Improving Pushdown System Model Checking*

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

[1] University of Wisconsin.
[2] GrammaTech, Inc.
{akash, reps}@cs.wisc.edu

Abstract. In this paper, we reduce pushdown system (PDS) model checking to a graph-theoretic problem, and apply a fast graph algorithm to improve the running time for model checking. Several other PDS questions and techniques can be carried out in the new setting, including witness tracing and incremental analysis, each of which benefits from the fast graph-based algorithm.

1 Introduction

Pushdown systems (PDSs) have served as an important formalism for program analysis and verification because of their ability to concisely capture interprocedural control flow in a program. Various tools [6, 18, 12, 10, 4] use pushdown systems as an abstract model of a program and use reachability analysis on these models to verify program properties. Using PDSs provides an infinite-state abstraction for the control state of the program. Some of these tools [6, 18, 4], however, can only verify properties that have a finite-state data abstraction. Other tools [10, 12] are based on the more generalized setting of weighted pushdown systems (WPDSs) [16] and are capable of verifying infinite-state data abstractions as well.

At the heart of all these tools is a PDS reachability-analysis algorithm that uses a chaotic-iteration strategy to explore all reachable states [2, 7, 17]. Even though there has been work to address the worst-case running time of this algorithm [5], to our knowledge, no one has addressed the issue of giving direction to the chaotic-iteration scheme to improve the running time of the algorithm in practice. In this paper, we try to improve the worst-case running time, as well as the running-time observed in practice. To provide a common setting to discuss most PDS model checkers, we use WPDSs to describe our improvements to PDS reachability.

An interprocedural control flow graph (ICFG) is a set of graphs, one per procedure, connected via special call and return edges [14]. A WPDS with a given initial query can also be decomposed into a set of graphs whose structure is similar. (When the underlying PDS is obtained by the standard encoding of an ICFG as a PDS for use in program analysis, these decompositions coincide.) Next, we use a fast graph algorithm, namely the Tarjan path-expression algorithm [19] to represent each graph as a regular expression. WPDS reachability can then be reduced to solving a set of regular equations. When the underlying PDS is obtained from a structured (reducible) control flow graph, the regular expressions can be found and solved very efficiently. Even when the control flow is not structured, the regular expressions provide a fast iteration strategy that improves over the standard chaotic-iteration strategy.

Our work is inspired by previous work on dataflow analysis of single-procedure programs [20]. There it was shown that a certain class of dataflow analysis problems

* Supported by ONR (N00014-01-1-{0708,0796}) and NSF (CCR-9986308 and CCF-0524051).

can take advantage of the fact that a (single-procedure) CFG can be represented using a regular expression. We generalize this observation to multiple-procedure programs, as well as to WPDSs. The contributions of this paper can be summarized as follows:

- We present a new reachability algorithm for WPDSs that improves on previously known algorithms for PDS reachability. The algorithm is asymptotically faster when the PDS is *regular* (decomposes into a single graph), and offers substantial improvement in the general case as well.
- The algorithm is completely demand-driven, and computes only that information needed for answering a particular user query. It has an implicit slicing stage where it disregards parts of the program not needed for answering the user query.
- We show that several other PDS analysis questions and techniques, including witness tracing and incremental analysis, carry over to the new approach.

The rest of the paper is organized as follows: §2 provides background on PDSs and WPDSs. §3 presents the previously known algorithm and our new algorithm for solving reachability queries on WPDSs. In §4, we describe algorithms for witness tracing and incremental analysis. §5 presents experimental results. §6 describes related work.

2 PDS Model Checking

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* $\langle p, \gamma u' \rangle \Rightarrow \langle p', uu' \rangle$ *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. The standard approach for modeling program control flow is as follows: Let $(\mathcal{N}, \mathcal{E})$ be an ICFG where each *call* node is split into two nodes: one has an interprocedural edge going to the entry node of the procedure being called; the second has an incoming edge from the exit node of the procedure. $\mathcal{N}$ is the set of nodes in this graph and $\mathcal{E}$ is the set of control-flow edges. Fig. 1(a) shows an example of an ICFG, Fig. 1(b) shows the pushdown system that models it. The PDS has a single state p, one stack symbol for each node in $\mathcal{N}$, and one rule for each edge in $\mathcal{E}$. We use rules with one stack symbol on the right-hand side to model intraprocedural edges, rules with two stack symbols on the right-hand side (*push* rules) for *call* edges, and rules with no stack symbols on the right-hand side (*pop* rules) for *return* edges. It is easy to see that a valid path in the program corresponds to a path in the pushdown system's transition system, and vice versa. Thus, PDSs can encode ordinary control flow graphs, but they also provide a convenient mechanism for modeling certain kinds of non-local control flow, such as setjmp/longjmp in C. At a setjmp, we push a special symbol on the stack, and at a longjmp with the same environment variable (identified using some preprocessing) we pop the stack until that symbol is reached. The longjmp value can be passed using the state of the PDS.

Because the number of configurations of a pushdown system is unbounded, it is useful to use finite automata to describe certain infinite sets of configurations.

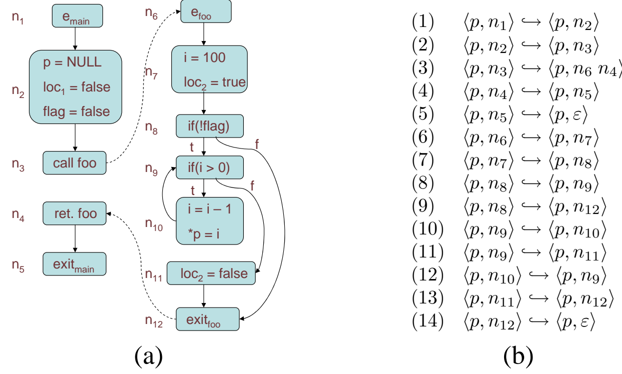

Fig. 1. (a) An ICFG. The e and *exit* nodes represent entry and exit points of procedures, respectively. `flag` is a global variable, $\texttt{loc}_1$ and $\texttt{loc}_2$ are local variables of `main` and `foo`, respectively. Dashed edges represent interprocedural control flow. (b) A pushdown system that models the control flow of the graph shown in (a).

Definition 2. *If $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system, then a* $\mathcal{P}$**-automaton** *is a finite automaton $(Q, \Gamma, \rightarrow, P, F)$ where $Q \supseteq P$ is a finite set of states, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the transition relation, P is the set of initial states, and F is the set of final states of the automaton. We say that a configuration $\langle p, u\rangle$ is accepted by a $\mathcal{P}$-automaton if the automaton can accept u when it is started in the state p (written as $p \xrightarrow{u}{}^{*} q$, where $q \in F$). A set of configurations is called* **regular** *if some $\mathcal{P}$-automaton accepts it.*

A weighted pushdown system is obtained by supplementing a pushdown system with a weight domain that is a bounded idempotent semiring [16, 3]. Such semirings are powerful enough to encode finite-state data abstractions such as the one required for Boolean program verification, as well as infinite-state data abstractions, such as copy-constant propagation and affine-relation analysis [12].

Definition 3. *A* **bounded idempotent semiring** *is a quintuple $(D, \oplus, \otimes, \overline{0}, \overline{1})$, where D is a set whose elements are called* **weights**, *$\overline{0}$ and $\overline{1}$ are elements of D, and $\oplus$ (the combine operation) and $\otimes$ (the extend operation) 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.*

Definition 4. *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 $f : \Delta \rightarrow D$ 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 $v(\sigma) \stackrel{\text{def}}{=} f(r_1) \otimes \ldots \otimes f(r_k)$. Moreover, for any two

configurations c and c' of $\mathcal{P}$, we use $path(c, c')$ to denote the set of all rule sequences $[r_1, \ldots, r_k]$ that transform c into c'. Reachability problems on pushdown systems are generalized to weighted pushdown systems as follows.

Definition 5. *Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$, and let $C \subseteq P \times \Gamma^*$ be a regular set of configurations. The* **generalized pushdown predecessor** (GPP) **problem** *is to find for each $c \in P \times \Gamma^*$:*

$$\delta(c) \stackrel{\text{def}}{=} \bigoplus \{ v(\sigma) \mid \sigma \in path(c, c'), c' \in C \}$$

The **generalized pushdown successor** (GPS) **problem** *is to find for each $c \in P \times \Gamma^*$:*

$$\delta(c) \stackrel{\text{def}}{=} \bigoplus \{ v(\sigma) \mid \sigma \in path(c', c), c' \in C \}$$

To illustrate the above definitions, let us encode Boolean programs as a WPDS. Consider the program shown in Fig. 1. It has one global variable `flag`. We ignore local variables for now, and details regarding their treatment can be found in [11]. Let G be the set of all valuations of global variables. In our case, $G = \{0, 1\}$ because we only have one global variable. Each ICFG edge can be associated with a transformer, which is a binary relation on G, and describes the effect of executing that edge on the global variables, e.g., the edge (n_2, n_3) will be associated with the relation $\{(0, 0), (1, 0)\}$ because `flag` is set to 0 (or `false`). Therefore, we use the weight domain $(2^{G \times G}, \cup, \circ, \emptyset, id)$, and for a PDS rule, we associate it with the transformer of the corresponding ICFG edge. Assertion checking in the program can be performed by seeing if a configuration c (or a set of configurations) can be reached with non-zero weight, i.e, $\delta(c) \neq \overline{0}$.

Boolean programs can also be encoded using PDSs by using the states of the PDS to encode valuations of global variables. However, WPDSs provide a more efficient representation of Boolean programs because the weights can symbolically encode transformers, for example, by using BDDs [17]. Moreover, WPDSs are strictly more powerful than PDSs because they can be used with *infinite-width* abstract domains to perform copy-constant propagation and affine relation analysis [12]. More details on the uses of PDSs for model checking, and their encoding as WPDSs can be found in [11].

3 Solving Reachability Problems

In this section, we review the existing algorithm for solving generalized reachability problems on WPDSs [16], which is based on chaotic iteration, and present our new algorithm, which uses Tarjan's path-expression algorithm [19]. We limit our discussion to GPP; GPS is similar but slightly more complicated.

3.1 Solving GPP using Chaotic Iteration

Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a WPDS where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system and $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is the weight domain. Let C be a regular set of configurations that is recognized by $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$. GPP is solved by saturating this automaton with new weighted transitions (each transition t has a weight label $l(t)$), to create automaton $\mathcal{A}_{pre^*}$, such that $\delta(c)$ can be read-off efficiently from $\mathcal{A}_{pre^*}$: $\delta(\langle p, u \rangle)$ is the combine of weights of all accepting paths for u starting from p, where the weight of a path is the extend of the weight-labels of the transitions in the path in order. We present the algorithm for building $\mathcal{A}_{pre^*}$ based on its abstract grammar problem.

Definition 6. *[16] Let $(S, \sqcap)$ be a meet semilattice. An* **abstract grammar** *over $(S, \sqcap)$ is a collection of context-free grammar productions, where each production θ has the*

form $X_0 \rightarrow g_\theta(X_1, \ldots, X_k)$. *Parentheses, commas, and* g_θ *(where* θ *is a production) are terminal symbols. Every production* θ *is associated with a function* $g_\theta \colon S^k \rightarrow S$. *Thus, every string* α *of terminal symbols derived in this grammar denotes a composition of functions, and corresponds to a unique value in* S*, which we call* $val_G(\alpha)$. *Let* $L_G(X)$ *denote the strings of terminals derivable from a nonterminal* X. *The* **abstract grammar problem** *is to compute, for each nonterminal* X*, the value* $\text{MOD}_G(X) = \bigsqcap_{\alpha \in L_G(X)} val_G(\alpha)$. *This value is called the* **meet-over-all-derivations** *value for* X.

We define abstract grammars over the meet semilattice $(D, \oplus)$, where D is the set of weights as given above. An example is shown in Fig. 2. The non-terminal t_3 can derive the string $\alpha = g_4(g_3(g_1))$ and $val(\alpha) = w_4 \otimes w_3 \otimes w_1$.

(1) $t_1 \rightarrow g_1(\epsilon)$	$g_1 = w_1$	(3) $t_2 \rightarrow g_3(t_1)$	$g_3 = \lambda x.w_3 \otimes x$
(2) $t_1 \rightarrow g_2(t_2)$	$g_2 = \lambda x.w_2 \otimes x$	(4) $t_3 \rightarrow g_4(t_2)$	$g_4 = \lambda x.w_4 \otimes x$

Fig. 2. A simple abstract grammar with four productions.

Production	for each
(1) $PopSeq_{(q,\gamma,q')} \rightarrow g_1(\epsilon)$ $g_1 = \overline{1}$	$(q, \gamma, q') \in \rightarrow_0$
(2) $PopSeq_{(p,\gamma,p')} \rightarrow g_2(\epsilon)$ $g_2 = f(r)$	$r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$
(3) $PopSeq_{(p,\gamma,q)} \rightarrow g_3(PopSeq_{(p',\gamma',q)})$ $g_3 = \lambda x.f(r) \otimes x$	$r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta, q \in Q$
(4) $PopSeq_{(p,\gamma,q)} \rightarrow g_4(PopSeq_{(p',\gamma',q')}, PopSeq_{(q',\gamma'',q)})$ $g_4 = \lambda x.\lambda y.f(r) \otimes x \otimes y$	$r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta, q, q' \in Q$

Fig. 3. An abstract grammar problem for solving GPP.

The abstract grammar for solving GPP is shown in Fig. 3. The grammar has one non-terminal $PopSeq_t$ for each possible transition $t \in Q \times \Gamma \times Q$ of $\mathcal{A}_{pre^*}$. The productions describe how the weights on those transitions are computed. Let $l(t)$ be the weight label on transition t. Then we want $l(t) = \text{MOD}(PopSeq_t)$. The meet-over-all-derivation value is obtained as follows [16]: Initialize $l(t) = \overline{0}$ for all transitions t. If $PopSeq_t \rightarrow g(PopSeq_{t_1}, PopSeq_{t_2})$ is a production of the grammar (with possibly fewer non-terminals on the right-hand side) then update the weight label on t to $l(t) \oplus g(l(t_1), l(t_2))$. The existing algorithm for solving GPP is a worklist-based algorithm that uses chaotic iteration to choose (i) a transition in the worklist and (ii) all productions that have this transition on the right side, and updates the weight on the transitions on the left-hand side of the productions as described earlier. If the weight on a transition changes then it is added to the worklist. Defn. 3(4) guarantees convergence.

Such a chaotic-iteration scheme is not very efficient. Consider the abstract grammar in Fig. 2. The most efficient way of saturating weights on transitions would be to start with the first production and then keep alternating between the next two productions until $l(t_1)$ and $l(t_2)$ converge before choosing the last production. Any other strategy would have to choose the last production multiple times. Thus, it is important to identify such "loops" between transitions and to stay within a loop before exiting it.

3.2 Solving GPP using Path Expressions

To find a better iteration scheme for GPP, we convert GPP into a hypergraph problem.

Definition 7. *A (directed)* ***hypergraph*** *is a generalization of a directed graph in which generalized edges, called* ***hyperedges****, can have multiple sources, i.e., the source of an edge is an ordered set of vertices. A* **transition dependence graph (TDG)** *for a grammar G is a hypergraph whose vertices are the non-terminals of G. There is a hyperedge from $(t_1, \cdots, t_n)$ to t if G has a production with t on the left-hand side and $t_1 \cdots t_n$ are the non-terminals that appear (in order) on the right-hand side.*

If we construct the TDG of the grammar shown in Fig. 3 when the underlying PDS is obtained from an ICFG, and the initial set of configurations is $\{\langle p, \varepsilon \rangle \mid p \in P\}$ (or $\rightarrow_0 = \emptyset$), then the TDG is identical to the ICFG (with edges reversed). Fig. 4 shows an example. This can be observed from the fact that except for the PDS states in Fig. 3, the transition dependences are almost identical to the dependences encoded in the pushdown rules, which in turn come from ICFG edges; e.g., the ICFG edge (n_1, n_2) corresponds to the transition dependence $((t_2), t_1)$ in Fig. 4, and the call-return pair (n_3, n_6) and (n_{12}, n_4) in the ICFG corresponds to the hyperedge $((t_4, t_6), t_3)$.

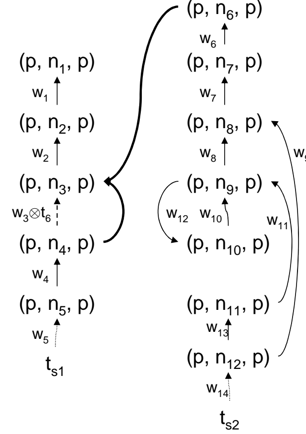

Fig. 4. TDG for the PDS shown in Fig. 1. A WPDS is obtained from the PDS by supplementing rule number i with weight w_i. Let t_j stand for the node (p, n_j, p). The thick bold arrows form a hyperedge. Nodes t_{s1} and t_{s2} are source nodes, and the dashed arrow is a summary edge. These, along with edge labels, are explained later in §3.2.

For such pushdown systems, constructing TDGs might seem unnecessary but it allows us to choose an initial set of configurations, which defines a region of interest in the program. Moreover, PDSs can encode much stronger properties than an ICFG, such as setjmp/longjmp in C programs. However, it is still convenient to think of a TDG as an ICFG. In the rest of this paper, we illustrate the issues using the TDG of the grammar in Fig. 3. We reduce the meet-over-all-derivation problem on the grammar to a meet-over-all-paths problem on its TDG.

Intraprocedural Iteration. We first consider TDGs of a special form: consider the intraprocedural case, i.e., there are no hyperedges in the TDG (and correspondingly no push rules in the PDS). As an example, assume that the TDG in Fig. 4 has only the part corresponding to procedure `foo()` without any hyperedges. In such a TDG, if an edge $((t_1), t)$ was inserted because of the production $t \rightarrow g(t_1)$ for $g = \lambda x.x \otimes w$ for some weight w, then label this edge with w. Next, insert a special node t_s into the TDG and for each production of the form $t \rightarrow g(\epsilon)$ with $g = w$, insert the edge $((t_s), t)$ and label it with weight w. t_s is called a source node. This gives us a graph with weights on each edge. Define the weight of a path in this graph in the standard (but reversed) way: the weight of a path is the extend of weights on its constituent edges in the reverse order. It is easy to see that $\text{MOD}(t) = \bigoplus\{v(\eta) \mid \eta \in path(t_s, t)\}$, where $path(t_s, t)$ is the set of all paths from t_s to t in the TDG and $v(\eta)$ is the weight of the path η. To solve for MOD, we could still use chaotic iteration, but instead we will make use of Tarjan's path-expression algorithm [19].

Problem 1. Given a directed graph G and a fixed vertex s, the **single-source path expression** (SSPE) problem is to compute a regular expression that represents $path(s, v)$ for all vertices v in the graph. The syntax of regular expressions is as follows: $r ::= \emptyset \mid \varepsilon \mid e \mid r_1 \cup r_2 \mid r_1.r_2 \mid r^*$, where e stands for an edge in G.

We can use the SSPE algorithm to compute regular expressions for $path(t_s, t)$, which gives us a compact description of the set of paths we need to consider. Also, the Kleene-star operator identifies loops in the TDG. Let $\otimes^c$ be the reverse of $\otimes$, i.e., $w_1 \otimes^c w_2 = w_2 \otimes w_1$. To compute $\text{MOD}(t)$, we take the regular expression for $path(t_s, t)$ and replace each edge e with its weight, $\emptyset$ with $\overline{0}$, ε with $\overline{1}$, $\cup$ with $\oplus$, . with $\otimes^c$, and solve the expression. The weight w^* is computed as $\overline{1} \oplus w \oplus (w \otimes w) \oplus \cdots$; because of the bounded-height property of the semiring, this iteration converges. The two main advantages of using regular expressions to compute $\text{MOD}(t)$ are: First, loops are identified in the expression, and the evaluation strategy saturates a loop before exiting it. Second, we can compute w^* faster than normal iteration could. For this, observe that

$$(\overline{1} \oplus w)^n = \overline{1} \oplus w \oplus w^2 \oplus \cdots \oplus w^n$$

where exponentiation is defined using $\otimes$, i.e., $w^0 = \overline{1}$ and $w^i = w \otimes w^{(i-1)}$. Then w^* can be computed by repeatedly squaring $(\overline{1} \oplus w)$ until it converges. If $w^* = \overline{1} \oplus w \oplus \cdots \oplus w^n$ then it can be computed in $O(\log n)$ operations. A chaotic-iteration strategy would take $O(n)$ steps to compute the same value. In other words, having a closed representation of loops provides an exponential speedup.[3]

Tarjan's algorithm uses *dominators* to construct the regular expressions for SSPE. This has the effect of computing the weight on the dominators of a node before computing the weight on the node itself. This avoids unnecessary propagation of partial weights to the node (which is the case when you exit a loop too early). Given a graph with m edges (or m grammar productions in our case) and n nodes (or non-terminals), regular expressions for $path(t_s, t)$ can be computed for all nodes t in time $O(m \log n)$ when the graph is *reducible*. Evaluating these expressions will take an additional $O(m \log n \log h)$ semiring operations, where h is the height of the semiring.[4] Because most high-level languages are well-structured, their ICFGs are mostly reducible. When the graph is not reducible, the running time degrades to $O((m \log n + k) \log h)$ semiring operations, where k is the sum of the cubes of the sizes of *dominator-strong components* of the graph. In the worst case, k can be $O(n^3)$. In our experiments, we seldom found irreducibility to be a problem: k/n was a small constant. A pure chaotic-iteration strategy would take $O(m\ h)$ semiring operations in the worst case. Comparing these complexities, we can expect the algorithm that uses path expressions to be much faster than chaotic iteration, and the benefit will be greater as the height of the semiring increases.

Interprocedural Iteration. We now generalize our algorithm to any TDG. For each hyperedge $((t_1, t_2), t)$, delete it from the graph and replace it with the edge $((t_1), t)$.

[3] This assumes that each semiring operation takes the same amount of time. In the absence of any assumption on the semiring being used, we aim to decrease the number of semiring operations. In some cases, e.g., BDD-based weight domains, repeated squaring may not reduce the overall running time. However, the user can supply a procedure for computing w^* whenever there is a more efficient way of computing it than by using simple iteration [13].

[4] The combined sizes of the regular expressions are bounded by the running time of the SSPE algorithm.

This new edge is called a *summary edge*, and node t_2 is called an *out-node*. For example, in Fig. 4 we would delete the hyperedge $((t_4, t_6), t_3)$ and replace it with $((t_4), t_3)$. The new edge is called a summary edge because it crosses a call-site (from a return node to a call node) and will be used to summarize the effect of a procedure call. Node t_6 is an out-node and will supply the procedure summary weight. The resultant TDG is a collection of connected graphs, with each graph roughly corresponding to a procedure. In Fig. 4, the transitions that correspond to procedures `main` and `foo` get split. Each connected graph is called an *intragraph*. For each intragraph, we introduce a source node as before and add edges from the source node to all nodes that have ϵ-productions. The weight labels are also added as before. For a summary edge $((t_1), t)$ obtained from a hyperedge $((t_1, t_2), t)$ with associated production function $g = \lambda x.\lambda y.w \otimes x \otimes y$, label it with $w \otimes t_2$, or $t_2 \otimes^c w$.

This gives us a collection of intragraphs with edges labeled with either a weight or a simple expression with an out-node. To solve for the MOD value, we construct a set of *regular equations*, which we call as out-node equations. For an intragraph G, let t_G be its unique source node. Then, for each out-node t_o in G, construct the regular expression for all paths in G from t_G to t_o, i.e., for $path(t_G, t_o)$. In this expression, replace each edge with its corresponding label. If the resulting expression is r and it contains out-nodes t_1 to t_n, add the equation $t_o = r(t_1, \cdots, t_n)$ to the set of out-node equations. Repeat this for all intragraphs. The resulting set of out-node equations describe all hyperpaths in the TDG to an out-node from the collection of all source nodes. The MOD value of the out-nodes is the greatest fix-point of these equations (with respect to $\sqsubseteq$ of Defn. 3(4)). For example, for the TDG shown in Fig. 4, assuming that t_1 is also an out-node, we would obtain the following out-node equations.[5]

$$t_6 = w_{14}.(w_9 \oplus w_{13}.w_{11}.(w_{12}.w_{10})^*.w_8).w_7.w_6$$
$$t_1 = w_5.w_4.(t_6.w_3).w_2.w_1$$

Here we have used . as a shorthand for $\otimes^c$. One way to solve these equations is by using chaotic iteration: start by initializing each out-node with $\overline{0}$ (the greatest element in the semiring) and update the values of out-nodes by repeatedly solving the equations until they converge. We can give direction to this iteration by constructing a dependence graph of these equation, where an equation $t_o = r(t_1, \cdots, t_n)$ gives rise to dependences $t_i \rightarrow t_o, 1 \leq i \leq n$. We take a *strongly connected component* (SCC) decomposition of this graph and solve all equations in one component before moving to equations in next component (in a topological order). We could also use regular expressions to define an evaluation order on these equations (details are given in [11]), but we chose a simpler implementation because SCCs in this dependence graph, which correspond to mutually recursive procedures, tend to be quite small in practice.

Each regular expression in the out-node equations summarizes all paths in an intragraph and can be quite large. Therefore, we want to avoid evaluating them repeatedly while solving the equations. To this end, we incrementally evaluate the regular expressions: only that part of an expression is reevaluated that contains a modified out-node. (In the algorithm given in Fig. 5, the entire expression may be traversed, but reevaluations are performed selectively.) A regular expression is represented us-

[5] The equations might be different depending on how the SSPE algorithm was implemented, but all such equations would have the same solution.

ing its abstract-syntax tree, where leaves are weights or out-nodes, and internal nodes correspond to $\oplus$, $\otimes$, or *. As a further optimization, all regular expressions share common subtrees, and are represented as DAGs instead of trees. The incremental algorithm we use takes care of this sharing and also identifies modified out-nodes in an expression automatically. At each DAG node we maintain two integers, `last_change` and `last_seen`, as well as the weight `weight` of the subdag rooted at the node. We assume that all regular expressions share the same leaves for out-nodes. We keep a global counter `update_count` that is incremented each time the weight of some out-node is updated. For a node, the counter `last_change` records the last update count at which the weight of its subdag changed, and the counter `last_seen` records the update count at which the subdag was reevaluated. The evaluation algorithm is shown in Fig. 5. When the weight of an out-node is changed, its corresponding leaf node is updated with that weight, `update_count` is incremented, and the out-node's counters are set to `update_count`.

```
1  procedure evaluate(r)
2  begin
3    if r.last_seen == update_count then return
4    case r = w, r = t_o return
5    case r = op(r1,r2)
6      evaluate(r1), evaluate(r2)
7      m = max{r1.last_change, r2.last_change}
8      if m > r.last_seen then
9        w = op(r1.weight, r2.weight)
10       if r.weight ≠ w then
11         r.last_change = m
12         r.weight = w
13     r.last_seen = update_count
14 end
```

Fig. 5. Incremental evaluation algorithm for regular expressions. Here `op` is the prefix version of $\otimes$, $\oplus$, or *. When `op` is *, r_2 can be ignored.

Once we solve for the values of the out-nodes, we can change the out-node labels on summary edges in the intragraphs and replace them with their corresponding weight. Then the MOD values for other nodes in the TDG can be obtained as in the intraprocedural version by considering each intragraph in isolation.

The time required for solving this system of equations depends on reducibility of the intragraphs. Let S_G be the time required to solve SSPE on intragraph G, i.e., $S_G = O(m \log n + k)$ where k is $O(n^3)$ in the worst-case, but is ignorable in practice. If the equations do not have any mutual dependences (corresponding to no recursion) then the running time is $\sum_G S_G \log h$, where the sum ranges over all intragraphs, because each equation has to be solved exactly once. In the presence of recursion, we use the observation that the weight of each subdag in a regular expression can change at most h times while the equations are being solved because it can only decrease monotonically. Because the size of a regular expression obtained from an intragraph G is bounded by S_G, the worst-case time for solving the equations is $\sum_G S_G\ h$. This bound is very pessimistic and is actually worse than that of chaotic iteration. Here we did not make use of the fact that incrementally computing regular expressions is much faster than reevaluating them. For a regular expression with one modified out-node, we only need

to perform semiring operations for each node from the out-node leaf to the root of the expression. For a nearly balanced regular expression tree, this path to the root can be as small as $\log S_G$. Empirically, we found that incrementally computing the expression required many fewer operations than recomputing the expression.

Unlike the chaotic-iteration scheme, where the weights of all TDG nodes are computed, we only need to compute the weights on out-nodes. The weights for the rest of the nodes can be computed lazily by evaluating their corresponding regular expression only when needed. For applications that just require the weight for a few TDG nodes, this gives us additional savings. We also limit the computation of weights of out-nodes to only those intragraphs that contain a TDG node whose weight is required. This corresponds to slicing the out-node equations with respect to the user query, which rules out computation in procedures that are irrelevant to the query.

Handling Local Variables. WPDSs were recently extended to Extended-WPDSs to provide a more convenient mechanism for handling local variables [12]. Reachability problems in EWPDS are also based on abstract grammars similar to the ones for a WPDS. Thus, we can easily adapt our algorithm to EWPDSs as well. Details are given in [11]. We use EWPDSs in our experiments.

4 Solving other PDS Problems

In this section, we give algorithms for some important PDS problems: witness tracing and incremental analysis. Our technical report [11] also gives an algorithm for differential weight propagation. Of these three, only witness tracing and differential propagation have been discussed before for WPDSs [16].

4.1 Witness Tracing

For program-analysis tools, if a program does not satisfy a property, it is often useful to provide a justification of why the property was not satisfied. In terms of WPDSs, it amounts to reporting a set of paths, or rule sequences, that together justify the reported weight for a configuration. Formally, using the notation of Defn. 5, the witness tracing problem for GPP is to find, for each configuration c, a set $\omega(c) \subseteq \bigcup_{c' \in C} path(c, c')$ such that $\bigoplus_{\sigma \in \omega(c)} v(\sigma) = \delta(c)$. This definition of witness tracing does not impose any restrictions on the size of the reported witness set because any compact representation of the set suffices for most applications. The algorithm for witness tracing for GPP [16] requires $O(|Q|^2 \; |\Gamma| \; h)$ memory. Our algorithm only requires $O(|ON| \; h)$ memory, where $|ON|$ is the number of out-nodes, which is expected to be much smaller than $|\Gamma|$.

In our new GPP algorithm, we already have a head start because we have regular expressions that describe all paths in an intragraph. In the intragraphs, we label each edge with not just a weight, but also the rule that justifies the edge. Push rules will be associated with summary edges and pop rules with edges that originate from a source node. Edges from the source node that were inserted because of production (1) in Fig. 3 are not associated with any rule (or with an empty rule sequence). After solving SSPE on the intragraphs, we can replace each edge with the corresponding rule label. This gives us, for each out-node, a regular expression in terms of other out-nodes that captures the set of all rule sequences that can reach that out-node. Next, while solving the regular equations, we record the weights on out-nodes; i.e., when we solve the equa-

tion $t_o = r(t_1, \cdots, t_n)$, we record the weights on $t_1, \cdots, t_n$ — say $w_1, \cdots, w_n$ — whenever the weight on t_o changes to, say, w_o. Then the set of rule sequences to create transition t_o with weight w_o is given by the expression r (where we replace TDG edges with their rule labels) by replacing each out-node t_i with the regular expression for all rule sequences used to create t_i with weight w_i (obtained recursively). This gives a regular expression for the witness set of each out-node. Witness sets for other transitions can be obtained by solving SSPE on the intragraphs by replacing out-node labels with their witness-set expression.

Here we only require $O(|ON|\ h)$ space for recording witnesses because we just have to remember the history of weights on out-nodes. For PDSs obtained from ICFGs and empty initial automaton, $|ON|$ is the number of procedures in the ICFG, which is very small compared to $|\Gamma|$.

4.2 Incremental Analysis

The first incremental algorithm for verifying finite-state properties on ICFGs was given by Conway et al. [4]. We can use the methods presented in this paper to generalize their algorithm to WPDSs. An incremental approach to model checking has the advantage of amortizing the verification time across program development or debugging time.

We consider two cases: addition of new rules and deletion of existing ones. In each case we work at the granularity of intragraphs. When a new rule is added, the fix-point solution of the out-node equations monotonically decreases and we can reuse all of the existing computation. We first identify the intragraphs that changed (have more edges) because of the new rule. Next, we recompute the regular expressions for out-nodes in those intragraphs and add them to the set of out-node equations.[6] Then we solve the equations as before, but set the initial weights of out-nodes to be their existing value. If new out-nodes were added, then set their initial value to $\overline{0}$.

Deletion of a rule requires more work. Again, we identify the changed intragraphs and recompute the out-node equations for them. We call out-nodes in these intragraphs as *modified* out-nodes. Next, we look at the dependence graph of the out-node equations as constructed in §3.2. We perform an SCC decomposition of this graph and topologically sort the SCCs. Then the weights for all out-nodes that appear before the first SCC that has a modified out-node need not be changed. We recompute the solution for other out-nodes in topological order, and stop as soon as the new values agree with previous values. We start with out-nodes in the first SCC that has a modified out-node and solve for their weights. If the new weight of an out-node is different from its previously computed weight, all out-nodes in later SCCs that are dependent on it are marked as modified. We repeat this procedure until there are no more modified out-nodes.

The advantage of doing incremental analysis in our framework is that very little information has to be stored between analysis runs: We only need to store weights on out-nodes.

5 Experiments

We are aware of two implementations of WPDSs: WPDS++ [8] and one used by nMoped [9]. We call the implementation of our algorithm as FWPDS (F stands for "fast"). It can

[6] There are incremental algorithms for SSPE as well, but we have not used them because solving SSPE for a single intragraph is usually very fast.

be plugged-in as a back-end for each of the WPDS libraries. WPDS++ also supports an optimized iteration strategy where the user can supply a priority-ordering on stack symbols, which is used by chaotic iteration to choose the transition with least priority first. We refer to this version as BFS-WPDS++ and supply it with a breadth-first ordering on the ICFG obtained by treating it as a graph. BFS-WPDS++ almost always performs better than WPDS++.

To measure end-to-end performance, FWPDS only computes the weight on transitions required by the application. We also report the time taken to compute the weight on all transitions and refer to this as FWPDS-Full. A comparison with FWPDS-Full will give an indication of "application-independent" improvement provided by our approach because it computes the same amount of information as the previous WPDS algorithms. However, we measure speedups using FWPDS running times to show the potential of using lazy-evaluation in real settings. FWPDS-Full uses a left-associative evaluation order for computing weights of regular expressions. It is also worth noting that repeated squaring for computing w^* did not cause any appreciable difference compared with using a simple iterative method.

We tested FWPDS on three applications that use WPDSs. In each, we perform GPS on the WPDS with the entry point of the program as the initial configuration. The first application performs affine-relation analysis (ARA) on x86 programs [12]. An x86 program is translated into a WPDS to find affine relationships between machine registers. The application only requires affine relationships at certain branch points [1]. Some of the results are shown in Table 1. Over all the experiments we performed, FWPDS provided an average speedup of 1.6 times (i.e., reduced running time by 38%) over BFS-WPDS++.

			Time (s)				Speedup
Prog	Insts	Procs	WPDS++	BFS-WPDS++	FWPDS-Full	FWPDS	
print	75539	697	1.23	1.02	0.77	0.41	2.48
finger	96123	893	11.14	7.94	7.13	4.44	1.79
winhlp32	157634	6491	25.51	19.61	17.32	11.00	1.78
regsvr32	225857	9625	58.70	38.83	37.15	24.65	1.57
cmd	230481	2317	69.19	46.33	52.38	34.87	1.33
notepad	239408	2911	54.08	40.8	41.85	26.50	1.54

Table 1. Comparison of ARA results. The last column show the speedup (ratio of running times) of FWPDS versus BFS-WPDS++. The programs are common Windows executables, and the experiments were run on 3.2 Ghz P4 machine with 4GB RAM.

The second application, BTRACE, is for debugging [10]. It performs path optimization on C programs: given a set of ICFG nodes, called critical nodes, it tries to find a shortest ICFG path that touches the maximum number of these nodes. The path starts at the entry point of the program and stops at a given failure point in the program. FWPDS only computes the weight at the failure point. As shown in Table 2, FWPDS performs much better than BFS-WPDS++ for this application, and the overall speedup was 4.3 times. Some experimental results on incremental analysis for BTRACE are presented in [11]: We observed a roughly 10-fold improvement by incrementally computing the solution after a deleted procedure was reinserted in the program.

The third application is nMoped [9], which is a model checker for Boolean programs. It uses a WPDS library for performing reachability queries. Weights are binary

			Time (s)			Speedup
Prog	ICFG nodes	Procs	BFS-WPDS++	FWPDS-Full	FWPDS	
make	40667	204	15.1	7.7	5.8	2.58
indent	28155	104	19.6	28.2	15.9	1.24
less	33006	359	22.4	8.6	5.3	4.19
patch	27389	133	70.2	23.2	17.1	4.09
gawk	86617	401	72.7	64.5	45.1	1.61
wget	44575	399	318.4	58.9	27.0	11.77

Table 2. Comparison of BTRACE results. The last column shows speedup of FWPDS over BFS-WPDS++. The critical nodes were chosen at random from ICFG nodes and the failure site was set as the exit point of the program. The programs are common Unix utilities, and the experiments were run on 2.4 GHz P4 machine with 4GB RAM.

relations on valuations of Boolean variables, and are represented using BDDs. We measure the performance of FWPDS against this library using a set of programs (and an error configuration for each program) supplied by S. Schwoon. We compute the set of all variable valuations that can hold at the error configuration by computing its meet-over-all-paths weight. As shown in Table 3, FWPDS is 2 to 5 times faster than nMoped. Our technical report [11] gives some other set of experiments, but they were on much smaller programs and led to inconclusive results.

nMoped can also be asked to stop as soon as it finds out that the error configuration is reachable (instead of exploring all paths leading to the error configuration). In that case, when the error configuration was reachable, nMoped performed much better than FWPDS, often completing in less than a second. This is expected because the evaluation strategy used by FWPDS is oriented towards finding the complete weight (MOD value) on a transition. For example, it might be better to avoid saturating a loop completely and propagate partially computed weights in the hope of finding out if the error configuration is reachable. However, when the error configuration is unreachable, or when the abstraction-refinement mode in nMoped is turned on, it explores all paths in the program and computes the MOD value of all transitions. In such situations, it may be better to use FWPDS.

Prog	nMoped	FWPDS-Full	FWPDS	Speedup
bugs5	13.11	13.03	7.25	1.81
slam-fixed	32.67	19.23	13.3	2.46
slam	6.32	5.21	3.27	1.93
unified-serial	37.10	19.65	12.46	2.98
iscsi1	29.15	27.12	14.08	2.07
iscsi10	178.22	59.63	31.29	5.70

Table 3. nMoped results. The last column shows speedup of FWPDS over nMoped. The programs were provided by S. Schwoon, and are not yet publically available.

6 Related Work

The basic strategy of using a regular expression to describe a set of paths has been used previously for dataflow analysis [20] of single-procedure programs. The only work that we are aware of that uses this technique for multi-procedure programs is by Ramalingam [15]. However, he used regular expressions for a particular analysis (execution

frequency analysis) and the technique was motivated by the special requirements of execution frequency analysis when creating procedure summaries, rather than efficiency. We have generalized the approach to apply to a much broader set of problems, namely anything that can be encoded as a WPDS, and showed how various enhancements (incremental recomputation of regular expressions, computing lazily, etc.) contribute to creating a faster analysis.

There has been a host of previous work on incremental program analysis as well as on interprocedural automaton-based analysis [4]. The incremental algorithm we have presented is similar to the algorithm in [4], but generalizes it to WPDSs and is thus applicable in domains other than finite-state property verification. A key difference with their algorithm is that they explore the property automaton on-the-fly as the program is explored. Our encoding into a WPDS requires the whole automaton before the program is explored. This difference can be significant when the automaton is large but only a small part of the automaton needs to be generated.

References

1. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Concurrency Theory (CONCUR)*, pages 135–150, 1997.
3. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, pages 62–73, 2003.
4. C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *CAV*, pages 449–461, 2005.
5. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, pages 232–247, 2000.
6. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *CAV*, pages 324–336, 2001.
7. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.
8. N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems, 2005. http://www.cs.wisc.edu/wpis/wpds++.
9. S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. nMoped, 2005. http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/nmoped/.
10. A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In *European Symposium On Programming*, pages 246–263, 2006.
11. A. Lal and T. Reps. Improving pushdown system model checking. Technical Report 1552, University of Wisconsin-Madison, Jan. 2006.
12. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, pages 434–448, 2005.
13. Y. Matsunaga, P. C. McGeer, and R. K. Brayton. On computing the transitive closure of a state transition relation. In *Design Automation Conference (DAC)*, pages 260–265, 1993.
14. E. W. Myers. A precise interprocedural data flow algorithm. In *POPL*, pages 219–230, 1981.
15. G. Ramalingam. Data flow frequency analysis. In *PLDI*, pages 267–277, 1996.
16. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 2005.
17. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.
18. S. Schwoon. Moped, 2002. http://www.fmi.uni-stuttgart.de/szs/tools/moped/.
19. R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.
20. R. E. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, 1981.