Computer Sciences Department

Solving Multiple Dataflow Queries Using WPDSs

Akash Lal
Thomas Reps

Technical Report #1632

March 2008

UNIVERSITY OF
WISCONSIN
MADISON

Solving Multiple Dataflow Queries Using WPDSs

Akash Lal[1] and Thomas Reps[1,2]

[1] University of Wisconsin; Madison, Wisconsin; USA. {akash, reps}@cs.wisc.edu
[2] GrammaTech, Inc.; Ithaca, NY; USA.

Abstract. A dataflow query asks for the set of reachable (abstract) states, given a starting set of states. In this paper, we show how to optimize multiple queries on the same program (each with a different starting set of states) for better overall running time. After a preprocessing phase, we obtain an asymptotic improvement in answering dataflow queries. We use *weighted pushdown systems* as the abstract model of a program. Our techniques are interprocedural. They are general, yet provide an impressive speedup. We applied our algorithm to three very different applications, one based on finding affine relations using linear algebra, and others for model checking Boolean programs, and obtained 1.5-fold to 7-fold speedups.

1 Introduction

Dataflow analysis is concerned with approximating program behavior. A *dataflow query* asks for the set of (abstract) program states (forward or backward) reachable from a given starting set of states, where a state is a $\langle \text{program location}, \text{data store} \rangle$ pair. One common application of (forward) dataflow analysis is to pose a single dataflow query from the initial state in which program execution starts. This produces an over-approximation of all program states that may arise during its execution. However, in certain situations, multiple dataflow queries need to be posed on the same program, each with a different starting set of states.

One such need arises in the analysis of concurrent programs, in the method presented in [14], which tracks program evolution for a bounded number of context switches. Here, a concurrent program consists of a set of threads that communicate via shared memory. For a thread t of interest, the environment (consisting of the other threads) is only given control a fixed number of times. Each time, the environment can change the state of shared memory, thus affecting the execution of thread t. The analysis of such programs requires multiple dataflow queries to be posed on t. Whenever the environment changes the state of shared memory, a new query is posed on t, starting from this state (paired with the local state of t).

Multiple queries are also useful for program understanding, e.g., to find out the net effect of executing from one statement to another (to find dependences between them). Finding a loop summary for each loop is another example. Our applications (§5), for which we developed the techniques of this paper, are based on these examples.

Answering multiple queries on the same program independently from each other usually involves repeated work. The alternative is to do some preprocessing and compute certain basic facts about the program that can be reused each time a new dataflow query is posed. In this paper, we exploit this observation to improve the running time needed for answering multiple dataflow queries on the same program.

At the intraprocedural level, this work is inspired by our previous result on speeding up a single dataflow query. In [10], we showed how to use Tarjan's path sequence algorithm [24], which computes regular expressions to represent a set of paths in a graph, to obtain a faster algorithm for the fixpoint computation needed to answer a single dataflow query. Answering multiple queries poses a different problem than speeding up a single query: its goal is to avoid *repeating* the fixpoint computation for each query. However, this paper shows that the information computed by the Tarjan's algorithm is still useful. We use it as a preprocessing step for speeding up multiple intraprocedural queries.

At the interprocedural level, a set of program paths can no longer be captured with a regular expression (the set may be a context-free language). We develop new techniques to address this complication: we show what preprocessing can be done to avoid recomputation across procedure boundaries, and how to isolate the intraprocedural computation to be able to use our intraprocedural algorithm.

Overall, with our techniques, the preprocessing is quite efficient, usually faster than solving two dataflow queries. After preprocessing, we obtain asymptotic improvements in answering each dataflow query (for programs whose control structure is mostly reducible), and only require iteration to a fixpoint when the starting set of states is infinite (i.e., in other cases, we do not need to go around program loops or recursive procedures). Our experiments show that this approach is advantageous even if as few as two queries need to be answered.

Our approach applies to any dataflow-analysis problem in which one has a domain of distributive dataflow-transfer functions closed under composition [23, 8]. Some examples can be found in [19, 18, 12]. This paper mainly presents the work using the framework of *weighted pushdown systems* (WPDSs) [19], which generalize previous work on interprocedural analysis frameworks [23, 9, 17]. However, we also sketch how variants of the technique can be incorporated in solvers that work over interprocedural control-flow graphs (ICFGs); see §4 and App. A.

The number of states of a WPDS can be infinite. For this reason, sets of states are represented symbolically using *weighted automata* [19]. Our algorithms must be able manipulate these automata because they are essential for applications like the one presented in [14], which is the subject of one of our experiments. In §4, we show how to speed up multiple queries given such automata representing the starting set of states.

The contributions of this paper can be summarized as follows:

- We show how information computed by Tarjan's path sequence algorithm can be used to obtain asymptotic improvements in answering multiple intraprocedural queries (§3).
- We give a new WPDS reachability algorithm for answering interprocedural queries that carries over the above asymptotic improvements (§4).
- We sketch variants of the technique that allow the ideas to be applied in other standard dataflow-analysis frameworks (§4 and App. A).
- We applied our techniques to three different applications (§5), and measured 1.5-fold to 7-fold speedups over previous techniques, including optimized ones [10].

The rest of the paper is organized as follows: §2 gives background on WPDSs. §3 presents our algorithm for the intraprocedural case, and §4 generalizes it to the interprocedural case (WPDSs). §5 reports experimental results. §6 discusses related work. App. A presents a version of our algorithm that works with ICFGs. App. B contains a proof of correctness of the algorithm in §4. In this paper, we focus on forward dataflow queries. The algorithm for backward queries is given in App. C.

2 Program Model

Definition 1. *A* **pushdown system** *is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where P is the set of states or control locations, Γ is the set of stack symbols and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is the set of pushdown rules. A* **configuration** *of $\mathcal{P}$ is a pair $\langle p, u \rangle$ where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$ where $p, p' \in P$, $\gamma \in \Gamma$ and $u \in \Gamma^*$. These rules define a transition relation $\Rightarrow$ on configurations of $\mathcal{P}$ as follows: If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$ then $\langle p, \gamma u' \rangle \Rightarrow \langle p', u u' \rangle$ for all $u' \in \Gamma^*$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.*

Without loss of generality, we restrict PDS rules to have at most two stack symbols on the right-hand side [22]. The standard approach for modeling program control flow with a pushdown system is as follows: P contains a single state $\{p\}$, Γ corresponds to program locations, and Δ corresponds to transitions in the interprocedural control-flow graph (ICFG)[3]: A CFG edge $u \rightarrow v$ is modeled by a PDS rule $\langle p, u \rangle \hookrightarrow \langle p, v \rangle$; A call to procedure g at location l that returns to r as $\langle p, l \rangle \hookrightarrow \langle p, g_{\text{enter}}\ r \rangle$; and a return from procedure g as $\langle p, g_{\text{exit}} \rangle \hookrightarrow \langle p, \varepsilon \rangle$. (Single-state PDSs are also referred to as context-free processes [4].) In such an encoding, a PDS configuration $\langle p, \gamma_1\ \gamma_2 \cdots \gamma_n \rangle$ stores the value of the program counter γ_1 and the stack of return addresses for unfinished calls as $\gamma_2, \gamma_3, \cdots, \gamma_n$ (in order).

A *weighted* pushdown system is obtained by supplementing a pushdown system with a weight domain that is a *bounded idempotent semiring* [19, 2]. Such semirings are capable of encoding a number of abstractions [18]. WPDSs can

[3] An ICFG is a set of CFGs, one for each procedure, with additional edges going from a call-site to the entry node of the callee and from its exit node to the return site.

encode the IFDS framework [17], and other dataflow analyses; see [19, 10] for more details.

Definition 2. *A* ***bounded idempotent semiring*** *(or "weight domain") is a tuple* $(D, \oplus, \otimes, \overline{0}, \overline{1})$*, where* D *is a set of* **weights**, $\overline{0}, \overline{1} \in D$*, and* $\oplus$ *(combine) and* $\otimes$ *(extend) are binary operators on* D *such that*

1. $(D, \oplus)$ *is a commutative monoid with* $\overline{0}$ *as its neutral element, and where* $\oplus$ *is idempotent.* $(D, \otimes)$ *is a monoid with the neutral element* $\overline{1}$*.*
2. $\otimes$ *distributes over* $\oplus$*, i.e., for all* $a, b, c \in D$ *we have* $$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c) \, .$$
3. $\overline{0}$ *is an annihilator with respect to* $\otimes$*, i.e., for all* $a \in D$, $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$*.*
4. *In the partial order* $\sqsubseteq$ *defined by* $\forall a, b \in D,\ a \sqsubseteq b$ *iff* $a \oplus b = a$*, there are no infinite descending chains.*

One may think of weights as dataflow transformers, extend as transformer composition, combine as *meet*, $\overline{0}$ as the transformer for an infeasible path, and $\overline{1}$ as the identity transformer. For example, if D_f is a finite set of dataflow facts, then the weight domain $(2^{D_f \times D_f}, \cup, \circ, \emptyset, \mathrm{id})$ can used for dataflow analysis on those set of facts, as is done in the IFDS framework [17].

The *height* $\mathcal{H}$ of a weight domain is defined to be the length of the longest descending chain in the semiring (if it exists). In this paper, we assume the height to be finite for ease of discussing complexity results. (For cases when the height of a weight domain is unbounded, the value $\mathcal{H}$ in the complexity results can be interpreted as the length of the longest descending chain that occurs while solving a particular problem instance, which is always bounded.)

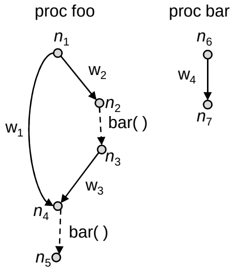
Definition 3. *A* **weighted pushdown system** *is a triple* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ *where* $\mathcal{P} = (P, \Gamma, \Delta)$ *is a pushdown system,* $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ *is a bounded idempotent semiring and* $f : \Delta \to D$ *is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

Let $\sigma \in \Delta^*$ be a sequence of rules. Using f, we can associate a value to σ, i.e., if $\sigma = [r_1, \ldots, r_k]$, then we define $v(\sigma) \overset{\text{def}}{=} f(r_1) \otimes \ldots \otimes f(r_k)$. Moreover, if for two configurations c and c' of $\mathcal{P}$, σ is a rule sequence that transforms c to c', we say $c \Rightarrow^\sigma c'$.

Definition 4. *Let* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ *be a WPDS, where* $\mathcal{P} = (P, \Gamma, \Delta)$*, and let* $S, T \subseteq P \times \Gamma^*$ *be regular sets of configurations. The* ***interprocedural meet-over-all-paths (IMOP)*** *value* $\mathit{IMOP}(S, T)$ *is defined as* $\bigoplus\{v(\sigma) \mid s \Rightarrow^\sigma t, s \in S, t \in T\}$*.*

The IMOP value describes the net transformation that occurs when going from one set of configurations to another. Sometimes we would write IMOP(s, t) for IMOP$(\{s\}, \{t\})$.

Fig. 1 shows how a program can be encoded using a WPDS. Each ICFG edge e is encoded as a PDS rule whose weight is the dataflow transformer for e. More details on encoding programs as WPDSs can be found in [19, 18].



(1) $\langle p, n_1 \rangle \hookrightarrow \langle p, n_2 \rangle$ w_2
(2) $\langle p, n_1 \rangle \hookrightarrow \langle p, n_4 \rangle$ w_1
(3) $\langle p, n_2 \rangle \hookrightarrow \langle p, n_6\ n_3 \rangle$ $\overline{1}$
(4) $\langle p, n_3 \rangle \hookrightarrow \langle p, n_4 \rangle$ w_3
(5) $\langle p, n_4 \rangle \hookrightarrow \langle p, n_6\ n_5 \rangle$ $\overline{1}$
(6) $\langle p, n_6 \rangle \hookrightarrow \langle p, n_7 \rangle$ w_4
(7) $\langle p, n_7 \rangle \hookrightarrow \langle p, \varepsilon \rangle$ $\overline{1}$

Fig. 1. A program with two procedures and its corresponding WPDS. Procedure calls are represented using dashed arrows.

Model Semantics In WPDSs, *program states* are represented using weighted configurations, which are configuration-weight pairs. The pair (c, w) describes the control state of the program as the PDS stack c, and the data state of the program using the weight w. A set of program states is represented by a function $\beta : P \times \Gamma^* \to D$, standing for the set $\{(c, \beta(c)) \mid c \in P \times \Gamma^*\}$. The set of all forward-reachable states starting from β is the set *poststar*$(\beta) = \{(c', \oplus_c\{\beta(c) \otimes \text{IMOP}(c, c')\}) \mid c' \in P \times \Gamma^*\}$. In this case, we say that configuration c' can be reached with weight *poststar*$(\beta)(c')$ (which is $\overline{0}$ if c' is not reachable). Backward-reachable states can be defined similarly.

For example, the initial state of the program in Fig. 1 is $(\langle p, n_1 \rangle, \overline{1})$. States reachable from this initial state include $(\langle p, n_6\ n_3 \rangle, w_2)$ and $(\langle p, n_6\ n_5 \rangle, w_1 \oplus (w_2 \otimes w_4 \otimes w_3))$.

3 Solving Multiple Intraprocedural Queries

Our interprocedural algorithm (§4) will need to solve multiple intraprocedural queries. Thus, we address the latter case first. A directed graph is a special case of a PDS (no call or return rules). When a weight domain is paired with a directed graph, we obtain a model for intraprocedural analysis. To simplify the discussion of intraprocedural algorithms, we specialize some definitions to weighted graphs.

Definition 5. *A* ***weighted graph*** *G is a tuple (V, E, λ), where (V, E) is a directed graph, and $\lambda : E \to D$ is a function that labels each edge with a weight.*

For vertices $v_1, v_2 \in V$, a path σ is defined as a sequence of edges that connect v_1 to v_2, in the standard way. In such a case, we say $v_1 \Rightarrow^{\sigma} v_2$. The weight of a path $\sigma = [e_1, e_2, \cdots, e_n]$, written as $\lambda(\sigma)$, is defined to be $\lambda(e_1) \otimes \lambda(e_2) \otimes \cdots \otimes \lambda(e_n)$. For sets of vertices $S, T \subseteq V$, the meet-over-all-paths (MOP) value is defined as the combine of weights of all paths that lead from a vertex in S to a vertex in T: $\text{MOP}(S, T) = \bigoplus\{\lambda(\sigma) \mid s \Rightarrow^{\sigma} t, s \in S, t \in T\}$. When $S = \{s\}$ and $T = \{t\}$ are singleton sets, we write $\text{MOP}[s, t]$ as a shorthand for $\text{MOP}(S, T)$.

For weighted graphs, *program states* are vertex-weight pairs (vertices replace PDS configurations). Computing reachable states reduces to solving the following query:

Definition 6. *Given a weighted graph G, and a set of vertices $S \subseteq V$ with a weight assignment $\mu : S \to D$, the* ***IntraQ-query*** *is to compute the weights* $\textsc{IntraQ}_G(S,\mu)(v)$ *for each $v \in V$, where* $\textsc{IntraQ}_G(S,\mu)(v) = \bigoplus_{s \in S}\{\mu(s) \otimes MOP[s,v]\}$.

$\textsc{IntraQ}_G(S,\mu)$ is the set of reachable states starting from $\{(s,\mu(s)) \mid s \in S\}$. We drop the subscript G in $\textsc{IntraQ}$ when it is obvious from the context.

When the graph G is fixed, we can preprocess it to quickly answer subsequent queries. We now present three different algorithms for solving this query, where each of them trades off preprocessing time against time required to solve a query.

Alg1: The first algorithm is the standard way of solving such queries using no preprocessing. It is a saturation algorithm: each vertex v has a weight $l(v)$. Initially, $l(s) = \mu(s)$ for $s \in S$, and $\overline{0}$ for other vertices. Next, the rules $l(v) := l(v) \oplus (l(u) \otimes \lambda(u,v))$ for each edge (u,v) are used to update the weights until a fixpoint is reached. Then $l(v)$ is the required value for $\textsc{IntraQ}(S,\mu)(v)$. This requires time $O_s(|E|\mathcal{H})$, where $\mathcal{H}$ is the height of the weight domain, and the notation $O_s(.)$ denotes the asymptotic number of semiring operations. (Because we consider weights as black boxes, the algorithms in this paper seek to minimize the number of semiring operations.)

The disadvantage of this method, which our other algorithms will address, is that it requires a fixpoint computation to be performed; this is reflected in the cost by the dependence on the height $\mathcal{H}$ of the weight domain, which can be large. For Boolean program verification, the height is exponential in the number of Boolean variables, and for affine-relation analysis, it is quadratic in the number of variables.

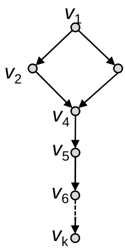


Fig. 2. A graph

Alg2: The second algorithm does the obvious preprocessing. It precomputes the values $\text{MOP}[v_1,v_2]$ for each $v_1, v_2 \in V$ by solving $\textsc{IntraQ}(\{v_1\},(v_1 \mapsto \overline{1}))$ for each v_1 using $\textsc{Alg}1$. Thus, preprocessing time is $O_s(|V||E|\mathcal{H})$. Once these MOP values are available, $\textsc{IntraQ}(S,\mu)(v)$ can be solved from its definition in time $O_s(|S|)$. Thus, $\textsc{IntraQ}(S,\mu)$ can be solved in time $O_s(|S||V|)$, which is independent of $\mathcal{H}$. This may seem like the most efficient approach, but we show next that one can do better.

Consider the graph in Fig. 2. Let w_{ij} be the weight on edge (v_i, v_j). Suppose that $S = \{v_2, v_3\}$, $\mu(v_2) = w_2$, and $\mu(v_3) = w_3$. Then $\textsc{Alg}2$ would require approximately $2|V|$ semiring operations because it considers v_2 and v_3 separately from each other. However, notice that vertex v_4 *dominates* all other vertices with respect to v_2 and v_3, i.e., any path in the graph starting at v_2 or v_3 must pass through v_4 before reaching vertices v_5 to v_k (and vertex v_1 is unreachable). Based on this observation, we can prove that $\textsc{IntraQ}(S, \mu)(v_i) = \textsc{IntraQ}(S, \mu)(v_4) \otimes \mathrm{MOP}[v_4, v_i]$ for $v_i \in \{v_5, \cdots, v_k\}$ Therefore, we only need to compute $\textsc{IntraQ}(S, \mu)(v_4)$ and other values can follow from this value using just one operation. This method would only require, approximately, $|V|$ number of operations.

This observation can be generalized to say that the weight on a vertex should be computed before the weights on vertices dominated by it are computed. This has been already captured nicely by Tarjan's algorithm [24] to solve path problems on graphs. However, it has only been used in the context of solving a single query, which we generalize to multiple queries. First, we summarize the essential details of Tarjan's algorithm.

Tarjan's Algorithm

Definition 7. *A* ***path expression*** *is a regular expression over the edges of a graph defined using the following grammar:* $r := \emptyset \mid \varepsilon \mid e \mid r_1.r_2 \mid r_1 \cup r_2 \mid r^*$ *where* e *is an edge in the graph. A path expression* r *is said to represent the set of paths in the language* $\mathcal{L}(r)$ *of* r *when interpreted as a regular expression. Furthermore, a path expression is said to be of type* (u, v) *if all paths in* $\mathcal{L}(r)$ *go from vertex* u *to vertex* v*.*

For example, for the graph in Fig. 2, the expression $((e_{12}.e_{24} \cup e_{13}.e_{34}).e_{45})$, where e_{ij} is the edge (v_i, v_j), denotes the set of all paths from v_1 to v_5, and is of type (v_1, v_5).

We extend the weight-evaluation function λ to path expressions as follows: $\lambda(\emptyset) = \overline{0}, \lambda(\varepsilon) = \overline{1}, \lambda(r_1.r_2) = \lambda(r_1) \otimes \lambda(r_2)$, $\lambda(r_1 \cup r_2) = \lambda(r_1) \oplus \lambda(r_2)$, and $\lambda(r^*) = \lambda(r)^*$. Here, we define the weight w^* as the infinite combine $\overline{1} \oplus w \oplus (w \otimes w) \oplus (w \otimes w \otimes w) \oplus ...$, which exists because of Defn. 2(item 4). One can show that $w^* = (\overline{1} \oplus w)^{\mathcal{H}}$, and calculate it using repeated squaring in time $O_s(\log \mathcal{H})$. Consequently, the following lemma holds. (We define $|r|$ to be the length of the expression.)

Lemma 1. *For a path expression* r *and the evaluation function* λ *defined as above,* $\lambda(r) = \bigoplus\{\lambda(\sigma) \mid \sigma \in \mathcal{L}(r)\}$*. Moreover, it can be calculated in time* $O_s(|r| \log \mathcal{H})$*.*

Tarjan's algorithm is based on computing path expressions to represent the set of paths between each pair of vertices. However, instead of computing a separate path expression for each pair of vertices, it computes a *path sequence*, which is a more concise way of representing all paths in a graph.

Definition 8. *A* ***path sequence*** *of a directed graph* $G = (V, E)$ *is a sequence* (r_1, u_1, v_1), $(r_2, u_2, v_2), \cdots (r_k, u_k, v_k)$*, where* $u_i, v_i \in V$*,* r_i *is a path expression of type* (u_i, v_i) *such that for any nonempty path* σ *in* G*, there is a sequence of indices* $1 \leq i_1 < i_2 < \cdots < i_l \leq k$ *and a partition of* σ *into nonempty paths* $\sigma = \sigma_1\sigma_2 \cdots \sigma_l$ *and* $\sigma_j \in \mathcal{L}(r_{i_j})$ *for all* $1 \leq j \leq l$*.*

Fig. 3(a) is an algorithm that uses a path sequence to create path expressions $r[s, v]$ that represent the set of all paths from s to v, for each $v \in V$ and a fixed $s \in V$ [24]. Using Lemma 1, we get $\mathrm{MOP}[s, v] = \lambda(r[s, v])$. Equivalently, the path expressions can be evaluated first and then put together to get the MOP weights, as shown in Fig. 3(b).

```
// initialize
for all $v \in V$ do
    $r[s,v] := \emptyset$
end for
$r[s,s] := \varepsilon$
// solve
for $i = 1$ to $k$ do
    $r[s,v_i] := r[s,v_i] \cup (r[s,u_i].r_i)$
end for
```

(a)

```
// initialize
for all $v \in V$ do
    $\mathrm{MOP}[s,v] := \overline{0}$
end for
$\mathrm{MOP}[s,s] := \overline{1}$
// solve
for $i = 1$ to $k$ do
    $\mathrm{MOP}[s,v_i] := \mathrm{MOP}[s,v_i] \oplus (\mathrm{MOP}[s,u_i] \otimes \lambda(r_i))$
end for
```

(b)

Fig. 3. Computing MOP values using the path sequence $\{(r_i, u_i, v_i)\}_{i=1}^{k}$.

Tarjan's algorithm computes a path sequence for a graph in time $O(|E| \log |V| + \delta)$, where δ is a term that denotes the *irreducibility* factor of the graph. For reducible graphs, $\delta = 0$ and, in general, δ is bounded by $|V|^3$. Because the graphs we work with come from CFGs of procedures, they are mostly reducible and the δ term can be ignored (which is confirmed by our experiments). The computed path expressions have a shared representation with a combined size bounded by the running time of the entire procedure. The length of the path sequence also has the same bound. Thus, evaluating all path expressions takes time $O_s((|E| \log |V| + \delta) \log \mathcal{H})$. After that, given a vertex s, solving for $\mathrm{MOP}[s, v]$ for all vertices v requires time $O_s(|E| \log |V| + \delta)$, which is almost linear in the size of the graph. We ignore the δ term in the rest of the paper.

Alg3: We now show how to use the path sequences computed by Tarjan's algorithm to solve multiple queries. Suppose that we wish to solve $\mathrm{I\small{NTRA}}\mathrm{Q}(S, \mu)$ on $G = (V, E, \lambda)$. Given S, μ and G, we construct a new graph $G' = (V', E', \lambda')$ by adding a new vertex to G: for some $v_0 \notin V$, $V' = V \cup \{v_0\}$, $E' = E \cup \{(v_0, s) \mid s \in S\}$, $\lambda' = \lambda \cup \{[(v_0, s) \mapsto \mu(s)] \mid s \in S\}$. Then, in this graph, $\mathrm{MOP}_{G'}[v_0, v] = \mathrm{I\small{NTRA}}\mathrm{Q}_G(S, \mu)(v)$. Thus, we need to compute MOP values on G'. This trick is similar to the standard one of reducing a multi-source reachability problem to a single-source reducibility problem. The following observation shows that a path sequence for G' can be computed from that of G:

Lemma 2. *If ps is a path sequence of G, then by concatenating the sequence $\{(v_0, s, (v_0, s)) \mid s \in S\}$ (with any arbitrary order chosen to enumerate vertices in S) in front of ps, one obtains a path sequence for G'.*

The preprocessing step of ALG3 computes a path sequence for G and evaluates the weight of each of its path expressions. Then to solve each query, ALG3 uses the path sequence for G', constructed using Lemma 2, as input to the algorithm in Fig. 3(b). This gives us the required weights $\mathrm{I\small{NTRA}}\mathrm{Q}_G(S, \mu)(v)$ as $\mathrm{MOP}_{G'}[v_0, v]$. ALG3 requires $O_s(|E| \log |V| \log \mathcal{H})$ preprocessing time and

$O_s(|S| + |E| \log |V|)$ time to solve each query. This is much better than Alg2 because for CFGs, $|E|$ is usually $O(|V|)$.

In essence, Alg3 works better than previous algorithms because it is able to concisely represent all values in $\mathrm{MOP}[\cdot, \cdot]$ and maintains enough of the graph structure in path sequences to take advantage of dominators. The above lemma ensures that path sequences can be reused for different queries. We used Alg3 in our experiments.

4 Solving Multiple Queries on WPDSs

In this section, we develop algorithms for the interprocedural case. For graphs, the number of vertices is finite, but for WPDSs, the number of configurations may be infinite (when the program is recursive), or very large (exponential in the number of procedures when not recursive). For this reason, sets of weighted configurations (or program states) are represented symbolically using weighted automata [19].

Definition 9. *A weighted automaton $\mathcal{A}$ is a finite-state automaton where each transition is additionally labeled with a weight. The weight of a path in the automaton is obtained by taking an extend of the weights on the transitions in the path in the backward direction. The automaton is said to accept a configuration $\langle p, u \rangle$ with weight w, denoted by $\mathcal{A}(\langle p, u \rangle)$, if w is the combine of weights of all accepting paths for u starting from state p in $\mathcal{A}$. The set of states of $\mathcal{A}$ is assumed to contain P, the set of PDS states.*

A weighted automaton $\mathcal{A}$ represents the set of states $\mathcal{R}(\mathcal{A}) = \{(c, \mathcal{A}(c)) \mid \mathcal{A} \text{ accepts } c, \mathcal{A}(c) \neq \overline{0}\}$. An important result is that for a weighted automaton $\mathcal{A}$, the set *poststar*$(\mathcal{R}(\mathcal{A}))$ (as defined in §2) can also be represented by a weighted automaton [19]. For brevity, we call such an automaton *poststar*$(\mathcal{A})$. Our goal is to preprocess a given WPDS so that *poststar*$(\mathcal{A})$ can be computed quickly for any given $\mathcal{A}$. We produce the same automaton for *poststar*$(\mathcal{A})$, as do previous algorithms [19, 10] (but faster, after the preprocessing stage).

An example is shown in Fig. 4(a). Note how the weight for a configuration $\langle p, n_7\ n_3 \rangle$ is represented in a compositional way in the automaton. Procedure `bar` is analyzed independently of its calling context, resulting in weight w_4 for transition (p, n_7, q). The weight w_2 at the call site n_3 to `bar` is captured on the transition (q, n_3, acc), resulting in a total weight of $w_4 \otimes w_2$ for $\langle p, n_7\ n_3 \rangle$. We will use the fact that procedures are analyzed independently of their calling context (also customary in most summary-based interprocedural analyses) in our favor. (As we shall see later, this implies that weights have to be propagated from a procedure to its callers, but not to procedures that it calls.)

Fix $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ to be a WPDS and $\mathcal{P} = (P, \Gamma, \Delta)$ to be its PDS. Fix $\mathcal{A}_{\mathrm{start}}$ to be the input query automaton, for which we want to compute *poststar*$(\mathcal{A}_{\mathrm{start}})$. To simplify the discussion, assume that $\mathcal{W}$ was created from a program as described in §2, and $P = \{p\}$ is a singleton set (our implementation handles any WPDS, however).

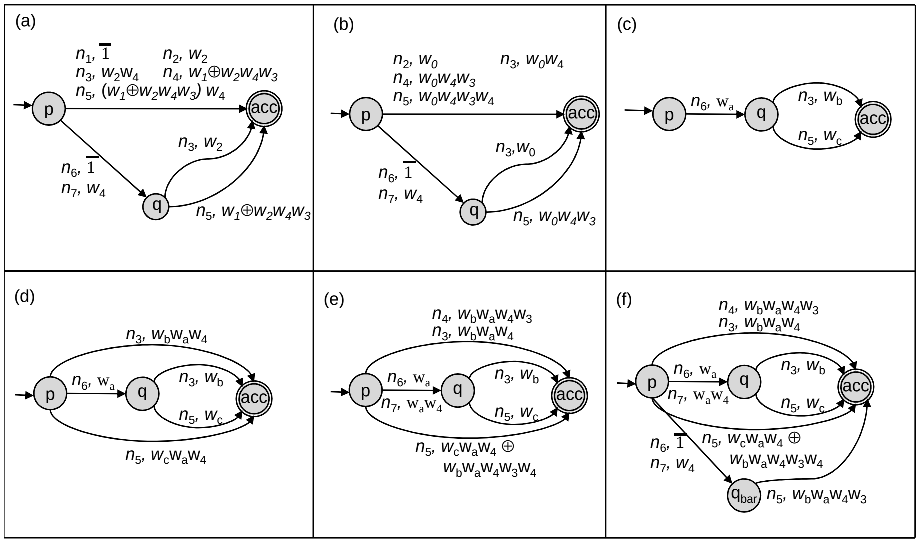


Fig. 4. Various automata related to the WPDS of Fig. 1. In all the weighted automata, juxtaposition of weights denotes their extend, *acc* is the accepting state, and parallel transitions have sometimes been collapsed into a single edge. Labels on transitions are (stack symbol, weight) pairs. (*a*) An automaton for *poststar*$(\{(\langle p, n_1\rangle, \overline{1})\})$. (*b*) An automaton for *poststar*$(\{(\langle p, n_2\rangle, w_0)\})$. (*c*) Automaton $\mathcal{A}_{\mathrm{start}}$. (*d*) Automaton $\mathcal{A}_{\mathrm{pop}}$ obtained after running the pop-phase on $\mathcal{A}_{\mathrm{start}}$. (*e*) Automaton $\mathcal{A}_{\mathrm{int}}$ created while running the growth phase on $\mathcal{A}_{\mathrm{pop}}$. (*f*) The final result of running the growth phase on $\mathcal{A}_{\mathrm{pop}}$.

Next, we give the preprocessing required by our algorithm; then an overview of how the algorithm would work on non-automata-based representations; then the algorithm for automata; followed by a discussion of the running-time complexity.

Preprocessing (*i*) First, we compute a summary for each procedure. (For a procedure starting at node e, it is defined as $\mathrm{IMOP}(\langle p, e\rangle, \langle p, \varepsilon\rangle)$). Using these summaries, we construct a weighted graph for each procedure from its CFG: the call edges (from call site to return site) are replaced with a summary of the called procedure. For $\gamma \in \Gamma$, let $\textsc{Pr}_\gamma$ be the procedure that contains γ, G_γ be the weighted graph for $\textsc{Pr}_\gamma$, e_γ its unique entry node, and x_γ its unique exit node. (Note that $\mathrm{MOP}_{G_\gamma}[e_\gamma, x_\gamma]$ also equals the summary for $\textsc{Pr}_\gamma$.) Next, for each weighted graph G of a procedure, we compute: (*ii*) its path sequence (preprocessing for $\textsc{Alg}3$) and (*iii*) values $\mathrm{MOP}_G[\gamma, x_\gamma]$ and $\mathrm{MOP}_G[e_\gamma, \gamma]$ for each node γ of the procedure.

The procedure summaries can be computed using standard algorithms, after which the path sequences can be constructed using Tarjan's algorithm. This would be an acceptable solution, but we can do better. We use our techniques from [10] to compute both of these at the same time. Briefly, the call-return edge in the CFG of a procedure is labeled with a variable whose value stands for the (as yet uncomputed) summary of the called procedure. Then the procedure summary is represented using a path expression (from entry node to return node) computed from its path sequence. This expression will have variables standing for summaries of called procedures. This gives rise to a set of equations whose solution solves for all summaries. In [10], we showed that this technique provides up to 5 times speedup over standard algorithms for computing procedure summaries, and we obtain a path sequence as a by-product. The path sequences can then be used to quickly compute the required MOP values for (*iii*) [24].

ICFG-version Before describing how our algorithm works with weighted automata, we give an informal description on how it would work with ICFGs (after preprocessing). Suppose that we are given a set R of node-weight pairs (starting states), where the nodes may be from multiple procedures, and we want to calculate the reachable set of node-weight pairs.

One challenge is to isolate the intraprocedural work. An $\textsc{IntraQ}$ query on a set $S = \{s_1, \cdots, s_k\}$ can also be solved by making a separate query for each s_i and taking a combine of the results, but this is far less efficient than making a single query on S. Thus, we want to minimize the number of $\textsc{IntraQ}$ queries made for each procedure. For example, for the program in Fig. 1, suppose $R = \{(n_6, w_a), (n_4, w_b)\}$. Then the pair (n_6, w_a) can produce the pairs $(n_3, w_a \otimes w_4)$ and $(n_5, w_a \otimes w_4)$ inside `foo`, when `bar` returns. We would then like to make just one $\textsc{IntraQ}$ query on `foo` with $S = \{n_3, n_4, n_5\}$ (and appropriate weights), instead of making a query with just n_4 first, and then realizing that the procedure has to be explored again from n_3 and n_5.

The algorithm proceeds in two phases. The first phase moves across procedure boundaries: if $(n, w) \in R$ then we propagate this weight to the callers of $\textsc{Pr}_n$. We add $(r, w \otimes \text{MOP}_{G_n}[n, x_n])$ to R for each return site r of calls to $\textsc{Pr}_n$ (if the pair (r, w') was already present in R, then change w' to $w' \oplus (w \otimes \text{MOP}_{G_n}[n, x_n])$). This continues until saturation. The use of (precomputed) $\text{MOP}[n, x_n]$ weights allow us to quickly jump from a procedure to its callers.

The second phase is intraprocedural. If $(n_1, w_1), \cdots, (n_k, w_k) \in R$ and the n_i are from the same procedure, run $\textsc{IntraQ}(\{n_1, \cdots, n_k\}, [n_i \mapsto w_i])$ to get weights on all other nodes in the procedure. This is repeated for all procedures. The resulting node-weight pairs represent all reachable states.[4]

The extension of these ideas to WPDSs have two complications: First, configurations add extra constraints on how weights get propagated to callers. For example, starting at configuration $\langle p, \gamma_1\gamma_2 \rangle$ constrains weight propagation to γ_2 when $\textsc{Pr}_{\gamma_1}$ returns (and not to its other return sites). Second, the number of

[4] An alternative to this algorithm would be to construct a single graph for the entire program, consisting of return edges and intraprocedural edges, then compute its path sequence and use that to process each ICFG query. The disadvantage with this method is that the graph need not be reducible, even if procedure CFGs were reducible, leading to a less efficient algorithm.

$$\begin{array}{llr}
\langle p, \gamma_1\ \gamma_2\ \gamma_3 \cdots \gamma_n \rangle & \Rightarrow^{\sigma_1} & \langle p, \gamma_2\ \gamma_3 \cdots \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
 & \Rightarrow^{\sigma_2} & \langle p, \gamma_3 \cdots \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
 & \Rightarrow^{*} & \cdots \\
 & \Rightarrow^{\sigma_k} & \langle p, \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
 & \Rightarrow^{\sigma_{k+1}} & \langle p, u_1\ u_2 \cdots u_j\ \gamma_{k+2} \cdots \gamma_n \rangle
\end{array}$$

Fig. 5. A path in the PDS's transition relation; $u_i \in \Gamma, j \geq 1, k \leq n, \sigma_h \in \Delta^*$.

configurations may be infinite, forcing us to use automata-based symbolic representations.

The above ICFG version only required at most one IntraQ query per procedure, which is ideal. The general version for WPDSs requires slightly more queries: at most $|Q|$ queries per procedure, where Q is the set of states of $\mathcal{A}_{\text{start}}$. In general, it requires m queries on a procedure if m is the number of states in $\mathcal{A}_{\text{start}}$ that have an incoming transition on a node of the procedure.

WPDS-version Consider a path $\sigma \in \Delta^*$ in the transition relation of a PDS that starts from a configuration $\langle p, \gamma_1\gamma_2 \cdots \gamma_n \rangle$. It can always be decomposed as $\sigma = \sigma_1\sigma_2 \cdots \sigma_k\sigma_{k+1}$ (see Fig. 5), such that $\langle p, \gamma_i \rangle \Rightarrow^{\sigma_i} \langle p, \varepsilon \rangle$ for $1 \leq i \leq k$ and $\langle p, \gamma_{k+1} \rangle \Rightarrow^{\sigma_{k+1}} \langle p, u_1u_2 \cdots u_j \rangle$ (or σ_{k+1} is empty when $k = n$). In other words, σ_i, $i \leq k$ is the rule sequence whose net effect is to pop off γ_i without looking at the stack below it, and σ_{k+1} is the rule sequence that does not look below γ_{k+1} but can replace it and add more symbols on top of the stack. We call the part where symbols are popped $(\sigma_1, \cdots, \sigma_k)$ the *pop phase* and the part where the stack grows (σ_{k+1}), the *growth phase.*

This property holds because PDS rules can only look at the top of the stack. For σ to touch γ_2, it must first pop off γ_1. When it does pop it off, this prefix would be σ_1 (and repeat inductively). If it does not pop off γ_1, then σ is already in the growth phase.

We compute *poststar*($\mathcal{A}_{\text{start}}$) by separating these two phases: We add transitions (with weights) to $\mathcal{A}_{\text{start}}$ to obtain $\mathcal{A}_{\text{pop}}$, which accepts all configurations reachable after the pop phase. Then we add transitions to $\mathcal{A}_{\text{pop}}$ to build $\mathcal{A}_{\text{final}}$, which accepts all configurations reachable after the growth phase. The former part is similar to the first phase of the ICFG-version of the algorithm, and the latter will correspond to the intraprocedural part, where each procedure is explored independently of others (similar to the second phase of the ICFG-version). A running example is shown in Fig. $4(c) - (f)$.

Terminology (i) A transition t with weight w is *added* to a weighted automaton $\mathcal{A}$ as follows: if t does not exist in $\mathcal{A}$, then insert it with weight w. If it exists in $\mathcal{A}$ with weight w', then change its weight to $w' \oplus w$. (ii) We say that $\mathcal{A}$ accepts a configuration c with weight *at least* w if $\mathcal{A}(c) \sqsubseteq w$ (Defn. 2, item 4). Note that all configurations are accepted with weight at least $\overline{0}$. (iii) The pop and growth phases are *saturation procedures.* They convert input $\mathcal{A}$ to output

$\mathcal{A}'$ by adding transitions to $\mathcal{A}$ until a fixpoint is reached; the fixpoint is the desired output $\mathcal{A}'$. Consequently, for all c, $\mathcal{A}'(c) \sqsubseteq \mathcal{A}(c)$, and thus for all c, $\mathcal{A}_{\text{final}}(c) \sqsubseteq \mathcal{A}_{\text{pop}}(c) \sqsubseteq \mathcal{A}_{\text{start}}(c)$.

Pop Phase Let w_γ be the weight with which γ can be popped, i.e., $w_\gamma = \text{IMOP}(\langle p, \gamma\rangle, \langle p, \varepsilon\rangle) = \text{MOP}_{G_\gamma}[\gamma, x_\gamma]$, which has been precomputed. We perform saturation on $\mathcal{A}_{\text{start}}$: if it accepts a configuration $\langle p, \gamma\ \gamma'\ u\rangle$, for any $u \in \Gamma^*$, with weight w, we make it accept $\langle p, \gamma'\ u\rangle$ with weight at least $w \otimes w_\gamma$, and repeat until a fixpoint is reached. This is done as follows: if (p, γ, q_1) and (q_1, γ', q_2) are transitions in the automaton with weight w_1 and w_2, respectively, then add the transition (p, γ', q_2) with weight $w_2 \otimes w_1 \otimes w_\gamma$ to the automaton. This process terminates because the number of new transitions added is bounded by $|T|$, where T is the set of transitions of $\mathcal{A}_{\text{start}}$. (This is because a transition (q_1, γ, q_2) in $\mathcal{A}_{\text{start}}$ can cause at most a single transition (p, γ, q_2) to be added to $\mathcal{A}_{\text{pop}}$.) Defn. 2 (item 4) ensures that weights on them can change at most $\mathcal{H}$ times. Moreover, the running time is bounded by $O_s(|T|\mathcal{H})$.

Intuitively, this works as follows: if there was a path in the automaton from q_2 to the final state on $u \in \Gamma^*$ with weight w, then the automaton can accept $\langle p, \gamma\ \gamma'\ u\rangle$ with weight $w \otimes w_2 \otimes w_1$. After the addition of the new transition, the automaton can accept $\langle p, \gamma'\ u\rangle$ with weight $w \otimes (w_2 \otimes w_1 \otimes w_\gamma)$. Fig. 4($d$) shows an example. For instance, $\mathcal{A}_{\text{start}}$ accepts $\langle p, n_6 n_3\rangle$ with weight $w_b \otimes w_a$, the weight with which n_6 can be popped is w_4, and $\mathcal{A}_{\text{pop}}$ accepts $\langle p, n_3\rangle$ with weight $w_b \otimes w_a \otimes w_4$.

Growth Phase For the growth phase, we need to consider all configurations reachable from the top symbols of currently accepted configurations, i.e., if $\langle p, \gamma\ u\rangle$, $u \in \Gamma^*$, is accepted by $\mathcal{A}_{\text{pop}}$ with weight w, and $\langle p, \gamma\rangle \Rightarrow^* \langle p, u'\rangle, u' \in \Gamma^+$ then $\langle p, u'\ u\rangle$ should be accepted by $\mathcal{A}_{\text{final}}$ with weight at least $w \otimes \text{IMOP}(\langle p, \gamma\rangle, \langle p, u'\rangle)$. This requires identification of the top symbols, but these are simply the ones on outgoing transitions from p in $\mathcal{A}_{\text{pop}}$ (nodes n_3, n_5 and n_6 in Fig. 4(d)).

Now we make use of the observation that called procedures are analyzed independently of their calling context, and reduce this phase to an intraprocedural one. For instance, see Fig. 4(b)—the weight w_0 need not be propagated to transitions involving nodes from `bar`.

The growth phase proceeds in two parts. The first part constructs automaton $\mathcal{A}_{\text{int}}$ such that if $\mathcal{A}_{\text{pop}}$ accepted configuration $\langle p, \gamma\ u\rangle$ with weight w and $\langle p, \gamma\rangle \Rightarrow^* \langle p, \gamma'\rangle$ then $\mathcal{A}_{\text{int}}$ accepts $\langle p, \gamma'\ u\rangle$ with weight at least $w \otimes \text{IMOP}(\langle p, \gamma\rangle, \langle p, \gamma'\rangle)$. This part requires running IntraQ queries. The second part constructs $\mathcal{A}_{\text{final}}$ from $\mathcal{A}_{\text{int}}$ by adding transitions with preprocessed weights.

For the first part, note that if $\langle p, \gamma\rangle \Rightarrow^* \langle p, \gamma'\rangle$, then γ' must be from the same procedure as γ (otherwise, the stack length would be different). Then $\text{IMOP}(\langle p, \gamma\rangle, \langle p, \gamma'\rangle) = \text{MOP}_{G_\gamma}[\gamma, \gamma']$. Hence, it suffices to do the following: if (p, γ, q) is a transition with weight w in $\mathcal{A}_{\text{pop}}$ then add transitions (p, γ', q) to it, for each γ' in the same procedure as γ, with weight $w \otimes \mathrm{MOP}_{G_\gamma}[\gamma, \gamma']$. This may add transitions with weight $\overline{0}$ if γ' is not reachable from γ, but such transitions can be removed without changing the meaning of a weighted automaton.

The above process can be optimized. Instead of looking at each transition in isolation, we handle them in bulk. For a state q of $\mathcal{A}_{\text{pop}}$, and a procedure $\textsc{Pr}$, let $S_q^{\textsc{Pr}}$ be the set of nodes s in $\textsc{Pr}$ such that (p, s, q) is a transition in $\mathcal{A}_{\text{pop}}$. Let $\mu_q^{\textsc{Pr}}$ be such that $\mu_q^{\textsc{Pr}}(s)$ is the weight on (p, s, q). Then add transition (p, s', q) with weight $\textsc{IntraQ}(S_q^{\textsc{Pr}}, \mu_q^{\textsc{Pr}})(s')$. It is easy to see that this imitates the above process, but is more efficient. This results in automaton $\mathcal{A}_{\text{int}}$. The running time is bounded by that required to answer $|Q||\text{Proc}|$ number of $\textsc{IntraQ}$ queries, where Q is the set of states of $\mathcal{A}_{\text{pop}}$ (same as those of $\mathcal{A}_{\text{start}}$), and $|\text{Proc}|$ is the number of procedures in the program.

An example is shown in Fig. 4(e): the algorithm invokes $\textsc{IntraQ}_{\texttt{bar}}(\{n_6\}, [n_6 \mapsto w_a])$ to add transitions between p and q. Next, it invokes $\textsc{IntraQ}_{\texttt{foo}}(\{n_3, n_5\}, [n_3 \mapsto w_b \otimes w_a \otimes w_4, n_5 \mapsto w_c \otimes w_a \otimes w_4])$ to add transitions between p and acc.

The second part of the growth phase adds transitions to accept configurations of called procedures. For each procedure $\textsc{Pr}$, add a new state $q_{\textsc{Pr}}$ to $\mathcal{A}_{\text{int}}$, and let $\textsc{Called}(\textsc{Pr})$ be *false* initially. Now repeat the following: if (p, γ, q) is a transition with weight w_1 and $\langle p, \gamma \rangle \hookrightarrow \langle p, c\ r \rangle$ is a WPDS rule with weight w_2, then (i) if $\textsc{Called}(\textsc{Pr}_c)$ is *false*, then set it to *true* and add transitions $(p, \gamma', q_{\textsc{Pr}_c})$ with weight $\mathrm{MOP}_{\textsc{Pr}_c}[c, \gamma']$, for each node γ' in $\textsc{Pr}_c$; (ii) add transition $(q_{\textsc{Pr}_c}, r, q)$ with weight $w_1 \otimes w_2$.

The intuition here is that with $\sigma = \langle p, \gamma \rangle \hookrightarrow \langle p, c\ r \rangle$, $\langle p, \gamma\ u \rangle \Rightarrow^{\sigma} \langle p, c\ r\ u \rangle$ for any $u \in \Gamma^*$. Addition of transitions $(p, c, q_{\textsc{Pr}_c})$ and $(q_{\textsc{Pr}_c}, r, q)$ ensures that the latter configuration is accepted (with appropriate weights). Next, c can reach node γ' in the same procedure with weight $\mathrm{MOP}_{\textsc{Pr}_c}[c, \gamma']$, for which the transitions $(p, \gamma', q_{\textsc{Pr}_c})$ are added. Note that the weight at the call site $(w_1 \otimes w_2)$ gets stored on the transition $(q_{\textsc{Pr}_c}, r, q)$. Thus, $\textsc{Pr}_c$ is analyzed independently of this weight and the weights on transitions $(p, \gamma', q_{\textsc{Pr}_c})$, for each γ' in $\textsc{Pr}_c$, are independent of the input query.

This process terminates because only a finite number of states are added. The trick of bounding the number of states is common in reachability algorithms for PDSs [19, 22]. The running time is bounded by $O_s(|Ret||Q|) + O(|\Gamma|)$, where Ret is the set of return sites in the program (only transitions added by (ii) require weight operations). This running time is subsumed by that of the first part. Fig. 4(f) shows an example.

Complexity First, we discuss the complexity of solving a query after preprocessing has been completed. Let Q be the set of states of $\mathcal{A}_{\text{start}}$, and T the set of its transitions. The pop phase has running time $O_s(|T|\mathcal{H})$. The growth phase, when using $\textsc{Alg3}$ for $\textsc{IntraQ}$ queries, has running time $O_s(|Q||Proc||E| \log |V|)$, where $|Proc|$ is the number of procedures in the program, and $|E|$ and $|V|$ are the average number of nodes per procedure. This gives a total worst-case running time of $O_s(|T|\mathcal{H} + |Q||Proc||E| \log |V|)$. The number of nodes per procedure usually remains constant even as program size increases. Treating $\log |V|$ as a constant, and writing $|E||\mathit{Proc}|$ as $|\Delta|$ (the number of WPDS rules), we get a total complexity of $O_s(|T|\mathcal{H} + |Q||\Delta|)$. This is asymptotically better than the complexity of previous algorithms [19, 10], which is $O_s(|T|\mathcal{H}+(|Q|+|\mathit{Proc}|)|\Delta|\mathcal{H})$ in each case. Note the reduced dependence on $\mathcal{H}$ for our algorithm (hence less fixpoint computation around loops and recursion).

If the initial set of configurations is finite (i.e., automaton $\mathcal{A}_{\text{start}}$ does not have any cycles), the running time of the pop phase can be bounded by $O_s(|T|)$, resulting in a total running time (after preprocessing) that is completely independent of the height of the weight domain (which is not true for any other WPDS reachability algorithm).

The complexity for preprocessing is dominated by the step that computes procedure summaries (and path sequences as a by-product). This just requires a single dataflow query, and its complexity is as mentioned for [10] above with $|T| = 1, |Q| = 2$. Using path sequences to compute the other preprocessing information is fairly quick.

For observed running times, the experiments reported in the next section show that the algorithms in this paper provide a speedup when running at least two queries.

5 Experiments

We refer to the implementation of the algorithm in this paper as SWPDS (Summary-WPDS). We compare against saturation-based [19] and optimized [10] approaches for solving WPDS queries, of which we pick the better running time and refer to it as OWPDS (Old-WPDS).

We carried out experiments on WPDSs obtained from three different applications. The first application is affine-relation analysis (ARA) of x86 programs [1]. A WPDS is produced from the x86 program using the weight domain for ARA described in [13]. The goal is to discover affine relationships (linear equalities) between machine registers.

The first experiment is to find out loop invariants (where loops are discovered by Bourdoncle's decomposition technique [3]). For outermost loops in a procedure, a loop summary is obtained as the weight IMOP($\langle p, \mathtt{head}\rangle, \langle p, \mathtt{head}\rangle$), where `head` is the head of the loop. This can be calculated by computing $\mathcal{A} = \mathit{poststar}(\{(\langle p, \mathtt{head}\rangle, \overline{1})\})$, and $\mathcal{A}(\langle p, \mathtt{head}\rangle)$. Loop invariants can be calculated easily from these summaries. These invariants give the conditions that hold at the head of the loop and are re-established after each iteration of the loop. We use common Windows executables, including code for the called libraries, and ran the experiments on a 3.2 GHz P4 processor with 3.3GB RAM running Windows XP.

A conventional way to solve these queries would be to compute the procedure summaries, plug them at the call-sites and then solve each loop as an intraprocedural problem. We call this technique OWPDS2. It uses ALG1 to solve each loop. Tab. 1 reports the following timings: the time taken to answer each query independently (OWPDS); the time taken by OWPDS2 (after procedure summaries have been computed); the preprocessing time for SWPDS (Setup); and the time taken to answer all queries using SWPDS, after preprocessing. We make two comparisons: (SWPDS+Setup) versus OWPDS, for which we are about 17 times faster (not shown in the table), and SWPDS versus OWPDS2, for which we are 1.5 times faster. We do not take the setup times into account in the second comparison because SWPDS preprocessing only computes procedure summaries (other preprocessing is unnecessary for this application).

Tab. 1 also shows a distribution of the obtained loop invariants. Loop invariants that indicate that a register remains unchanged after each loop iteration (even though it may get modified inside the loop) are reported separately from other kinds of invariants. The last eight columns show the number of loops that have a certain number of constant registers or affine invariants. For example, in `latex`, 53 loops do not have any constant registers, and 31 loops have 2 linearly independent invariants. These invariants can be beneficial to other analysis (e.g., they identify loop-induction variables).

				Time (s)					Constant Registers				Other Invariants			
Prog	Insts	Procs	Loops	OWPDS	OWPDS2	Setup	SWPDS	Speedup	0	1-3	4-6	7-8	0	1	2	≥ 3
latex	63711	609	280	168	5.7	28	4.3	**1.3**	53	44	152	31	124	125	31	0
attrib	103473	964	537	290	8.3	46	5.1	**1.7**	97	114	271	55	227	254	56	0
ftp	130352	1271	634	731	13.5	26	8.7	**1.6**	130	126	320	58	290	280	64	0
notepad	167430	1609	749	597	12.1	43	8.2	**1.5**	162	156	369	62	325	336	87	1
cmd	192579	1783	869	3415	24.1	64	18.0	**1.3**	256	156	391	66	431	355	80	3

Table 1. ARA experiments. The speedup is reported for SWPDS versus OWPDS2.

The second application is Boolean program verification using Moped [22]. Boolean programs are converted to WPDSs, and dataflow analysis is used for proving properties of the program. Boolean programs can also be represented using PDSs, but using WPDSs is advantageous because it allows symbolic manipulation of data using BDDs (which is essential for the scalability of Moped). In our experiments, the Boolean programs were obtained as a result of predicate abstraction. The following experiments were run on a 3GHz P4 processor with 2GB RAM running Linux, and the results are reported in Tab. 2.

We ran queries starting from the set of configurations ($n Ret^*$), where n is a program node and *Ret* is the set of return-site nodes. Such queries are useful for finding out the net effect in going from one program statement to another (for finding dependencies between the two). After the setup time, SWPDS was 4 times than OWPDS on forward reachability queries, and 7 times faster on backward reachability. This experiment also shows that two dataflow queries are enough to recover the SWPDS preprocessing time.

The third application (also based on Moped, running on the same Linux platform), considers context-bounded model checking (CBMC) [14], which aims to find all reachable states of a concurrent program under a bound on the number of context switches. Because we lack a front-end to abstract concurrent programs,

				Forward Reach.			Backward Reach.			CBMC		
Prog	Nodes	Procs	Setup	SWPDS	OWPDS	Speedup	SWPDS	OWPDS	Speedup	SWPDS	OWPDS	Speedup
bugs5	36971	291	11.9	4.2	18.4	**4.4**	1.4	8.4	**5.8**	98	183	**1.7**
unified-serial	38234	291	15.4	5.3	24.5	**4.7**	1.7	12.1	**7.1**	129	238	**1.6**
slam	7161	97	3.5	2.7	14.2	**5.3**	0.4	2.1	**6.0**	87	115	**1.3**
iscsiprt1	4803	82	0.6	0.27	0.84	**3.1**	0.06	0.36	**6.0**	6.3	12	**1.7**
ufloppy13	5679	64	1.5	0.6	2.0	**3.1**	0.07	0.7	**10.3**	12	20	**1.5**

Table 2. Experiments on Boolean programs: (*i*) Forward and backward reachability from the set of configurations $n\ Ret^*$. The node n was chosen randomly, and running times, reported in seconds, were averaged across 5 queries. The speedup is reported per query, ignoring the setup time. (*ii*) Simulated CBMC queries. The speedup reported takes the setup time into account.

we performed simulated experiments on sequential programs. We assume that the global variables of the program are shared with an environment that can randomly change their value, and the environment itself does not possess any local state. We ran one branch of CBMC along which control is transferred to the environment 5 times. Essentially, this requires the following: for random global states $g_1, \cdots, g_5$, if $\mathcal{A}_0$ describes the initial configuration of the (sequential) program, then compute $\mathcal{A}_1 = \mathit{poststar}(\mathcal{A}_0)$ and $\mathcal{A}_{i+1} = \mathit{poststar}(\textsc{Modify}(\mathcal{A}_i, g_i))$ for $i = 1$ to 5. Here, $\textsc{Modify}(\mathcal{A}, g)$ is an automaton that represents the same set of states as $\mathcal{A}$, but with the shared state changed to g. Because the result of running *poststar* is a bigger automaton than the original one, we report the total time taken to run all the queries. The average speedup was 1.6 times.

The varying amount of speedups in the different applications seem to be related to the size of S in $\textsc{IntraQ}(S, \mu)$ queries. For the ARA experiments, $|S|$ was always 1 (maximum speedup over OWPDS); for the second application, S was usually the set of return sites in a procedure; for CBMC, S consisted of all nodes in a procedure (least amount of speedup). Worst-case complexity does not predict this effect; this is an observation about measured behavior.

6 Related Work

The goal of incremental program analysis [5, 15, 11, 21, 6] is to reuse as much information as possible from previous fixpoint computations to calculate a new fixpoint when a small change is made to the program. Our work has aspects that resemble incremental computing in that we avoid recomputing the same information in response to changes in the query. However, we have a single preprocessing step to compute summaries and path sequences; this information is used by multiple dataflow queries, but there is no additional information tabulated during one query for use by a later query. This is because we do not look into the weights (and avoid caching computations over them), and base our optimizations only on the program control structure.

Another closely related category of work is that on demand-driven dataflow analysis [16, 20, 7]. There the focus is to do only as much work as is required to solve a query, and not redo it in a subsequent query. However, these techniques assume a particular form for the weights, and do look inside them (to work with *exploded CFGs*). We make fewer assumptions about the weights. These techniques would not be applicable to the weight domains we considered in our first application or be able to work with BDDs, as required by the other applications.

Technically, the most closely related piece of work is our previous work on speeding up a *single* dataflow query [10] called FWPDS. It used Tarjan's algorithm at the intraprocedural level to compute regular expressions for solving MOP values, and combined it with techniques like incremental computation of regular expressions to extend it for interprocedural analysis. In this paper, we use Tarjan's algorithm to compute path sequences. We combine it with a new WPDS reachability algorithm that shows how to summarize and reuse information at the interprocedural level. Moreover, FWPDS required the starting set of configurations to be in hand before it built the graphs on which it ran Tarjan's algorithm (these graphs need not be CFGs from the program). Thus, it would build different graphs for different queries, preventing it from sharing information between them. SWPDS outperforms FWPDS (included as OWPDS in §5).

References

1. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.
2. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.
3. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *FMPA*, 1993.
4. O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR*, 1992.
5. J. Cai and R. Paige. Program derivation by fixed point computation. *SCP*, 11(3), 1989.
6. C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *CAV*, 2005.
7. E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *POPL*, 1995.
8. S. Graham and M. Wegman. A fast and usually linear algorithm for global flow analysis. *J. ACM*, 23(1), 1976.
9. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.
10. A. Lal and T. Reps. Improving pushdown system model checking. In *CAV*, 2006.
11. Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *TOPLAS*, 20(3), 1998.
12. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.
13. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.
14. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.
15. G. Ramalingam and T. W. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.

16. T. Reps. Solving demand versions of interprocedural analysis problems. In *CC*, 1994.
17. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
18. T. Reps, A. Lal, and N. Kidd. Program analysis using weighted pushdown systems. In *FSTTCS*, 2007.
19. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SCP*, volume 58, 2005.
20. S. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2), 1996.
21. D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *ICLP*, 2003.
22. S. Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.
23. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1981.
24. R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.

A Optimizing Multiple Queries on ICFGs

This section shows how multiple queries on ICFGs can be sped up. The difference with WPDS-based queries is that these queries do not manipulate automata. Here, a program state is simply a program node paired with a weight, where the node may be from any procedure. Given a set of such states, a dataflow query on ICFG asks for the set of reachable states in the same form. Let R_{start} be the set of starting states. For each program node n, let $C_n \subseteq \mathit{Ret}^*$ be the set of valid return contexts for n. The set C_n is generated by the non-terminal X_{PR_n} in the grammar that has the following production for each a, b, and r such that procedure a calls procedure b with return site r:

$$X_b \rightarrow \varepsilon \mid rX_a$$

This grammar enforces the constraint that when a procedure returns, control goes to one of the return sites from where it was called. The set R_{start} represents the set of weighted configurations $W_{\text{start}} = \{(\langle p, n\ u\rangle, w) \mid (n, w) \in R_{\text{start}}, u \in C_n\}$. If $W_{\text{final}} = \mathit{poststar}(W_{\text{start}})$, then an ICFG query asks for the set R_{final} consisting of all pairs (n, w) where $w = \oplus_u \{w_u \mid (\langle p, n\ u\rangle, w_u) \in W_{\text{final}}\}$.

As is standard, a path in an ICFG that starts at the program entry is defined as one that follows the matched-parentheses grammar with unbalanced open parenthesis, as defined by non-terminal S_1 in Fig. 6. This enforces the property that a procedure return must go to the most recent call-site, and the path may make calls that do not have a corresponding return (for unfinished calls).

Paths that may start at an arbitrary program node have to follow a grammar whose language consists of all *substrings* of words from the language for S_1. This is defined by the non-terminal S_2 in Fig. 6. It is a language of matched parenthesis with unbalanced closed parenthesis on the left and unbalanced open parenthesis on the right.

If we add an edge, labeled with the terminal e_m, from a call-site directly to the matching return-site (representing a *summary* edge for the called procedure), the grammar for valid paths is changed to one that uses the non-terminal M' instead of M, and the starting non-terminal is S_2. This can be concisely written as $((M'\ ]_i)^*\ M'\ ([_i\ M')^*\ M')$. Note that the brackets [ ] are no longer matched in this grammar because such occurrences are replaced with summary edges.

We process all valid paths in the ICFG by first considering ones generated by $(M'\ ]_i)^*$. We call this phase the *return phase* because the only interprocedural edges considered are return edges labeled with $]_i$ (similar to the pop phase for PDSs). Next, we consider paths generated by $(M'\ ([_i\ M')^*\ M')$ and call this phase the *call phase* because the interprocedural edges considered are call edges (similar to the growth phase for PDSs).

$M \rightarrow \varepsilon \mid e$	$U_c \rightarrow \varepsilon$	$U_r \rightarrow \varepsilon$	$S_1 \rightarrow U_c$	$M' \rightarrow \varepsilon \mid e$
$\mid\ [_i\ M\ ]_i$	$\mid\ M\ U_c$	$\mid\ M\ U_r$	$S_2 \rightarrow U_r\ U_c$	$\mid\ e_m$
$\mid\ M\ M$	$\mid\ [_i\ U_c$	$\mid\ ]_i\ U_r$		$\mid\ M'\ M'$

Fig. 6. Grammars for interprocedurally valid paths in an ICFG. Edges in the ICFG are labeled as follows: all intraprocedural edges are labeled with the terminal e, and the matching procedure call and return edges for call site i as $[_i$ and $]_i$, respectively.

The preprocessing for the ICFG algorithm is the same as that for the WPDS version. The return phase has to consider paths $(M'\ ]_i)$ repeatedly. These paths connect a node n to one of the valid return sites for $\textsc{Pr}_n$. Hence, the return phase saturates the set R_{start} as follows: For every pair (n, w), add $(r, w \otimes \text{MOP}_{G_n}[n, x_n])$ to the set, for each return site r of calls to $\textsc{Pr}_n$ (if the pair (r, w') is already present, then change w' to $w' \oplus (w \otimes \text{MOP}_{G_n}[n, x_n])$). The running time of this phase is bounded by $O_s(|R_{\text{start}}| + (|\mathit{Ret}|\mathcal{H}))$. Call the resulting set of states R_{return}. Note that $|R_{\text{return}}| \leq |R_{\text{start}}| + |\mathit{Ret}|$.

The call phase proceeds as follows: First, it considers paths generated by M'. This requires intraprocedural propagation. If $(n_1, w_1), \cdots, (n_k, w_k) \in R_{\text{return}}$ and the n_i are from the same procedure, add $(n, \textsc{IntraQ}(\{n_1, \cdots, n_k\}, [n_i \mapsto w_i])(n))$ to the set, and repeat for all procedures. Call the resultant set R_{int}. This is similar to the construction of the automaton $\mathcal{A}_{\text{int}}$ in the WPDS version. It requires at most one IntraQ query per procedure. While the WPDS version did not have to do much work beyond this step, the ICFG version still requires a fixpoint operation. This is because weighted automata represent weights of configurations in a compositional way, and hence weights did not have to be propagated from callers to their callees. This is not the case with the ICFG version. For example, for the program in Fig. 1, if $R_{\text{start}} = \{(n_4, w_a)\}$ then we must have the pair $(n_7, w_a \otimes w_4)$ in the final set of reachable states. In the WPDS version, this could be done without requiring any weight operations because the result is represented using two transitions that together accept the configuration $\langle p, n_7\ n_5 \rangle$.

Next, we perform saturation on the set R_{int} to consider all paths generated by $([_i\ M')^*\ M'$, which is the same as $[_i\ (M'\ [_i)^*)\ M'$ because empty paths can be ignored. Any path generated by $(M'\ [_i)$ connects a node n to the entry node of one of the callees of $\textsc{Pr}_n$. The saturation is performed using the following two rules: (i) for a pair (n, w) such that n is a call site to a procedure starting at node e, add $(e, w \otimes w_c)$ to the set; (ii) for a pair (e, w), such that e is the entry node of a procedure, add $(n, w \otimes MOP_{G_e}[e, n])$ to the set for each n that is a call site. Because the number of call sites is bounded by $|Ret|$, this saturation requires time $O_s(|Ret|\mathcal{H})$. Call the resulting set of states R'_{int}.

Next, we need to propagate weights intraprocedurally again (for the last M'). If $(n_1, w_1), \cdots, (n_k, w_k) \in R'_{\text{int}}$ and the n_i are from the same procedure, add $(n, \textsc{IntraQ}(\{n_1, \cdots, n_k\}, [n_i \mapsto w_i])(n))$ to the set, and repeat for all procedures. This requires at most one $\textsc{IntraQ}$ query per procedure. The resulting set is R_{final}, as required.

The complexity of the entire algorithm described above is $O_s(|R_{\text{start}}| + (|Ret|\mathcal{H}) + |\mathit{\Delta}| \log |V|)$, where $|\mathit{\Delta}|$ is the number of edges in the ICFG and $|V|$ is the average number of nodes per procedure.

B Proofs

In this section, we give a proof of correctness of the algorithm in §4. We have to prove that $\mathcal{A}_{\text{final}}$ indeed represents the set of all forward reachable states from the ones represented by $\mathcal{A}_{\text{start}}$. The proof is most easily presented using the correctness of the forward-reachability algorithm from [19]. Hence, we first recall some details of that algorithm, which we call $\textsc{Alg4}$. We refer to the algorithm from §4 as $\textsc{Alg5}$.

B.1 Algorithm $\textsc{Alg4}$

This algorithm is also a saturation procedure. Let $\mathcal{A}$ be the weighted automaton representing starting set of states, and $\mathcal{A}_{post^*}$ the result of running this algorithm.

Let Q be the states of $\mathcal{A}$. Then $p \in Q$ (where p is the only PDS state, which is also the initial state of $\mathcal{A}$) and, without loss of generality, we restrict $\mathcal{A}$ to not have any transitions leading to state p. The automaton $\mathcal{A}_{post^*}$ has states $Q \cup Q_{\text{mid}}$, where Q_{mid} is defined to be the set of states containing a state q_e for each procedure entry node e

The saturation algorithm starts by adding transitions (p, e, q_e) to $\mathcal{A}$ with weight $\overline{1}$ for each procedure entry e. Next, the algorithm proceeds by repeatedly applying the rules shown in Fig. 7, in any order. When a fixpoint is reached, transitions on ε are removed. The resultant automaton $\mathcal{A}_{post^*}$ represents all forward reachable from $\mathcal{A}$.

1. For each pop rule $\langle p,\gamma\rangle \hookrightarrow \langle p,\varepsilon\rangle$ with weight w_1 and transition (p,γ,q) with weight w_2, add transition (p,ε,q) with weight $w_1 \otimes w_2$.
2. For each rule $\langle p,\gamma\rangle \hookrightarrow \langle p,\gamma'\rangle$ with weight w_1 and transition (p,γ,q) with weight w_2, add transition (p,γ',q) with weight $w_1 \otimes w_2$.
3. For each call rule $\langle p,\gamma\rangle \hookrightarrow \langle p,\gamma'\gamma''\rangle$ with weight w_1 and transition (p,γ,q) with weight w_2, add transition $(q_{\gamma'},\gamma'',q)$ with weight $w_1 \otimes w_2$.
4. For transitions (p,ε,q) with weight w_1 and (q,γ,q') with weight w_2, add the transition (p,γ,q') with weight $w_2 \otimes w_1$.

Fig. 7. Saturation rules for Alg4.

B.2 Correctness of Alg5

We subscript IMOP with the WPDS over which it is computed. Recall that $\mathcal{W}$ is the WPDS for which reachability has to be performed, and its stack alphabet is Γ.

Pop Phase Let $\mathcal{W}'$ be the WPDS with rules $\langle p,\gamma\rangle \hookrightarrow \langle p,\varepsilon\rangle$ with weight $w_\gamma = \mathrm{IMOP}_{\mathcal{W}}(\{\langle p,\gamma\rangle\},\{\langle p,\varepsilon\rangle\})$ for each $\gamma \in \Gamma$. The algorithm for pop-phase runs in the same fashion as running Alg4 on $\mathcal{A}_{\mathrm{start}}$ using $\mathcal{W}'$ (only the saturation rules 1 and 4 can fire from Fig. 7). Hence, $\mathcal{A}_{\mathrm{pop}}$ represents all forward reachable states from $\mathcal{A}_{\mathrm{start}}$ under $\mathcal{W}'$. This means: $\mathcal{A}_{\mathrm{pop}}(c) = \bigoplus_{c'}\{\mathcal{A}_{\mathrm{start}}(c') \otimes \mathrm{IMOP}_{\mathcal{W}'}(c',c)\}$.

The weight $\mathrm{IMOP}_{\mathcal{W}'}(c',c)$ is exactly the weight with which the pop phase on $\mathcal{W}$ can take c' to c. The above equality shows that for all configurations c', $\mathcal{A}_{\mathrm{pop}}$ accepts configuration c with weight at least $\mathcal{A}_{\mathrm{start}}(c') \otimes \mathrm{IMOP}_{\mathcal{W}'}(c',c)$.

Growth Phase Let $\mathcal{W}''$ be the WPDS with the same rules as $\mathcal{W}$ but without the pop rules of $\mathcal{W}$. For each call rule $\langle p,\gamma\rangle \hookrightarrow \langle p,\gamma'\ \gamma''\rangle$ with weight w in $\mathcal{W}$, add the rule $\langle p,\gamma\rangle \hookrightarrow \langle p,\gamma''\rangle$ with weight $w \otimes \mathrm{IMOP}_{\mathcal{W}}(\{\langle p,\gamma'\rangle\},\{\langle p,\varepsilon\rangle\})$. Hence, $\mathcal{W}''$ is the WPDS with return rules removed and procedure calls replaced with summary edges (but the call rules remain in $\mathcal{W}''$).

It is easy to see that the WPDS $\mathcal{W}''$ captures the growth phase of any path in $\mathcal{W}$: the value $\mathrm{IMOP}_{\mathcal{W}}(\langle p,\gamma\rangle,\langle p,u\rangle)$, when $u \in \Gamma^+$ equals the value $\mathrm{IMOP}_{\mathcal{W}''}(\langle p,\gamma\rangle,\langle p,u\rangle)$. This holds because $\mathcal{W}''$ can short-circuit a procedure call and its subsequent return using the summary rule. Conversely, because $\mathcal{W}''$ is not allowed to pop, it cannot look below the top-most element of the stack and must always grow the stack.

Now we show that running the growth phase of Alg5 on $\mathcal{A}_{\mathrm{pop}}$ using $\mathcal{W}$ results in the same automaton as the one obtained by running Alg4 on $\mathcal{A}_{\mathrm{pop}}$ using $\mathcal{W}''$ (provided the check for Called(Pr) being *false* in Alg5 is removed—this is just a small optimization irrelevant for the correctness of the algorithm).

Consider the execution of Alg4. Let Q_{mid} be the new states added by Alg4. Because we have the freedom to apply saturation rules of Alg4 in any order, choose the order in which saturation rule 2 from Fig. 7 is used repeatedly until saturation, and then saturation rule 3 is used until saturation, and the process is repeated. Note that $\mathcal{W}''$ has a non-call rule for each edge in the weighted graph of each procedure.

First, the repeated application of saturation rule 2 on outgoing transitions from p that go to a state not in Q_{mid} mimics the construction of $\mathcal{A}_{\text{int}}$ by Alg5. When a fixpoint is reached, Alg4 would have changed $\mathcal{A}_{\text{pop}}$ to $\mathcal{A}_{\text{int}}$.

Second, the repeated application of saturation rule 2 on outgoing transitions from p that go to a state in Q_{mid} until a fixpoint mimics the addition of transitions (p, γ, q_e) with weight $\text{MOP}_{G_e}[e, \gamma]$ in Alg5.

These are all the transitions that saturation rule 2 can affect. Next, saturation rule 3 is used on call rules. This mimics the second phase of the growth phase of Alg5, when transitions leading out from Q_{mid} are added. Because rule 3 does not affect any transition used by rule 2, the saturation procedure of Alg4 stops here. Hence Alg4 does not do anything more than Alg5. This proves that the following holds:

$$\mathcal{A}_{\text{final}}(c) = \bigoplus_{c'} \{\mathcal{A}_{\text{pop}}(c') \otimes \text{IMOP}_{\mathcal{W}''}(c', c)\}$$

The above results can be combined to obtain the following:

$$\mathcal{A}_{\text{final}}(c) = \bigoplus_{c', c''} \{\mathcal{A}_{\text{start}}(c'') \otimes \text{IMOP}_{\mathcal{W}'}(c'', c') \otimes \text{IMOP}_{\mathcal{W}''}(c', c)\}$$

The fact that every WPDS path can be decomposed into a pop phase followed by a growth phase can be stated formally as: $\text{IMOP}_{\mathcal{W}}(c'', c) = \oplus_{c'} \{\text{IMOP}_{\mathcal{W}'}(c'', c') \otimes \text{IMOP}_{\mathcal{W}''}(c', c)\}$. Hence, we get:

$$\mathcal{A}_{\text{final}}(c) = \bigoplus_{c''} \{\mathcal{A}_{\text{start}}(c'') \otimes \text{IMOP}_{\mathcal{W}}(c'', c)\}$$

This proves that $\mathcal{A}_{\text{final}}$ indeed represents the set of forward reachable states from $\mathcal{A}_{\text{start}}$.

C Optimizing Multiple Backward Dataflow Queries

In this section, we give the algorithm for optimizing multiple backward dataflow queries on WPDSs. Given a set β of states (weighted configurations), the set of backward reachable states is defined as the set $\textit{prestar}(\beta) = \{(c', \oplus_c \{\text{IMOP}(c', c) \otimes \beta(c)\}) \mid c' \in P \times \Gamma^*\}$.

The standard algorithm for solving backward reachability can be found in [19]. The algorithm operates on weighed automata that are slighty different from those defined in Defn. 9. The difference is that the weight of an accepting path in the automaton is computed by taking the extend of the weights associated with the transitions in the *forwards* direction.

Let BIntraQ(S, μ) be the backward version of IntraQ(S, μ). It can be solved efficiently by computing a path sequence of the graph and then using an algorithm similar to the one shown in Fig. 3(b) to compute BIntraQ$(S, \mu)(v) = \oplus_{s \in S} \text{MOP}[v, s] \otimes \mu(s)$ for each vertex v in the graph.

The preprocessing phase remains the same as described in §4. A query is solved by first going backwards through the growth phase, and then backwards through the pop phase. Let $\mathcal{A}_0$ be the starting set of states (the query).

Let $w_\gamma = \text{MOP}_{G_\gamma}[\gamma, x_\gamma]$ be the pop-weight for γ. Let $\mathcal{W}'$ be the WPDS consisting of the rules $\langle p, e_\gamma \rangle \hookrightarrow \langle p, \gamma \rangle$ with weight $\text{MOP}_{G_\gamma}[e_\gamma, \gamma]$ for each $\gamma \in \mathit{\Gamma}$. Moreover, for each call rule $\langle p, \gamma \rangle \hookrightarrow \langle p, e\ \gamma' \rangle$ with weight w in $\mathcal{W}$, add the rule $\langle p, \gamma \rangle \hookrightarrow \langle p, \gamma' \rangle$ with weight $w \otimes w_e$.

The algorithm proceeds as follows: First, perform backward reachability on $\mathcal{A}_0$ using the WPDS $\mathcal{W}'$ and the standard algorithm from [19]. Let the result be $\mathcal{A}_1$. This part corresponds to going backward across procedure calls (from callee to caller). Second, perform BIntraQ queries. For each state q in $\mathcal{A}_1$ and procedure Pr, suppose that $\{(p, \gamma_i, q) | 1 \leq i \leq n, \gamma_i \in \text{Pr}\}$ is the set of all the transitions in $\mathcal{A}_1$ with nodes from Pr. Let $S = \{\gamma_1, \cdots, \gamma_n\}$ and $\mu(\gamma_i)$ be the weight on (p, γ_i, q). Solve $\text{BIntraQ}(S, \mu)$ and add transition (p, γ, q) with weight $\text{BIntraQ}(S, \mu)(\gamma)$ to the automaton. This completes the growth phase. Next, add transition (p, γ, p) with weight w_γ, for each $\gamma \in \mathit{\Gamma}^*$, to complete the pop phase. The resultant automaton represents $\mathit{prestar}(\mathcal{A}_0)$.