A Sliding-Window Algorithm for On-The-Fly Interprocedural Program Analysis

Xin Li(✉)[1] and Mizuhito Ogawa[2]

[1] East China Normal University, Shanghai, China
xinli@sei.ecnu.edu.cn
[2] Japan Advanced Institute of Science and Technology, Nomi, Japan
mizuhito@jaist.ac.jp

Abstract. Program analysis plays an important role in finding software flaws. Due to dynamic language features like late binding, there are many program analysis problems for which one could not assume a prior program control flow, e.g., Java points-to analysis, the disassembly of binary codes with indirect jumps, etc. In this work, we give a general formalization of such kind of on-the-fly interprocedural program analysis problems, and present a sliding-window algorithm for it in the framework of weighted pushdown systems. Our sliding window algorithm only consists of a series of local static analyses conducted on an arbitrary number of program methods, which does not sacrifice the precision of the whole program analysis at the manageable cost of caching intermediate analysis results during each iteration. We have implemented and evaluated the sliding-window algorithm by instantiating the framework with Java points-to analysis as an application. Our empirical study showed that the analysis based on the sliding-window algorithm always outperforms the whole program analysis on runtime efficiency and scalability.

1 Introduction

Program analysis plays an important role in finding software flaws. An interprocedural (or context-sensitive) program analysis distinguishes and produces analysis results for different calling contexts, whereas an intraporcedural (or context-insensitive) program analysis would confuse them and incur a loss of analysis precision. Precise interprocedural program analyses are crucial to the successful verification of software from the real-world. Due to dynamic language features like late binding, there are many program analysis problems for which one could not assume a prior program control flow, e.g., Java points-to analysis, the disassembly of binary codes with indirect jumps, etc. These analyses are known to be mutually dependent on call graph construction and the underlying system is generated on-the-fly as the analysis proceeds. It is challenging to design precise interprocedural program analyses involving heaps and dynamic language features while being scalable to large-scale software.

In this work, motivated by Java points-to analysis, we are concerned with designing practically more efficient algorithms for solving such kind of *on-the-fly interprocedural program analysis* (OTFIPA) problems. To this end, we first

give a general formalization of the analysis problem, by mildly extending the classic analysis problem for computing the meet-over-all-valid-path values, and then present a sliding-window algorithm for it that analyzes the program in pieces in isolation. Our approach adapts the powerful framework of weighted pushdown systems (WPDSs) [8], which is known as a generalized framework for interprocedural program analysis (or context-sensitive program analysis) in which method calls and returns are correctly matched with one another. Pushdown systems (PDSs) are natural formalism for modelling the interprocedural control flow of imperative programs, and WPDSs extend PDSs by associating each transition with a weight that is often encoded from a program transformer in classic dataflow analyses. Efficient algorithms have been developed for pushdown model checking by automata-theoretic approach [4], and they are carried over to WPDSs.

The major difficulties of designing an efficient algorithm for OTFIPA are that, the dependency among program parts can be cyclic, and the underlying system for analysis is enlarged on-the-fly by frequently posing dataflow queries on relevant program points. Classic solutions to tackling the first issue include, either building a dependency graph of program parts before the analysis, and analyzing program parts in their topological order after collapsing loops, or breaking such cyclic dependency by providing each program part with summary information of external program parts which it depends upon. The later solution results in modular analysis techniques which is desirable to scalable program analyses. However, it is a long-standing challenge to generate a precise procedure summary for non-trivial dataflow analysis problems. In particular, for the kind of on-the-fly program analysis problems, e.g., higher-order functions in functional programs, dynamic dispatch in Java, it is difficult to adapt classic methods for modular analysis to such occasions, as pointed out in Section 8.5 of [3].

Instead of challenging a modular analysis or collapsing loops with sacrificing the precision, we take a mild approach to improving the runtime efficiency for solving the OTFIPA problem without compromising the analysis precision. Our key idea is to generate, cache and reuse two types of intermediate analysis results that implicitly carry procedure summaries when invoking WPDS model checking as the underlying analysis engine. One is for resolving the interdependency among methods, and the other is for locally computing the whole-program analysis results without revisiting the whole program, to answer the on-the-fly dataflow queries that can be overwhelming. Notably, our algorithm is conducted in a sliding-window fashion: the analysis slides over the discovered program coverage so far, and iteratively analyzes a sized subset of methods until the accumulated analysis results from a series of local analyses stablize.

In summary, this paper makes the following contributions:

– We give a general formalization for the kind of OTFIPA problems that are mutually dependent of discovering the program coverage (Section 3.1). Such a formal clarification provides us with a basis and framework for reasoning about the correctness of our sliding-window algorithm for tackling the problem. We also show with an example of copy constant propagation that

dataflow analyses with simple conditionals can be instantiated as an instance of the problem (Section 3.2).
- We present a sliding-window algorithm for the OTFIPA problem, by adapting the inner algorithmic structures of WPDS model checking based on the $\mathcal{P}$-automata techniques (Section 4). Our analysis allows to analyze the program in pieces of an arbitrary size with preserving the precision of the whole-program analysis by caching the minimum intermediate analysis results during the analysis.
- We demonstrate experimentally the effectiveness of our approach with instantiating Java points-to analysis in the algorithmic framework (Section 5). Our preliminary empirical study shows that, the sliding-window algorithm brought a 2X speedup over the whole-program analysis for most benchmarks and successfully verify two benchmarks that exceed the time budget when running the whole-program analysis.

Last but not least, we also formally prove the correctness of our approach, for which an interested reader may wish to consult an extended version of the paper.

Related Work This work is motivated by `Japot` [6] that is a context-sensitive points-to analyzer for Java designed in the framework of WPDSs. There has been a host of work on points-to analysis. To our knowledge, almost other existing practical points-to analyzer took a cloning-based approach (that resembles inline expansion) to achieving context-sensitivity, which has an inherit limit on analyzing recursive procedural calls. We are concerned with scalable stacking-based points-to analysis algorithms for Java that precisely handles recursive procedure calls by WPDSs. In [6], the authors attempted to carefully interleave the whole program analysis with local ones on small parts of the program in a restricted manner. They used the model checker as a black box, and did not resolve the interdependency among program parts. The whole program analysis is compulsory for ensuring soundness as the final step of the analysis. By adapting the inner algorithmic structures of WPDSs, this work upgrades `Japot` to a more efficient analyzer which only consists of local static analyses yet preserves the original precision of a whole program analysis.

It is desirable to design a modular analysis by generating procedure summaries for each method and analyzing the program in pieces with instantiating the procedure summaries of callee methods. Many techniques were proposed to achieve a certain degree of modularity. However, it remains a challenge to design a precise modular analysis for context-sensitive heap analysis like Java points-to analysis [12]. Our approach is not modular analysis strictly speaking, because we never generate procedure summaries and the program parts are analyzed iteratively. Yet, the intermediate results that we generate, cache, and reuse in the analysis carry some information that are related to procedure summaries.

Our work can be turned as incremental analyses, since we cache in the analysis necessary information for conducting a local analysis on any part of the program. To our knowledge, [2] is the first work on incremental algorithms for safety

analysis of recursive state machines. Lal and Reps presented in [5] a new reachability algorithm of WPDS, and discussed how to derive an incremental algorithm from their new setting. The authors also proposed a technique to improve the running time for (weighted) pushdown model checking. Their technique could be plugged in our tool as a more efficient engine for weighted pushdown systems.

2 Preliminaries

2.1 Weighted Pushdown Model Checking

Definition 1. *A **pushdown system** (PDS) $\mathcal{P}$ is (P, Γ, Δ), where P is a finite set of control locations, Γ is a fini,te stack alphabet, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is a finite set of transition rules. A transition rule $(p, \gamma, q, \omega) \in \Delta$ is written as $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$. A **configuration** of P is a pair $\langle p, \omega \rangle$ where $p \in P$ and $\omega \in \Gamma^*$. A set of configurations C is **regular** if $\{\omega \mid \langle p, \omega \rangle \in C\}$ is regular. A transition relation $\Rightarrow$ is defined on configurations of $\mathcal{P}$, such that $\langle p, \gamma\omega' \rangle \Rightarrow \langle q, \omega\omega' \rangle$ for any $\omega' \in \Gamma^*$ if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$. Given a set C of configurations, we define $pre^*(C) = \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*(C) = \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$ which are the sets of pre-images and post-images of C, respectively.*

A pushdown system is a pushdown automaton without the input alphabet. It is known that any pushdown system can be simulated by a pushdown system for which $|\omega| \leq 2$ for each transition rule $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$ [10]. In the paper, we assume such a normalized form of pushdown systems.

Definition 2. *A **bounded idempotent semiring** $\mathcal{S}$ is $(D, \oplus, \otimes, \bar{0}, \bar{1})$, where $\bar{0}, \bar{1} \in D$, and*

1. *$(D, \oplus)$ is a commutative monoid with $\bar{0}$ as its unit element, and $\oplus$ is idempotent, i.e., $a \oplus a = a$ for all $a \in D$;*
2. *$(D, \otimes)$ is a monoid with $\bar{1}$ as the unit element;*
3. *$\otimes$ distributes over $\oplus$, i.e., for all $a, b, c \in D$, we have $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)$;*
4. *for all $a \in D, a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$;*
5. *A partial ordering $\sqsubseteq$ is defined on D such that $a \sqsubseteq b$ iff $a \oplus b = a$ for all $a, b \in D$, and there are no infinite descending chains in D.*

It is not hard to see that $\bar{0}$ is the greatest element in D.

Definition 3. *A **weighted pushdown system** (WPDS) $\mathcal{W}$ is a triplet $(\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system, $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ is a bounded idempotent semiring, and $f : \Delta \to D$ is a function that assigns a weight in D to each transition rule in Δ.*

Let $\sigma = (r_0, \ldots, r_k)$ be a transition sequence where $r_i \in \Delta$ for each $0 \leq i \leq k$. A value associated with σ is defined by $val(\sigma) = f(r_0) \otimes \cdots \otimes f(r_k)$. Given $c, c' \in Q \times \Gamma^*$, we denote by $path(c, c')$ the set of transition sequences that transform configurations from c into c' for each.

Definition 4. *Given a WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ where $\mathcal{P} = (P, \Gamma, \Delta)$, and regular sets of configurations $S, T \subseteq P \times \Gamma^*$. The model checking problem for WPDS is to compute the following value:*

$$\mathsf{WPMC}[\mathcal{W}](S,T) = \bigoplus\{val(\sigma) \mid \sigma \in path(c,c'), c \in S, c' \in T\}$$

When applying WPDSs to program analysis, the pushdown system models the interprocedural program control flow with matched calls and returns. The weights in D typically encode program transformers, $\otimes$ models (the reverse of) function composition, and $\oplus$ combines data flows at join points of the program.

2.2 Saturation-based Algorithm for WPDS Model Checking

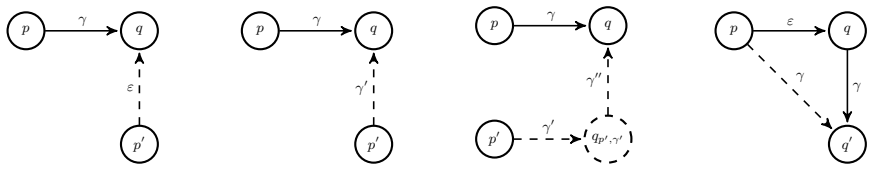
PDSs are appealing partly due to having efficient model checking algorithms based on the $\mathcal{P}$-automata techniques. A $\mathcal{P}$-automaton is a NFA (non-deterministic finite automaton) that recognizes a *regular* set of pushdown configurations.

Definition 5. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$. A $\mathcal{P}$-**automaton** $\mathcal{A} = (Q, \Sigma, \rightarrow, P, F)$ is a NFA, where $Q \supseteq P$ is a finite set of states, $\Sigma = \Gamma \cup \{\varepsilon\}$ is a finite alphabet, $\rightarrow \subseteq Q \times \Sigma \times Q$ is a set of transitions, and P and$F \subseteq Q$ are the sets of initial and final states, respectively. We define $\rightarrow^* \subseteq Q \times \Gamma^* \times Q$ as the smallest relation satisfying that, (i) $p \xrightarrow{\epsilon}{}^* p$ for any $p \in Q$; (ii) $p \xrightarrow{\gamma}{}^* p'$ if $(p, \gamma, p') \in \rightarrow$; (iii) $p \xrightarrow{\omega\gamma}{}^* p'$ if $p \xrightarrow{\omega}{}^* p''$ and $p'' \xrightarrow{\gamma}{}^* p'$ for some $p'' \in Q$. A configuration $\langle p, \omega \rangle$ is accepted by $\mathcal{A}$ if $p \xrightarrow{\omega}{}^* q$ for some $q \in F$. A set of configurations C is regular if it is accepted by some $\mathcal{P}$-automaton. We denote by $\mathcal{A}_C$ the $\mathcal{P}$-automaton that accepts a regular set C of configurations, and sometime refer to the $\mathcal{P}$-automaton by the set of transitions in it.*

One crucial property of pushdown systems is that, the set of pre-images and post-images of a *regular* set of configurations is also *regular*. Given a $\mathcal{P}$-automaton $\mathcal{A}_C$ that recognizes a regular set C of configurations. The pre-images $pre^*(C)$ and post-images $post^*(C)$ can be computed by augmenting $\mathcal{A}_C$ with new edges and states with applying backward and forward saturation rules until convergence, respectively. In the paper, we limit our focus to forward saturation and illustrate in Figure 1 the saturation rules for computing $post^*(C)$. In the figure, solid edges and states reside in the current automaton, and dashed edges and states are newly added by saturation rules.

Let l be a mapping from the edges in a $\mathcal{P}$-automaton $\mathcal{A}$ to weights. The model checking problem $\mathsf{WPMC}[\mathcal{W}](S,T)$ in Definition 4 can be solved by first forward saturating $\mathcal{A}_S$ while updating l upon stablization. Initially, $l(t) = \bar{1}$ for each transition t in $\mathcal{A}_S$, and $l(t) = \bar{0}$, otherwise. The rules for updating weights with respect to the saturation rules in Figure 1 are given as follows:
(a) $l(p', \varepsilon, q) = l(p, \gamma, q) \otimes f(r_{pop})$ (b) $l(p', \gamma', q) = l(p, \gamma, q) \otimes f(r_{normal})$
(c) $l(p', \gamma', q_{p',\gamma'}) = \bar{1}$ and $l(q_{p',\gamma'}, \gamma'', q) = l(p, \gamma, q) \otimes f(r_{push})$
(d) $l(p, \gamma, q') = l(q, \gamma, q') \otimes l(p, \varepsilon, q)$

Next, for any configuration $c = \langle p, \gamma_1 \dots \gamma_n \rangle$ in C and the saturated automaton $\mathcal{A}_{post^*(C)}$, we define $\mathcal{A}(c) = \bigoplus\{l(q_{n-1}, \gamma_n, q_n) \otimes \dots \otimes l(q_1, \gamma_2, q_2) \otimes l(p, \gamma_1, q_1) \mid$



(a) r_{pop} (b) r_{normal} (c) r_{push} (d) contraction

Fig. 1. Saturation rules for computing $post^*(C)$, where r_{pop} is $\langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle$, r_{normal} is $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle$, and r_{push} is $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$. Figure 1(d) shows a contraction rule for taking care of ε-transitions during the saturation.

$q_n \in F\}$, and for any set C of configurations, define $\mathcal{A}(C) = \bigoplus\{\mathcal{A}(c) \mid c \in C\}$. Then we have $\mathsf{WPMC}[\mathcal{W}](S, T) = \mathcal{A}_{post^*(S)}(T)$, and the algorithm for efficiently computing $\mathcal{A}(C)$ will be elaborated in Algorithm 2.

3 On-The-Fly Interprocedural Program Analysis

3.1 A Formal Description of OTFIPA

Assume abstract interpretation has been properly applied to the analysis problem if necessary. Let $((\mathcal{D}, \preceq_{\mathcal{D}}), \sqcap_{\mathcal{D}}, \top_{\mathcal{D}})$ be a meet semi-lattice tailored for the analysis, where $\sqcap_{\mathcal{D}}$ is the binary operator for computing the greatest lower bound, and $\top_{\mathcal{D}}$ is the greatest element in $\mathcal{D}$. Let $\mathcal{V} = \{V_1, \ldots, V_k\}$ be the set of variables in the analysis. An *environment* $\mathcal{E} \in \mathcal{D}^k$ of the program is a k-tuple of values in $\mathcal{D}$, and we denote by $\mathcal{E}(i)$ (or $\mathcal{E}(V_i)$) the value for the variable V_i, and extend $\sqcap_{\mathcal{D}}$ and $\preceq_{\mathcal{D}}$ to environments element-wise. The set of environments is denoted by $\mathcal{E}nv$, and the initial environment $(\top_{\mathcal{D}}, \ldots, \top_{\mathcal{D}}) \in \mathcal{D}^k$ is denoted by $\mathcal{E}_0$ which is the greatest element in $\mathcal{E}nv$. An *environment transformer* $\tau : \mathcal{E}nv \to \mathcal{E}nv$ is a map on environments that is *distributive* (thus monotonic) wrt $\sqcap_{\mathcal{D}}$, i.e., $\tau(\mathcal{E}_1 \sqcap_{\mathcal{D}} \mathcal{E}_2) = \tau(\mathcal{E}_1) \sqcap_{\mathcal{D}} \tau(\mathcal{E}_2)$. The set of environment transformers is denoted by $\mathcal{T}$. Let $(\mathcal{T}, \sqcap, \top)$ be a meet semi-lattice where $\sqcap$ is the greatest lower bound operator on $\mathcal{T}$ defined by $\tau_1 \sqcap \tau_2 = \lambda e.(\tau_1(e) \sqcap_{\mathcal{D}} \tau_2(e))$, and $\top = \lambda e.\, \mathcal{E}_0$ is the greatest element in $\mathcal{T}$.

Program analysis often first builds an interprocedural control flow graph (ICFG) of the program, and then solves the analysis problem as path problems over it. Here, we explore a so-called *supergraph* of the program for representing an ICFG. A supergraph is a collection of control flow graphs (CFG), where a CFG is constructed for a method as usual, except that each method call is represented by two nodes in the graph: a node for *call site* and a node for *return point*, and CFGs are connected in the graph by *call edges* from call sites to callee' entry points and *return edges* from callees' exits to the corresponding return points.

Let $\mathcal{M}$ be the set of methods in the program. A *supergraph* G is a triplet $(\mathcal{N}, \rightarrow_G, l)$ where $\mathcal{N} = \{\mathtt{n}_1, \ldots, \mathtt{n}_m\}$ is a set of nodes, $\rightarrow_G \subseteq \mathcal{N} \times \mathcal{N}$ is a set of edges, and $l : (\rightarrow_G) \to \mathcal{T}$ is a map that associates each edge with an environment transformer. In particular, we denote by $\mathcal{R} \subseteq \rightarrow_G$ the set of call edges, called

call relation. Given a set of methods $M \subseteq \mathcal{M}$ and a call relation $R \subseteq \mathcal{R}$, one can construct a supergraph, denoted by $G_{\downarrow_{M,R}}$.

A *valid* path in G is a path where call edges and return edges are well-matched with each other, and such valid paths constitutes some context-free language. For any node $\mathtt{n} \in \mathcal{N}$, the (possibly infinite) set of valid paths leading from $\mathtt{e_{main}}$ to $\mathtt{n}$ is denoted by $\mathsf{VPath}(\mathtt{e_{main}}, \mathtt{n})$. Let $\sigma = [t_0, \ldots, t_n]$ be a sequence of edges that forms a path in G. We define $\tau_\sigma = l(t_n) \circ l(t_{n-1}) \circ \cdots \circ l(t_0)$. Here, $\circ$ denotes the ordinary function composition. The **meet-over-all-valid-path** (MOVP) problem for G is to compute that, for each $\mathtt{n} \in \mathcal{N}$,

$$\begin{aligned} \mathsf{MOVP}[G](\mathtt{n}) &\stackrel{def}{=} \sqcap_{\mathcal{D}}\{\tau_\sigma(\mathcal{E}_0) \mid \sigma \in \mathsf{VPath}(\mathtt{e_{main}}, \mathtt{n})\} \\ &= (\sqcap\{\tau_\sigma \mid \sigma \in \mathsf{VPath}(\mathtt{e_{main}}, \mathtt{n})\})(\mathcal{E}_0) \quad \text{(By definition of } \sqcap) \end{aligned}$$

That is, it computes all the valid dataflow values flowing to each node.

We denote by $\overrightarrow{\mathsf{MOVP}[G]} \in \mathcal{E}nv^m$ an m-tuple of environments such that its i^{th} projection, denoted by $\overrightarrow{\mathsf{MOVP}[G]}[i]$ (or $\overrightarrow{\mathsf{MOVP}[G]}[\mathtt{n}_i]$), is the value $\mathsf{MOVP}[G](\mathtt{n_i})$, for each $i \in [1..m]$. Note that, since $\sqcap_{\mathcal{D}}\emptyset = \mathcal{E}_0$, we have $\overrightarrow{\mathsf{MOVP}[\emptyset]} = \mathcal{E}_0^m$. We introduce a binary relation $\sqsubseteq$ on $\mathcal{E}nv^m$ such that $\overrightarrow{\mathcal{E}}_1 \sqsubseteq \overrightarrow{\mathcal{E}}_2$ iff $\overrightarrow{\mathcal{E}}_1[i] \preceq_{\mathcal{D}} \overrightarrow{\mathcal{E}}_2[i]$ for each $i \in [1..m]$. Then $[\mathcal{E}_0, \ldots, \mathcal{E}_0] \in \mathcal{E}nv^m$ is the greatest element.

A function $\phi : \mathcal{E}nv^m \to 2^{\mathcal{M}} \times 2^{\mathcal{R}}$ is a *contract function* for G if ϕ is anti-monotonic. It characterizes dynamic program features, and its semantics is problem-specific. For instance, for Java points-to analysis, ϕ encodes the semantics of dynamic dispatch implemented in Java virtual machine, such that for any $\overrightarrow{\mathcal{E}} \in \mathcal{E}nv^m$, $\phi(\overrightarrow{\mathcal{E}})$ returns the union of methods that can be dispatched at each node $\mathtt{n}_i$ according to the value $\overrightarrow{\mathcal{E}}[\mathtt{n_i}]$, paired with the call relation for those methods. Note that, some program nodes do not matter to the change of the program coverage and may not be considered in the contract function. Provided with ϕ, we define an *enlargement function* $\eta : 2^{\mathcal{M}} \times 2^{\mathcal{R}} \to 2^{\mathcal{M}} \times 2^{\mathcal{R}}$ as follows:

$$\eta = \lambda(x, y).(x, y) \cup \phi\left(\overrightarrow{\mathsf{MOVP}[G_{\downarrow_{x,y}}]}\right)$$

where $\cup$ is extended to a pair of sets element-wise. It characterizes the process of discovering the program coverage. One can conclude with Lemma 1.

Lemma 1. *The function η is monotonic, and the least fixed point of η exists, and it coincides with $\bigcup_{j=0}^{\infty} \eta^j(\emptyset, \emptyset)$.* □

Definition 6. *Let gfp be the greatest fixed point operator, and let lfp be the least fixed point operator. An **on-the-fly interprocedural program analysis** (OTFIPA) problem is to compute* [3]

(i) the least fixed point of η, i.e., $\mathsf{lfp}(\eta)$, which is the set of methods M_r involved in the analysis problem, and a call relation R over them; and

(ii) the tuple $\overrightarrow{\mathsf{MOVP}[G_{\downarrow_{M_r,R}}]}$ of environments, which is the dataflow analysis results of solving the MOVP problem for $G_{\downarrow_{M_r,R}}$.

[3] There are datalfow analysis problems alternatively formalized over *exploded supergraph* [9], upon which one can similarly define the OTFIPA problem.

```
n0: int x, y = 0;          void foo(int b) {          void main() {
void bar(int a) {              n9: return b; }            n1: x = 2;
n4: if (x < 3)             void noop() {              n2, n3: bar(x); }
    n5, n6: y = foo(a);        n9: y = x;
else n7, n8: noop (); }        n10: x = 3; }
```

Fig. 2. A code snippet for illustrating copy constant propagation analysis.

3.2 A Running Example

This section describes an example of a *copy constant propagation* (CCP) analysis, partly following [7] and ajusts the example as an instance of OTFIPA. CCP is one of the classic dataflow analysis used for compiler optimization. The analysis is to check whether the value of a variable would remain as a constant along some program execution, so that the constant assigned to the variable can be substituted when the variable is used.

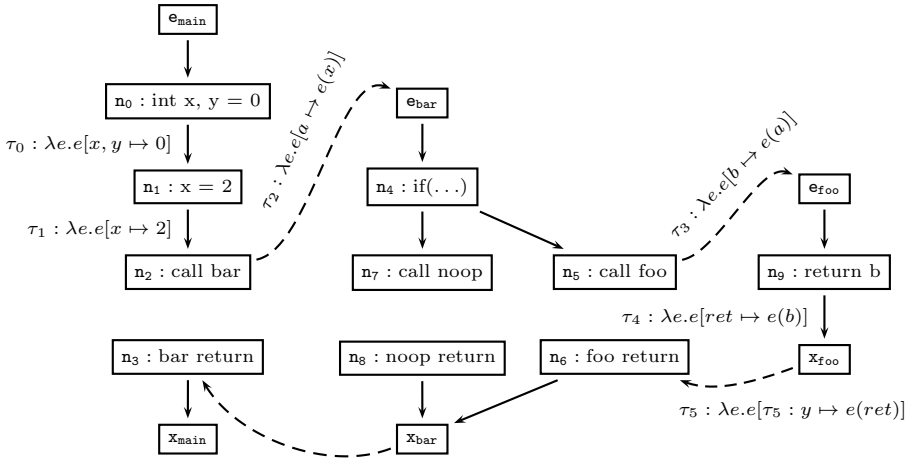


Fig. 3. A supergraph of Figure 2, where the method `noop` is omitted.

An example code snippet is shown in Figure 2, where x, y are global variables, a, b are local variables, and each statement in the code is indexed by $\mathtt{n}_i$. Its supergraph is shown in Figure 3, where solid lines are intraprocedural edges, and dotted lines are call and return edges. Since the method `noop` is not called, we omit it from the graph. Extra nodes $\mathtt{e_f}$ and $\mathtt{x_f}$ denote the unique entry and exit point of a method `f`. Each edge in the graph is labelled with an environment transformer τ. If τ is an identity function $\lambda e.e$, it is omitted in the graph. The abstract domain for the analysis is $\mathcal{D} = \mathbb{Z} \cup \{\bot, \top_{\mathcal{D}}\}$. Here, $\mathbb{Z}$ denotes integers,

$\top_{\mathcal{D}}$ denotes a value that is irrelevant and is the greatest element, and $\bot$ denotes a value that is not a constant. For all d in $\mathbb{Z}$, $\bot \preceq_{\mathcal{D}} d$ and $d \preceq_{\mathcal{D}} \top_{\mathcal{D}}$, and for any d_1, d_2 in $\mathbb{Z}$ with $d_1 \neq d_2$, d_1 and d_2 are incomparable. In the transformers, *ret* is a fresh symbol that is introduced to denote the return value of foo.

It is known to be non-trivial (often undecidable) to analyze conditionals in dataflow analysis. Therefore, conditionals are usually abstracted to be non-deterministic choices as an over-approximation of the program branching. Thus whether the method foo and noop are invoked depends on the dataflow reaching at the node $\mathtt{n}_4$, and we know that foo is reachable in this case since $\mathsf{MOVP}[G](\mathtt{n}) = [x \mapsto 2, y \mapsto 2]$. Then it can be modeled as an OTFIPA problem. For any $\overrightarrow{\mathcal{E}} \in \overrightarrow{\mathcal{D}}$,

$$\phi(\overrightarrow{\mathcal{E}}) = \begin{cases} (\{\mathtt{main}, \mathtt{bar}, \mathtt{foo}\}, R_0) & \text{if } \overrightarrow{\mathcal{E}}[\mathtt{n}_4](x) \neq \bot \text{ and } \overrightarrow{\mathcal{E}}[\mathtt{n}_4](x) < 3 \\ (\{\mathtt{main}, \mathtt{bar}, \mathtt{noop}\}, R_1) & \text{if } \overrightarrow{\mathcal{E}}[\mathtt{n}_4](x) \neq \bot \text{ and } \overrightarrow{\mathcal{E}}[\mathtt{n}_4](x) \geq 3 \\ (\{\mathtt{main}, \mathtt{bar}, \mathtt{foo}, \mathtt{noop}\}, R_2) & \text{if } \overrightarrow{\mathcal{E}}[\mathtt{n}_4](x) = \bot \\ (\emptyset, \emptyset) & \text{if } \overrightarrow{\mathcal{E}}[\mathtt{n}_4](x) = \top_{\mathcal{D}} \end{cases}$$

where $R_0 = \{(\mathtt{main}, \mathtt{bar}), (\mathtt{bar}, \mathtt{foo})\}$, $R_1 = \{(\mathtt{main}, \mathtt{bar}), (\mathtt{bar}, \mathtt{noop})\}$, $R_2 = R_0 \cup R_1$. It is not hard to see that ϕ is anti-monotonic. In particular, the third case above corresponds to abstracting the conditional as a non-deterministic choice when $\overrightarrow{\mathcal{E}}[\mathtt{n}_4](x) = \bot$.

4 Algorithms for the OTFIPAProblem

4.1 A Whole-Program Analysis Algorithm

We can solve the OTFIPA problem by using WPDS as the underlying program analysis engine. Given a supergraph $G = (\mathcal{N}, \rightarrow_G, l)$. We can define a WPDS $\mathcal{W}_G = ((P, \Gamma, \Delta), \mathcal{S}, f)$, where there is a unique control location $\star$, i.e, $P = \{\star\}$; the stack alphabet is the set of nodes in G, i.e., $\Gamma = \mathcal{N}$; and Δ is constructed by the follows rules:

- $\langle \star, \mathtt{n} \rangle \hookrightarrow \langle \star, \mathtt{n}' \rangle \in \Delta$, if there exists an intraprocedural edge $(\mathtt{n}, \mathtt{n}') \in \rightarrow_G$.
- $\langle \star, \mathtt{n} \rangle \hookrightarrow \langle \star, \mathtt{e_f}\, \mathtt{n}' \rangle \in \Delta$, if there exists a call edge $(\mathtt{n}, \mathtt{e_f}) \in \rightarrow_G$, where $\mathtt{n}'$ is the return point matching the call site $\mathtt{n}$, and $\mathtt{e_f}$ is the entry of the callee $\mathtt{f}$.
- $\langle \star, \mathtt{x_f} \rangle \hookrightarrow \langle \star, \epsilon \rangle \in \Delta$, if there exists a return edge $(\mathtt{x_f}, \mathtt{n}) \in \rightarrow_G$ for some $\mathtt{n}$, where $\mathtt{x_f}$ is the exit of the callee $\mathtt{f}$.

The environment transformers associated with edges are directly modelled as weights. To form the semiring $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$, we define the commutative monoid $(D, \oplus)$ to be $(\mathcal{T}, \sqcap, \top)$, $\bar{0}$ is taken as $\top$, $\otimes$ is defined as the reverse of function composition, and $\bar{1}$ is taken as an identity transformer $\lambda e.e$. We denote by $\mathtt{GenWPDS}(G)$ the above procedure that generates a WPDS for G. Given the encoding above, we can conclude with the following fact.

Fact 7 *For each node $n \in \mathcal{N}$, we have that*

$$\mathit{MOVP}[G](n) = \mathit{WPMC}[\mathcal{W}_G](S, T)(\mathcal{E}_0)$$

, where $S = \{\langle \star, e_{main} \rangle\}$ and $T = \{\langle \star, n\omega \rangle \mid \omega \in \Gamma^*\}$, respectively.

$\langle \star, \mathtt{e}_{\mathtt{main}} \rangle \hookrightarrow \langle \star, \mathtt{n}_0 \rangle$ $\quad$ $\langle \star, \mathtt{e}_{\mathtt{bar}} \rangle \hookrightarrow \langle \star, \mathtt{n}_4 \rangle$

$\langle \star, \mathtt{n}_0 \rangle \overset{\tau_0}{\hookrightarrow} \langle \star, \mathtt{n}_1 \rangle$ $\quad$ $\langle \star, \mathtt{n}_4 \rangle \hookrightarrow \langle \star, \mathtt{n}_5 \rangle$ $\quad$ $\langle \star, \mathtt{e}_{\mathtt{foo}} \rangle \hookrightarrow \langle \star, \mathtt{n}_9 \rangle$

$\langle \star, \mathtt{n}_1 \rangle \overset{\tau_1}{\hookrightarrow} \langle \star, \mathtt{n}_2 \rangle$ $\quad$ $\langle \star, \mathtt{n}_5 \rangle \overset{\tau_3}{\hookrightarrow} \langle \star, \mathtt{e}_{\mathtt{foo}}\, \mathtt{n}_6 \rangle$ $\quad$ $\langle \star, \mathtt{n}_9 \rangle \overset{\tau_4}{\hookrightarrow} \langle \star, \mathtt{x}_{\mathtt{foo}} \rangle$

$\langle \star, \mathtt{n}_2 \rangle \overset{\tau_2}{\hookrightarrow} \langle \star, \mathtt{e}_{\mathtt{bar}}\, \mathtt{n}_3 \rangle$ $\quad$ $\langle \star, \mathtt{n}_6 \rangle \hookrightarrow \langle \star, \mathtt{x}_{\mathtt{bar}} \rangle$ $\quad$ $\langle \star, \mathtt{x}_{\mathtt{foo}} \rangle \overset{\tau_5}{\hookrightarrow} \langle \star, \epsilon \rangle$

$\langle \star, \mathtt{n}_3 \rangle \hookrightarrow \langle \star, \mathtt{x}_{\mathtt{main}} \rangle$ $\quad$ $\langle \star, \mathtt{x}_{\mathtt{bar}} \rangle \hookrightarrow \langle \star, \epsilon \rangle$

Fig. 4. WPDS transition rules encoded for the example in Figure 2 where the transition rules encoded for the method `noop` are omitted.

By Lemma 1 and Fact 7, one may solve the OTFIPA problem as the limit of $\{\eta^j(\emptyset, \emptyset) \mid j \geq 0\}$ by iteratively calling WPDS model checker for solving the MOVP problem on $G_{\downarrow_{\eta^j(\emptyset,\emptyset)}}$ (that is the supergraph for all the methods currently discovered to be involved in the analysis problem up to the j^{th} iteration).

Example 1. The encoded WPDS transition rules of the code snippet in Figure 2 is given in Figure 4 that are grouped method-wise. We denote the system by $\mathcal{W}_{ccp}$. Let $S = \{\langle \star, \mathtt{e}_{\mathtt{main}} \rangle\}$ and let $T = \{\langle \star, \mathtt{x}_{\mathtt{main}} \omega \rangle \mid \omega \in \Gamma^*\}$ be the source and target configurations. We have $\mathsf{WPMC}[\mathcal{W}_{ccp}](S, T) = \lambda e.e[x \mapsto \bot, y \mapsto 2]$ which computes the dataflow values from the entry to the exit of main, by abstracting the program branchings as non-deterministic choices. By applying the result to the initial environment $\mathcal{E}_0$, we obtain $[x \mapsto \bot, y \mapsto 2]$ which says that x is not a constant and y is a constant at the exit of main. As far as the OTFIPA problem is concerned when the conditional at $\mathtt{n}_4$ is $x < 3$, we obtain a different analysis result $[x \mapsto 2, y \mapsto 2]$ that, both x and y are constants at the exit of `main`. It is an artificially coined example, yet shows how a OTFIPA problem differs with an ordinary program analysis problem.

4.2 A Sliding-Window Analysis Algorithm

This section presents a sliding-window algorithm for OTFIPA by adapting the inner algorithmic structure of WPDS model checking. As given in Algorithm 1, Line 1 declares those global data structures that are updated through each iteration, where $\overrightarrow{d} \in \mathcal{T}^m$ is the m-tuple of environment transformers for all the program points; M_r is the set of reachable methods and R is the call relation to be discovered by the analysis, respectively, and they are the analysis results of solving the OTFIPA problem upon the algorithm terminates; δ_R records transitions relevant to return values of the callees and δ_S records transitions of summary values propagated from the calling methods, and they are intermediate analysis results cached and reused through the iterations.

Definition 8. *Mark variables in Algorithm 1 with superscript iteration numbers to denote their values at the entry of the while loop in that iteration. A function* `Schdule`$: 2^{\mathcal{M}} \times \mathbb{N} \to 2^F$ *is a* ***scheduler*** *for Algorithm 1 if, for each $M \subseteq \mathcal{M}$ and $i \in \mathbb{N}$,* `Schdule`$(M, i) \subseteq M_r \cap \{f \in \mathcal{M} \mid \bigcup_{1 \leq j \leq i} f.checked^{(j)} = 0\}$. *A scheduler is* ***fair*** *if, for each $i > 0$, there exists $f \in M_r^{(i)}$ with $f.checked^{(i)} = 0$, then there exists $j > i$ with $f \in M_w^{(j)}$.*

Algorithm 1: `SwaOTFIPA`$(\mathcal{M}, \phi)$: A Sliding Window Algorithm for OTFPA

```
1  M_r := ∅; R := ∅; d⃗ := [0̄, ..., 0̄]; δ_R = ∅; δ_S := ∅; l := λt.0̄; iteration := 0;
2  foreach f ∈ M do f.checked := 0;
3  while (not ∀f ∈ M_r. f.checked = 1) do
4      ℰ⃗ := d⃗(ℰ_0);
5      (M_r, R) := φ(ℰ⃗);
6      M_w := Schdule(M_r, iteration++) ;
7      (δ_w, l) := SatPost(GenWPDS(G_{↓M_w,R}), δ_R, l);
8      (d⃗_w, δ_S, l) := GenValue(δ_w ∪ δ_S, l);
9      δ_R := δ_R ∪ {(q, ϵ, q') | (q, ϵ, q') ∈ δ_w};
10     foreach f in M_w do f.checked := 1;
11     UpdatedNode := {n_i ∈ N | ∃i ∈ [1..m]. d⃗[i] ≠ d⃗_w[i]};
12     foreach f ∈ M_r \ M_w do
13         if DepMeth(f) ∩ UpdatedNode ≠ ∅ then f.checked := 0;
14     ℰ⃗_w := d⃗_w(ℰ_0); M_r := M_r ∪ φ(ℰ⃗_w); d⃗ := d⃗ ⊕ d⃗_w;
15 return (M_r, R, d⃗);
```

Each method $f \in \mathcal{M}$ is designated with a boolean variable *checked*, and $f.checked = 0$ means that f has to be analyzed in the analysis, and it is not necessarily to be included in the next iteration, otherwise. Initially, each method $f \in \mathcal{M}$ is declared to be unchecked (Line 2). If there remains any method $f \in M_r$ to be analyzed (Line 3), then the while loop will repeat. Line 4 applies $\overrightarrow{d}$ to the initial environment $\mathcal{E}_0$ and returns the current program environments $\overrightarrow{\mathcal{E}}$. Then by applying the contract function to $\overrightarrow{\mathcal{E}}$ at Line 5, the methods and the call relation currently discovered can be obtained. At Line 6, the scheduler (formally given in Definition 8) takes a set of methods M_w from M_r, called a *sliding window*, to be analyzed in the iteration, such that $f.checked = 0$ for each $f \in M_w$. At Line 7, the algorithm generates a WPDS $\mathcal{W}$ for the supergraph $G_{\downarrow M_w,R}$, and forward saturates δ_R, i.e., to compute $post^*[\mathcal{W}](\delta_R)$ given the mapping l. Here, we denote by `SatPost` the backward saturation procedure conducted on a weighted $\mathcal{P}$-automaton described in Section 2.2. After Line 7, we obtain a $\mathcal{P}$-automaton δ_w and the updated mapping l from automata transitions to their weights. Based on the results, we are ready to read out the result $\overrightarrow{d}_w$ from the weighted automaton $\delta_w \cup \delta_S$ for solving the MOVP problem for the sliding window, by invoking the subprocedure `GenValue` at Line 8. It also returns the updated summary transitions δ_S along with the updated label l.

At Line 9, δ_R is augmented with those newly-introduced in δ_w that are all resulted from pop transitions. Each method $f \in M_w$ is marked as checked at Line 10. At Line 11, the set $UpdatedNode$ is collected for which the program environments (or environment transformers) are updated by the current analysis. Line 12 to 13 pinpoint the set of methods that would be affected by the new analysis results. Here, $DepMeth(f)$ collects the set of nodes in the supergraph

Algorithm 2: `GenValue`(δ, l): Generating the Analysis Result for OTFIPA

```
1  let B = (Q, Γ, δ, P, {q_f}) be the
     P-automaton constructed from δ;
2  let V : Q → D be a mapping;
3  foreach q ∈ Q \ {q_f} do V(q) := 0̄;
4  V(q_f) := 1̄; ws := {q_f};
5  while ws ≠ ∅ do
6      select and remove q from ws;
7      foreach t = (q', γ, q) ∈ δ do
8          if q' ∉ P then
9              new := V(q') ⊕ (V(q) ⊗ l(t));
10             if new ≠ V(q') then
11                 V(q') := new;
12                 ws := ws ∪ {q'};
13 foreach q ∈ Q \ (P ∪ {q_f}) do
14     δ := δ ∪ {(q, *, q_f)};
15     l(q, *, q_f) := V(q);
16 δ_S := δ ∩ (Q × {*} × {q_f});
17 d⃗ := [0̄, ..., 0̄];
18 foreach (p, γ, q) ∈ δ with p ∈ P do
19     d⃗[(p, γ)] :=
         d⃗[(p, γ)] ⊕ (V(q) ⊗ l(q, γ, q'))
20 return (d⃗, δ_S, l)
```

that are source ends of the incoming edges into the nodes in the CFG of f. Line 14 returns the newly-computed environment for the program nodes in the sliding window, returns the updated set of reachable methods, and unify $\overrightarrow{d}$ with the newly-updated value $\overrightarrow{d}_w$ by extending $\oplus$ to m-tuple element-wise.

Note that, the program is analyzed method-wise and each sliding window consists of a set of methods. To decouple the interprocedural program analysis into intrapocedural counterparts, we slightly modify the procedure `GenWPDS` such that, for any transition $r : \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$ that encodes some call edge, we split it into two transitions as follows:

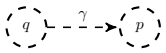


Fig. 5. A new forward saturation rule for $r : \langle p, \varepsilon\rangle \hookrightarrow \langle q, \gamma\rangle$ with $l(q, \gamma, p) = f(r)$. Dashed lines and nodes will be added into the automaton in question.

$$r_{caller} : \langle p, \gamma\rangle \hookrightarrow \langle q_{p',\gamma'}, \gamma''\rangle \text{ and } r_{callee} : \langle q_{p',\gamma'}, \varepsilon\rangle \hookrightarrow \langle p', \gamma'\rangle$$

and for any transition in the form of $\langle p, \varepsilon\rangle \hookrightarrow \langle q, \gamma\rangle$, we have $\langle p, \omega\rangle \Rightarrow \langle q, \gamma\omega\rangle$ for any $\omega \in \Gamma^*$. The transition r_{caller} belongs to the caller, and r_{callee} belongs to the callee, with $f(r_{caller}) = f(r)$ and $f(r_{callee}) = \bar{1}$. The forward saturation rule for r_{caller} is the same as the one for r_{normal} given in Figure 1 (b), and the new rule for r_{callee} is shown in Figure 5. Besides, let $\mathcal{A}_S$ be the $\mathcal{P}$-automaton that recognizes the set S of source configurations. We add a set of new transitions into the entry method (i.e., *main*) for encoding the transitions in $\mathcal{A}_S$ in the sliding-window analysis. For each transition (q, γ, q') in $\mathcal{A}_S$, we prepare the new transition $r : \langle q', \varepsilon\rangle \hookrightarrow \langle q, \gamma\rangle$ and add it to the WPDS transitions encoded from the entry method with $f(r) = \bar{1}$.

The subprocedure `GenValue` is given in Figure 2. It is centered around computing a mapping V from the automata states to weights. Intuitively, $V(q)$ is to store the weight $\mathcal{A}(\mathcal{L}(\mathcal{A}, q))$ (Recall that $\mathcal{A}(C)$ is defined in Section 2.2 for

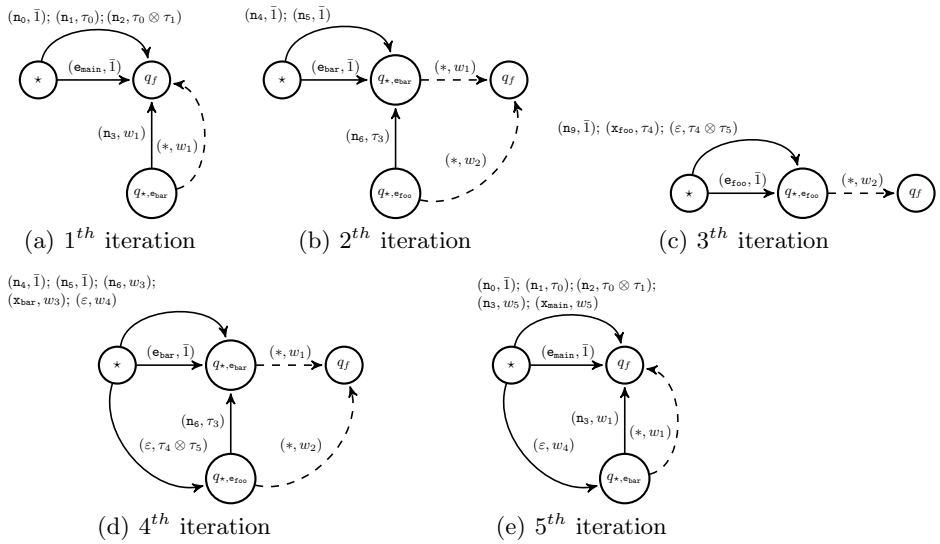


Fig. 6. A sliding-window analysis for the example in Figure 2, where $w_1 = \tau_0 \otimes \tau_1 \otimes \tau_2$; $w_2 = w_1 \otimes \tau_3$; $w_3 = \tau_3 \otimes \tau_4 \otimes \tau_5$; $w_4 = w_3$; $w_5 = w_1 \otimes w_4$

a regular set C of configurations). Initially, $V(q_f) = \bar{1}$ for the final state, and $V(q) = \bar{0}$ for any other state q, and the workset ws is set as $\{q_f\}$ (Line 3-4). The while loop (Line 5-12) will repeat if the ws is not empty . For each state in the workset, the algorithm backward propagates the weights (Line 9) and updates the weights on each state (Line 11) until no more updates are possible. Line 13-14 compute and update the summary transitions. A summary transition $(q, *, q_f)$ can be regarded as an edge for the transitive closure of any path leading from q to q_f in the automaton, and $l(q, *, q_f)$ combines weights along those paths. Line 18-19 finally read out the analysis result for each node in the supergraph. Note that, here we do not assume P is a singleton set, and the algorithm also works for a more general setting when one may take a different encoding of WPDSs.

Theorem 1. *Suppose Algorithm 1 is called with a fair scheduler, and the OTFIPA problem can be encoded into a WPDS model checking problem. Then Algorithm 1 terminates and returns the results of solving the OTFIPA problem.* □

Example 2. In Figure 6, we illustrate how to conduct a sliding-window analysis for the OTFIPA problem in Figure 2. The analysis consists in five iterations, and the figures show the weighted $\mathcal{P}$-automaton constructed in each iteration after Line 9 in Algorithm 1, respectively. An edge t in the automaton is labelled with a pair (γ, w) of the alphabet symbol γ and its weight, i.e., $l(t)$. Suppose that a single method is analyzed for each iteration, i.e., the size of a sliding window is set to be $|M_w| = 1$.

The algorithm starts with analyzing the method *main* (Figure 6 (a)), and generates a summary edge shown in the dashed line. Since $\overrightarrow{d}[\mathtt{n_2}]$ is updated,

the method *bar* would be affected. Then it analyzes *bar* in Figure 6 (b). Here, one has to know the program environment at $\mathtt{n}_4$ to judge which conditional brach should be taken. Thanks to caching the summary edge $(q_{\star,\mathtt{e}_{\mathtt{bar}}}, (*, w_1), q_f)$ that is generated in (a), one can read out the current analysis result $\overrightarrow{d}[\mathtt{n}_4] = [x \mapsto 2, y \mapsto 0]$ in (b), and knows that the method foo will be invoked. Since $\overrightarrow{d}[\mathtt{n}_5]$ is updated, the method foo would be affected and is analyzed in Figure 6 (c). Since $\overrightarrow{d}[\mathtt{x}_{\mathtt{foo}}]$ is updated, the calling method *bar* would be affected and is analyzed again in Figure 6 (d), where the pop transition $(\star, (\varepsilon, \tau_4 \otimes \tau_5), q_{\star,\mathtt{e}_{\mathtt{foo}}})$ that is newly-generated in (c) is stored into the automaton before the saturation procedure starts. After the analysis at this iteration, $\overrightarrow{d}[\mathtt{x}_{\mathtt{bar}}]$ is updated, then its caller *main* would be affected and is analyzed again in Figure 6 (e), where the pop transition $(\star, (\varepsilon, w_4), q_{\star,\mathtt{e}_{\mathtt{bar}}})$ that is newly-generated in (d) is stored into the automaton before the saturation procedure starts. Finally, we can read out the analysis result $\overrightarrow{d}[\mathtt{x}_{\mathtt{main}}] = [x \mapsto 2, y \mapsto 2]$ at the program exit $\mathtt{x}_{\mathtt{main}}$, which tells that both x and y are constants through the program execution.

5 Experiments

We developed a points-to analyser for Java called `mJapot` by instantiating the `SwaOTFIPA` algorithm, following the context-sensitive, field-sensitive, and flow-insensitive Java points-to analysis by WPDS in [6]. In our analysis, we used and extended the WPDS model checker jMoped [4] as the backend analysis engine, for computing forward saturations and reading out analysis results in each sliding window analysis. We use Soot 2.5.0 [11] for preprocessing from Java to Jimple codes which our points-to analyzer was built upon. We evaluate `mJapot` on the Ashes benchmark suite [5] and the DaCapo benchmark suite [1] given in the #App. column in Table 5. These applications are de facto benchmarks when evaluating Java points-to analysis. We analyze DaCapo benchmark with JDK 1.5, and Ashes benchmarks for which JDK 1.3 suffices. All experiments were performed on a Mac OS X v.10.9.2 with 1.7 GHz Intel Core i7 processor, and 8GB RAM. A 4GB RAM is allocated for Java virtual machine.

To measure the performance of points-to analysis, we take *call graph generation* in terms of reachable methods as client analysis. Table 1 shows the preliminary experimental results. The number of reachable methods is given in the "# Methods" column with taking Java libraries into account. The sub-column " `CHA`" is the result by conducting CHA of Spark in soot-2.5.0. The sub-column "`mJapot`" gives results computed by our incremental points-to analysis, and the "# Statements" column gives the number of Jimple statements that `mJapot` analyzed. The "# `WPA`" and "# `SWA`" columns give the time in seconds of the whole-program analysis and sliding-window analysis, respectively. In the table, k is the size of the sliding window, i.e., the number of the methods analyzed in each iteration.

[4] https://www7.in.tum.de/tools/jmoped/
[5] http://www.sable.mcgill.ca/ashes

# App.	# WPA(s)	# SWA(s)			# Acc.	# Methods		# Stmts
		$k = \infty$	$k = 5000$	$k = 3000$		CHA	mJapot	(mJapot)
soot-c	250	153	140	**139**	1.8	5460	5079	83,936
sablecc-j	616	**194**	209	204	3.2	13,055	9068	144,584
antlr	656	365	390	**348**	1.9	10,728	9133	156,913
pmd	669	**313**	478	332	2.1	12,485	10,406	180,170
hsqldb	350	186	**175**	180	2.0	9983	8394	142,629
xalan	385	**176**	185	193	2.2	9977	8392	141,415
luindex	438	190	**189**	202	2.3	10,596	8961	152,592
lusearch	436	219	**216**	227	2.0	11,190	9580	163,958
eclipse	767	**382**	455	383	2	12,703	10,404	179,539
bloat	_	**4748**	4894	4778	> 1.5	12,928	11,090	194,063
jython	_	4633	4857	**2924**	> 2.5	14,603	12,033	202,326
chart	_	_	_	_	_	30,831	_	_

Table 1. Comparison between the whole-program analysis and the SwaOTFIPA-based sliding-window analysis, where _ means time out (> 2 hours).

We set a bound k on the number of methods of each sliding window, shown in the sub-column "$k = \infty$", "$k = 5000$" and "$k = 3000$", respectively, where $k = \infty$ means that we take all methods from the current workset for the analysis. We show the smallest number in bold type. As shown in the "# Acc." column, over all the experiments we performed, SWA provided us an average 2X speedup over WPA. Note that, it performs almost the same when the size of sliding window changes for most benchmarks expect for "jython", which indicates that the algorithm can be useful for analyses having a limited memory budget. Besides, the number of reachable methods detected by CHA are reduced by 16% using mJapot. Note that mJapot is more efficient than Japot [6] because the backend model checker was changed from the one in C to the one in Java. Since the frontend analyzer is implemented in Java, it reduced the huge disk IO time for exchanging information between the model checker and the analyzer.

6 Conclusion

We studied the OTFIPA problems, for which one could not assume a prior interprocedural control flow of the program, and therefore, the discovery of the program coverage is often mutually dependent on the analysis problem, such as Java points-to analysis. We give a general formalization of the OTFIPA problem, and present a sliding-window algorithm for it. Our algorithm is conducted in a sliding window fashion that iteratively analyzes the program in an arbitrary set of methods, which can be useful for analysis having a tight memory budget. We implemented the algorithm and evaluated it with a context-sensitive points-to analysis for Java. The preliminary empirical study confirmed the effectiveness of our approach.

Acknowledgment

We would like to thank anonymous referees for useful comments. The work has been partially supported by Shanghai Pujiang Program (No. 17PJ1402200), the JSPS KAKENHI Grant-in-Aid for Scientific Research(B) (15H02684) and the JSPS Core-to-Core Program (A. Advanced Research Networks).

References

1. Blackburn, S.M., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications. pp. 169–190. OOPSLA '06, ACM Press, New York, NY, USA (2006)
2. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In: Proceedings of the 17th International Conference on Computer Aided Verification. pp. 449–461. CAV'05 (2005)
3. Cousot, P., Cousot, R.: Modular static program analysis. In: Horspool, R.N. (ed.) Compiler Construction, 11th International Conference, CC 2002. vol. 2304, pp. 159–178. Springer (2002), https://doi.org/10.1007/3-540-45937-5_13
4. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Proceedings of the 12th International Conference on Computer Aided Verification. pp. 232–247. CAV '00, Springer-Verlag, London, UK (2000), http://dl.acm.org/citation.cfm?id=647769.734087
5. Lal, A., Reps, T.: Improving pushdown system model checking. In: Proceedings of the 18th International Conference on Computer Aided Verification. pp. 343–357. CAV'06, Springer-Verlag, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11817963_32
6. Li, X., Ogawa, M.: Stacking-based context-sensitive points-to analysis for java. In: Proceedings of the 5th international Haifa verification conference on Hardware and software: verification and testing. pp. 133–149. HVC'09, Springer-Verlag, Berlin, Heidelberg (2011)
7. Reps, T.W., Lal, A., Kidd, N.: Program analysis using weighted pushdown systems. In: FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference. pp. 23–51 (2007), http://dx.doi.org/10.1007/978-3-540-77050-3_4
8. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. 58(1-2), 206–263 (2005)
9. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comput. Sci. 167(1-2), 131–170 (Oct 1996), http://dx.doi.org/10.1016/0304-3975(96)00072-2
10. Schwoon, S.: Model-Checking Pushdown Systems. Ph.D. thesis (2002)
11. Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing java bytecode using the soot framework: Is it feasible? In: Watt, D.A. (ed.) Compiler Construction: 9th International Conference. pp. 18–34. CC '00, Springer, Berlin, Heidelberg (2000), https://doi.org/10.1007/3-540-46423-9_2
12. Yorsh, G., Yahav, E., Chandra, S.: Generating precise and concise procedure summaries. In: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 221–234. POPL '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1328438.1328467