![P]\!]_{\mathcal{M}}$.

Proof. Again, our approach does not change the iteration order and only changes where the deallocations are performed. Therefore, it is sufficient to show that for all $u \rightarrow v$, $\textsc{Post}[u]$ is available whenever $\texttt{Inst}[v]$ is executed.

Suppose that this is false: there exists an edge $u \rightarrow v$ that violates it. Let d' be element in ${\textsc{Dpost}^{\ell}}_{\text{opt}}[u]$ that causes this violation. Then, the execution trace of P has execution of $\texttt{Inst}[v]$ after the deallocation of $\textsc{Post}[u]$ in $\texttt{Inst}[d']$, with no execution of $\texttt{Inst}[u]$ in between. Because $\textsc{Post}[u]$ is deleted inside the loop of $\texttt{Inst}[d']$, $\texttt{Inst}[v]$ must be nested in $\texttt{Inst}[d']$ or be executed after $\texttt{Inst}[d']$ to be affected. That is, it must be either $v \preceq_{\mathsf{N}} d'$ or $d' \prec v$. Also, because of how ${\textsc{Dpost}^{\ell}}_{\text{opt}}[u]$ is computed, $u \preceq_{\mathsf{N}} d'$.

First consider the case $v \preceq_{\mathsf{N}} d'$. By definition of WTO and $u \rightarrow v$, it is either $u \prec v$ or $u \preceq_{\mathsf{N}} v$. In either case, $\texttt{Inst}[u]$ gets executed before $\texttt{Inst}[v]$ reads $\textsc{Post}[u]$. Therefore, deallocation of $\textsc{Post}[u]$ in $\texttt{Inst}[d']$ cannot cause the violation.

Alternatively, consider $d' \prec v$ and $v \not\preceq_{\mathsf{N}} d'$. Because $u \preceq_{\mathsf{N}} d'$, $\textsc{Post}[u]$ is generated in each iteration over $\mathcal{C}(d')$, and the last iteration does not delete $\textsc{Post}[u]$. Therefore, $\textsc{Post}[u]$ will be available when executing $\texttt{Inst}[v]$. Therefore, such u, d' does not exists, and the theorem is true. □

Theorem 12. ${\textsc{Dpost}^{\ell}}_{opt}$ *is optimal. That is, given an* `FM` *program* P*, memory footprint of* $[\![P]\!]_{\mathcal{M} \lightning {\textsc{Dpost}^{\ell}}_{opt}}$ *is smaller than or equal to that of* $[\![P]\!]_{\mathcal{M}}$ *for all valid memory configuration* $\mathcal{M}$*.*

Proof. Because one cannot know when a component would stabilize in advance, the decision to delete intermediate $\textsc{Post}[u]$ cannot be made earlier than the stabilization check of a component that contains u. Our approach makes such decisions in all relevant components that contains u.

If $u \preceq_{\mathsf{N}} d$, ${\textsc{Dpost}^{\ell}}_{\text{opt}}[u] = [\![u]\!]_{\preceq_{\mathsf{N}}} \cap \lfloor d]\!]_{\preceq_{\mathsf{N}}}$. Because $\textsc{Post}[u]$ is deleted in $\texttt{Inst}[d]$, we do not have to consider components in $[\![d]\!]_{\preceq_{\mathsf{N}}} \setminus \{d\}$. Alternatively, if $u \not\preceq_{\mathsf{N}} d$, ${\textsc{Dpost}^{\ell}}_{\text{opt}}[u] = [\![u]\!]_{\preceq_{\mathsf{N}}} \setminus [\![d]\!]_{\preceq_{\mathsf{N}}}$. Because $\textsc{Post}[u]$ is deleted $\texttt{Inst}[d]$, we do not have to consider components in $[\![u]\!]_{\preceq_{\mathsf{N}}} \setminus [\![d]\!]_{\preceq_{\mathsf{N}}}$. Therefore, ${\textsc{Dpost}^{\ell}}_{\text{opt}}$ is optimal. □

Theorem 13. ${\textsc{Dpre}^{\ell}}_{opt}$ *is valid. That is, given an* `FM` *program* P *and a valid memory configuration* $\mathcal{M}$*,* $[\![P]\!]_{\mathcal{M} \lightning {\textsc{Dpre}^{\ell}}_{opt}} = [\![P]\!]_{\mathcal{M}}$*.*

Proof. $\textsc{Pre}[u]$ is only used in assertion checks and to perform widening in $\texttt{Inst}[u]$. Because u is removed from $\textsc{Dpre}^{\ell}[u]$, the deletion does not affect widening.

For all $v \in \textsc{Dpre}^{\ell}[u]$, $v \preceq_{\mathsf{N}} \textsc{Achk}_{\text{opt}}[u]$. Because $\textsc{Pre}[u]$ is not deleted when $\mathcal{C}(v)$ is stabilized, $\textsc{Pre}[u]$ will be available when performing assertion checks in $\texttt{Inst}[\textsc{Achk}_{\text{opt}}[u]]$. Therefore, $\textsc{Dpre}^{\ell}$ is valid. □

Theorem 14. ${\textsc{Dpre}^{\ell}}_{opt}$ *is optimal. That is, given an* `FM` *program* P*, memory footprint of* $[\![P]\!]_{\mathcal{M} \lightning {\textsc{Dpre}^{\ell}}_{opt}}$ *is smaller than or equal to that of* $[\![P]\!]_{\mathcal{M}}$ *for all valid memory configuration* $\mathcal{M}$*.*

Proof. Because one cannot know when a component would stabilize in advance, the decision to delete intermediate $\textsc{Pre}[u]$ cannot be made earlier than the stabilization check of a component that contains u. Our approach makes such decisions in all components that contains u. Therefore, $\textsc{Dpre}^{\ell}{}_{\mathrm{opt}}$ is optimal. $\square$

Theorem 1. *The memory configuration* $\mathcal{M}_{\mathrm{opt}}(\textsc{Dpost}_{\mathrm{opt}}, \textsc{Achk}_{\mathrm{opt}}, \textsc{Dpost}^{\ell}{}_{\mathrm{opt}}, \textsc{Dpre}^{\ell}{}_{\mathrm{opt}})$ *is optimal.*

Proof. This follows from theorems Theorem 11 to 14. $\square$

A.3 Correctness and efficiency of `GenerateFMProgram` in Sect. 4

This section presents the theorems and proofs about the correctness and efficiency of `GenerateFMProgram` (Algorithm 1, Sect. 4).

Theorem 2. *`GenerateFMProgram` correctly computes $\mathcal{M}_{opt}$, defined in Sect. 3.*

Proof. We show that each map is constructed correctly.

- $\textsc{Dpost}_{\mathrm{opt}}$: Let v' be the value of $\textsc{Dpost}_{\mathrm{opt}}[u]$ before overwritten in Line 50, 37, or 41. Descending post DFN ordering corresponds to a topological sorting of the nested SCCs. Therefore, in Line 50 and 37, $v' \prec v$. Also, because $v \preceq_{\mathsf{N}} h$ for all $v \in N_h$ in Line 41, $v' \preceq_{\mathsf{N}} v$. In any case, $v' \leq v$. Because $\mathtt{rep}(v)$ essentially performs $\mathtt{Lift}(u, v)$ when restoring the edges, the final $\textsc{Dpost}_{\mathrm{opt}}[u]$ is the maximum of the lifted successors, and the map is correctly computed.
- $\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}$: The correctness follows from the correctness of $\mathtt{T}$. Because the components are constructed bottom-up, $\mathtt{rep}(u)$ in Line 51 and 38 returns $\max_{\preceq_{\mathsf{N}}}(\llbracket u \rrbracket_{\preceq_{\mathsf{N}}} \setminus \llbracket \textsc{Dpost}_{\mathrm{opt}}[u] \rrbracket_{\preceq_{\mathsf{N}}})$. Also, $\mathsf{N}^* = \preceq_{\mathsf{N}}$. Thus, $\textsc{Dpost}^{\ell}{}_{\mathrm{opt}}$ is correctly computed.
- $\textsc{Achk}_{\mathrm{opt}}$: At the end of the algorithm $\mathtt{rep}(v)$ is the head of maximal component that contains v, or v itself when v is outside of any components. Therefore, $\textsc{Achk}_{\mathrm{opt}}$ is correctly computed.
- $\textsc{Dpre}^{\ell}{}_{\mathrm{opt}}$: Using the same reasoning as in $\textsc{Achk}_{\mathrm{opt}}$, and because $\mathsf{N}^* = \preceq_{\mathsf{N}}$, $\textsc{Dpre}^{\ell}{}_{\mathrm{opt}}$ is correctly computed.

$\square$

Theorem 3. *Running time of `GenerateFMProgram` is almost-linear.*

Proof. The base WTO-construction algorithm is almost-linear time [26]. The starred lines in Algorithm 1 visit each edge and vertex once. Therefore, time complexity still remains almost-linear time. $\square$

References

1. Amato, G., Scozzari, F.: Localizing widening and narrowing. In: Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20–22, 2013. Proceedings. pp. 25–42 (2013). https://doi.org/10.1007/978-3-642-38856-9_4
2. Amato, G., Scozzari, F., Seidl, H., Apinis, K., Vojdani, V.: Efficiently intertwining widening and narrowing. Sci. Comput. Program. **120**, 1–24 (2016). https://doi.org/10.1016/j.scico.2015.12.005
3. Apinis, K., Seidl, H., Vojdani, V.: Enhancing top-down solving with widening and narrowing. In: Probst, C.W., Hankin, C., Hansen, R.R. (eds.) Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays. Lecture Notes in Computer Science, vol. 9560, pp. 272–288. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-27810-0_14
4. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (2008). https://doi.org/10.1016/j.scico.2007.08.001
5. Bertrane, J., et al.: Static analysis by abstract interpretation of embedded critical software. ACM SIGSOFT Softw. Eng. Notes **36**(1), 1–8 (2011). https://doi.org/10.1145/1921532.1921553
6. Beyer, D.: Automatic verification of C and java programs: SV-COMP 2019. In: Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III, pp. 133–155 (2019). https://doi.org/10.1007/978-3-030-17502-3_9
7. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2017). https://doi.org/10.1007/s10009-017-0469-y
8. Blanchet, B., et al.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]. Lecture Notes in Computer Science, vol. 2566, pp. 85–108. Springer, Cham (2002). https://doi.org/10.1007/3-540-36377-7_5
9. Bodden, E.: Inter-procedural data-flow analysis with IFDS/IDE and soot. In: Bodden, E., Hendren, L.J., Lam, P., Sherman, E. (eds.) Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012, pp. 3–8. ACM (2012). https://doi.org/10.1145/2259051.2259052
10. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Bjørner, D., Broy, M., Pottosin, I.V. (eds.) FMP&TA 1993. LNCS, vol. 735, pp. 128–141. Springer, Heidelberg (1993). https://doi.org/10.1007/BFb0039704
11. Brat, G., Navas, J.A., Shi, N., Venet, A.: IKOS: A framework for static analysis based on abstract interpretation. In: Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1–5, 2014. Proceedings, pp. 271–277 (2014). https://doi.org/10.1007/978-3-319-10431-7_20
12. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.)

NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18–20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 459–465. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_33

13. Chawdhary, A., King, A.: Compact difference bound matrices. In: Chang, B.E. (ed.) Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27–29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10695, pp. 471–490. Springer (2017). https://doi.org/10.1007/978-3-319-71237-6_23
14. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, pp. 238–252 (1977). https://doi.org/10.1145/512950.512973
15. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Sagiv, S. (ed.) Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4–8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3444, pp. 21–30. Springer, Cham (2005). https://doi.org/10.1007/978-3-540-31987-0_3
16. Facebook: Sparta. https://github.com/facebookincubator/SPARTA (2020)
17. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: An abstract domain of uninterpreted functions. In: Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17–19, 2016. Proceedings. pp. 85–103 (2016). https://doi.org/10.1007/978-3-662-49122-5_4
18. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Exploiting sparsity in difference-bound matrices. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8–10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 189–211. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_10
19. gllvm. https://github.com/SRI-CSL/gllvm (2020)
20. Granger, P.: Static analysis of arithmetical congruences. Int. J. Comput. Math. **30**(3–4), 165–190 (1989). https://doi.org/10.1080/00207168908803778
21. Halbwachs, N., Henry, J.: When the decreasing sequence fails. In: Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11–13, 2012. Proceedings, pp. 198–213 (2012). https://doi.org/10.1007/978-3-642-33125-1_15
22. Halbwachs, N., Merchat, D., Gonnord, L.: Some ways to reduce the space dimension in polyhedra computations. Formal Methods Syst. Des. **29**(1), 79–95 (2006). https://doi.org/10.1007/s10703-006-0013-2
23. Henry, J., Monniaux, D., Moy, M.: PAGAI: a path sensitive static analyser. Electron. Notes Theor. Comput. Sci. **289**, 15–25 (2012). https://doi.org/10.1016/j.entcs.2012.11.003
24. Heo, K., Oh, H., Yang, H.: Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8–10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 237–256. Springer, Cham (2016). https://doi.org/10.1007/978-3-662-53413-7_12

25. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
26. Kim, S.K., Venet, A.J., Thakur, A.V.: Deterministic parallel fixpoint computation. PACMPL **4**(POPL), 14:1–14:33 (2020). https://doi.org/10.1145/3371082
27. Kim, S.K., Venet, A.J., Thakur, A.V.: Memory-efficient fixpoint computation (2020). https://arxiv.org/abs/2009.05865
28. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with SMT solvers. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, January 20–21, 2014. pp. 607–618. ACM (2014). https://doi.org/10.1145/2535838.2535857
29. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Found. Trends Program. Lang. **4**(3–4), 120–372 (2017). https://doi.org/10.1561/2500000034
30. Monniaux, D.: The parallel implementation of the astrée static analyzer. In: Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2–5, 2005, Proceedings, pp. 86–96 (2005). https://doi.org/10.1007/11575467_7
31. Naeem, N.A., Lhoták, O., Rodriguez, J.: Practical extensions to the IFDS algorithm. In: Gupta, R. (ed.) Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20–28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6011, pp. 124–144. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11970-5_8
32. Navas, J.A.: Crab: Cornucopia of abstractions: a language-agnostic library for abstract interpretation. https://github.com/seahorn/crab (2019)
33. Oh, H., Heo, K., Lee, W., Lee, W., Park, D., Kang, J., Yi, K.: Global sparse analysis framework. ACM Trans. Program. Lang. Syst. **36**(3), 8:1–8:44 (2014). https://doi.org/10.1145/2590811
34. Oh, H., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for c-like languages. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China - June 11–16, 2012, pp. 229–238 (2012). https://doi.org/10.1145/2254064.2254092
35. Okasaki, C., Gill, A.: Fast mergeable integer maps. In: Workshop on ML, pp. 77–86 (1998)
36. Ramalingam, G.: Identifying loops in almost linear time. ACM Trans. Program. Lang. Syst. **21**(2), 175–188 (1999). https://doi.org/10.1145/316686.316687
37. Ramalingam, G.: On loops, dominators, and dominance frontiers. ACM Trans. Program. Lang. Syst. **24**(5), 455–490 (2002). https://doi.org/10.1145/570886.570887
38. Rastello, F.: On Sparse Intermediate Representations: Some Structural Properties and Applications to Just-In-Time Compilation. University works, Inria Grenoble Rhône-Alpes (Dec 2012). https://hal.inria.fr/hal-00761555, habilitation à diriger des recherches, École normale supérieure de Lyon
39. Reps, T.W., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Conference Record of POPL 1995: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23–25, 1995, pp. 49–61 (1995). https://doi.org/10.1145/199448.199462

40. Reps, T.W., Sagiv, S., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11–13, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2937, pp. 252–266. Springer, New York (2004). https://doi.org/10.1007/978-3-540-24622-0_21
41. Reps, T.W., Thakur, A.V.: Automating abstract interpretation. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17–19, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9583, pp. 3–40. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_1
42. Seidl, H., Vogler, R.: Three improvements to the top-down solver. In: Sabel, D., Thiemann, P. (eds.) Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03–05, 2018, pp. 21:1–21:14. ACM (2018). https://doi.org/10.1145/3236950.3236967
43. Singh, G., Püschel, M., Vechev, M.T.: Making numerical program analysis fast. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015, pp. 303–313 (2015). https://doi.org/10.1145/2737924.2738000
44. Singh, G., Püschel, M., Vechev, M.T.: Fast polyhedra abstract domain. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017, pp. 46–59. ACM (2017). https://doi.org/10.1145/3009837.3009885
45. Singh, G., Püschel, M., Vechev, M.T.: Fast numerical program analysis with reinforcement learning. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14–17, 2018, Proceedings, Part I. pp. 211–229 (2018). https://doi.org/10.1007/978-3-319-96145-3_12
46. Singh, G., Püschel, M., Vechev, M.T.: A practical construction for decomposing numerical abstract domains. In: Proceedings ACM Programming Language **2**(POPL), 55:1–55:28 (2018). https://doi.org/10.1145/3158143
47. Tarjan, R.E.: Applications of path compression on balanced trees. J. ACM **26**(4), 690–715 (1979). https://doi.org/10.1145/322154.322161
48. Technology, K.: Codehawk. https://github.com/kestreltechnology/codehawk (2020)
49. Thakur, A.V., Elder, M., Reps, T.W.: Bilateral algorithms for symbolic abstraction. In: Miné, A., Schmidt, D. (eds.) Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11–13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7460, pp. 111–128. Springer, Cham (2012). https://doi.org/10.1007/978-3-642-33125-1_10
50. Thakur, A.V., Lal, A., Lim, J., Reps, T.W.: Posthat and all that: automating abstract interpretation. Electron. Notes Theor. Comput. Sci. **311**, 15–32 (2015). https://doi.org/10.1016/j.entcs.2015.02.003
51. Thakur, A.V., Reps, T.W.: A method for symbolic computation of abstract operations. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7–13, 2012 Proceedings. Lecture Notes in Computer Science, vol. 7358, pp. 174–192. Springer, Heeidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_17

52. Venet, A., Brat, G.P.: Precise and efficient static array bound checking for large embedded C programs. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9–11, 2004 pp. 231–242 (2004). https://doi.org/10.1145/996841.996869
53. Wang, K., Hussain, A., Zuo, Z., Xu, G.H., Sani, A.A.: Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8–12, 2017, pp. 389–404 (2017). https://doi.org/10.1145/3037697.3037744
54. Weiss, C., Rubio-González, C., Liblit, B.: Database-backed program analysis for scalable error propagation. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, vol. 1. pp. 586–597 (2015). https://doi.org/10.1109/ICSE.2015.75
55. Zuo, Z., Gu, R., Jiang, X., Wang, Z., Huang, Y., Wang, L., Li, X.: Bigspa: an efficient interprocedural static analysis engine in the cloud. In: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20–24, 2019, pp. 771–780. IEEE (2019). https://doi.org/10.1109/IPDPS.2019.00086