Computer Sciences Department

BTrace: Path Optimization for Debugging

Akash Lal
Junghee Lim
Marina Polishchuk
Ben Liblit

Technical Report #1535

October 2005

UNIVERSITY OF
WISCONSIN
MADISON

BTrace: Path Optimization for Debugging*

Akash Lal
<akash@cs.wisc.edu>

Junghee Lim
<junghee@cs.wisc.edu>

Marina Polishchuk
<mpoli@cs.wisc.edu>

Ben Liblit
<liblit@cs.wisc.edu>

Computer Sciences Department
University of Wisconsin-Madison

October 17, 2005

Abstract

We present and solve a path optimization problem on programs. Given a set of program nodes, called critical nodes, we find a shortest path through the program's control flow graph that touches the maximum number of these nodes. Control flow graphs over-approximate real program behavior; by adding dataflow analysis to the control flow graph, we narrow down on the program's actual behavior and discard paths deemed infeasible by the dataflow analysis. We derive an efficient algorithm for path optimization based on weighted pushdown systems. We present an application for path optimization by integrating it with the Cooperative Bug Isolation Project (*CBI*), a dynamic debugging system. CBI mines instrumentation feedback data to find suspect program behaviors, called bug predictors, that are strongly associated with program failure. Instantiating critical nodes as the nodes containing bug predictors, we solve for a shortest program path that touches these predictors. This path can be used by a programmer to debug his software. We present some early experience on using this hybrid static/dynamic system for debugging.

1 Introduction

Static analysis of programs has been used for a variety of purposes including compiler optimizations, verification of safety properties, and improving program understanding. Static analysis has the advantage of considering all possible executions of a program and therefore gives strong guarantees on the program's behavior. In this paper, we present a static analysis technique that can be

*Supported in part by ONR under contracts N00014-01-1-0796 and N00014-01-1-0708, NSF under grant CCR-0305387, and a gift from Microsoft Research.

used for finding a program execution sequence that is optimal with respect to some criteria. In particular, given a target set of program locations, which we call *critical nodes*, we consider all possible execution traces through the program to find a trace that touches the maximum number of these critical nodes and has the shortest length among all such traces. Since reachability in programs is undecidable in general, we over-approximate the set of all possible traces through a program by considering all paths in the program's control flow graph, and solve the optimization problem on this graph. We also consider how dataflow analysis [2] can be added to the control flow graph to narrow down on the program's actual behavior by discarding paths deemed infeasible by the dataflow analysis. We show that the powerful framework of weighted pushdown systems [23] can be used to represent and solve several variations of the path optimization problem.

As a primary motivating application, we have implemented our path optimization algorithm and integrated it with the Cooperative Bug Isolation Project (*CBI*) [15] to create the BTrace debugging support tool. CBI is an automated bug isolation system. It adds lightweight dynamic instrumentation to software to gain information about runtime behavior. Using this information, it identifies certain suspect program behaviors, called *bug predictors*, that are strongly associated with program failure. These bug predictors help to identify the causes and circumstances of failure, and have been used successfully to find previously unknown bugs [18]. CBI is primarily a dynamic system based on mining feedback data from observed runs. Our work on BTrace represents the first major effort to combine CBI's dynamic, statistical approach with more formal, static program analysis.

BTrace enhances CBI output by giving more context for interpreting the bug predictors. We use CBI bug predictors as our set of critical nodes and construct a path from the entry point of the program to the failure site that touches the maximum number of these bug predictors. CBI associates a numerical score with each bug predictor, with higher scores denoting stronger association with failure. We therefore extend BTrace to find a shortest path that maximizes the sum of prediction scores of the predictors it touches. That is, BTrace finds a path such that the sum of predictor scores of all predictors on the path is maximal, and no shorter path has the same score. We also allow the user to add constraints in the form of the stack traces left behind by the failed program to restrict attention to paths that have unfinished calls exactly in the order they appear in the stack trace, and ordering constraints that restrict the order in which predictors can be touched. These constraints improve the utility of BTrace for debugging purposes by producing a path that is close enough to the actual failing execution of the program to give the user substantial insight into the root causes of failure. We present preliminary experimental results in Section 5 to support this claim. Constructing a shortest path allows the programmer to look at the smallest piece of code necessary to find the bug.

Under the extra constraints described above, the path optimization problem solved by BTrace can be stated as follows:

THE BTRACE PROBLEM. *Given the control flow graph (N, E) of a program with nodes N and edges E; a single node $n_f \in N$ (representing the crash site of a program); a set of critical nodes $B \subseteq N$ (representing the bug predictors); and a function $\mu : B \to \mathbb{R}$ (representing predictor scores), find a path in the control flow graph that first maximizes $\sum_{n \in S} \mu(n)$ where $S \subseteq B$ is the set of critical nodes that the path touches and then minimizes its length. Furthermore, restrict the search for this optimal path to only those paths that satisfy the following constraints:*

1. ***Stack trace.*** *Given a stack trace, consider only those paths that reach n_f with unfinished calls exactly in the order they appear in the stack trace.*

2. ***Ordering.*** *Given a list of node pairs (n_i, m_i) where $n_i, m_i \in B$ and $0 \leq i \leq k$ for some k, consider only those paths that do not touch node m_i before node n_i.*

3. ***Dataflow.*** *Given a dataflow analysis framework, consider only those paths that are not ruled out as infeasible by the dataflow analysis. The requirements on the dataflow analysis framework are specified later in Section 3.5.*

Finding a feasible path through a program when one exists is, in general, undecidable. Therefore, even with powerful dataflow analysis, BTRACE can return an infeasible path that can never appear in any real execution of the program. We consider this acceptable as we judge the usefulness of a path by how much it helps a programmer debug his program, rather than its feasibility.

The key contributions of this paper are as follows:

- We present an algorithm that optimizes path selection in a program according to the criteria described above. We use weighted pushdown systems to provide a common setting under which all of the mentioned optimization constraints can be satisfied.

- We describe a hybrid static/dynamic system that combines optimal path selection with CBI bug predictors to support debugging.

The remainder of the paper is organized as follows: Section 2 presents a formal theory for representing paths in a program. Section 3 derives our algorithm for finding an optimal path. Section 4 gives additional background on CBI and considers how the path optimization algorithm can be used in conjunction with CBI for debugging programs. Section 5 presents experimental results. Section 6 discusses some of the related work in this area, and Section 7 concludes with some final remarks and future work.

2 Describing Paths in a Program

This section introduces the basic theory behind our approach. In Section 2.1, we formalize the set of paths in a program as a pushdown system. Next, in Sec-

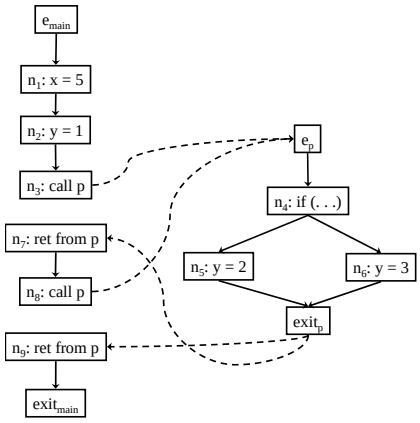


Figure 1: An interprocedural control flow graph. The e and *exit* nodes represent entry and exit points of procedures, respectively. Dashed edges represent interprocedural control flow.

tion 2.2, we introduce weighted pushdown systems that have the added ability to associate a value with each path.

2.1 Paths in a Program

A control flow graph (CFG) of a program is a graph where nodes are program statements and edges represent possible flow of control between statements. Figure 1 shows the CFG of a program with two procedures. We adopt the convention that each function call in the program is represented by two nodes: one is the source of an interprocedural call edge to the callee's entry node and the second is the target of an interprocedural return edge from the callee's exit node back to the caller. In Figure 1, nodes n_3 and n_7 represent one call from *main* to p; nodes n_8 and n_9 represent a second call.

Not all paths (sequences of nodes connected by edges) in the CFG are valid. For example, the path

$$[e_{main}\ n_1\ n_2\ n_3\ e_p\ n_4\ n_5\ exit_p\ n_9]$$

is invalid because the call at node n_3 should return to node n_7, not node n_9. In general, the valid paths in a CFG are described by a context-free language of matching call/return pairs: for each call, only the matching return edge can be taken at the exit node. For this reason, it is natural to use pushdown systems to describe paths in a program [13, 23].

Definition 1. *A **pushdown system** is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where P is the set of states, Γ is the set of stack symbols and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is the set of pushdown rules. A rule $r = (p, \gamma, q, u) \in \Delta$ is written as $\langle p, \gamma \rangle \hookrightarrow \langle q, u \rangle$.*

$$
\begin{aligned}
r_1 &= \langle p, e_{main}\rangle \hookrightarrow \langle p, n_1\rangle & \qquad r_9 &= \langle p, e_p\rangle \hookrightarrow \langle p, n_4\rangle \\
r_2 &= \langle p, n_1\rangle \hookrightarrow \langle p, n_2\rangle & \qquad r_{10} &= \langle p, n_4\rangle \hookrightarrow \langle p, n_5\rangle \\
r_3 &= \langle p, n_2\rangle \hookrightarrow \langle p, n_3\rangle & \qquad r_{11} &= \langle p, n_4\rangle \hookrightarrow \langle p, n_6\rangle \\
r_4 &= \langle p, n_3\rangle \hookrightarrow \langle p, e_p\ n_7\rangle & \qquad r_{12} &= \langle p, n_5\rangle \hookrightarrow \langle p, exit_p\rangle \\
r_5 &= \langle p, n_7\rangle \hookrightarrow \langle p, n_8\rangle & \qquad r_{13} &= \langle p, n_6\rangle \hookrightarrow \langle p, exit_p\rangle \\
r_6 &= \langle p, n_8\rangle \hookrightarrow \langle p, e_p\ n_9\rangle & \qquad r_{14} &= \langle p, exit_p\rangle \hookrightarrow \langle p, \varepsilon\rangle \\
r_7 &= \langle p, n_9\rangle \hookrightarrow \langle p, exit_{main}\rangle & & \\
r_8 &= \langle p, exit_{main}\rangle \hookrightarrow \langle p, \varepsilon\rangle & &
\end{aligned}
$$

Figure 2: A pushdown system that models the control flow graph shown in Figure 1

A pushdown system is a finite automaton with a stack (Γ^*). It does not take any input, as we are interested in the transition system it describes, not the language it generates.

DEFINITION 2. *A **configuration** of a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ is a pair $\langle p, u\rangle$ where $p \in P$ and $u \in \Gamma^*$. The rules of the pushdown systems describe a **transition relation** $\Rightarrow$ on configurations as follows: if $r = \langle p, \gamma\rangle \hookrightarrow \langle q, u\rangle$ is some rule in Δ, then $\langle p, \gamma u'\rangle \Rightarrow \langle q, uu'\rangle$ for all $u' \in \Gamma^*$.*

Let (N, E) be a CFG, where N is the set of nodes and E is the set of edges. Then a pushdown system (P, Γ, Δ) for the CFG can be constructed as follows: $P = \{p\}$, $\Gamma = N$ and Δ is constructed from the following rules:

1. For each intraprocedural edge $(n, m) \in E$, add the rule $\langle p, n\rangle \hookrightarrow \langle p, m\rangle$.
2. For each exit point $n \in N$ of any procedure, add the rule $\langle p, n\rangle \hookrightarrow \langle p, \varepsilon\rangle$.
3. For each interprocedural call edge $(n, m) \in E$, where n is the call site and m the entry point of callee, if the corresponding return site of n is n_r, add the rule $\langle p, n\rangle \hookrightarrow \langle p, m\ n_r\rangle$.

Figure 2 shows an example of this construction for the CFG in Figure 1. Note that we just use a single state. It is not difficult to see that the transition system of a pushdown system obtained in this fashion can describe all valid paths in the CFG. Figure 3 shows a path in the CFG and the corresponding transitions in the pushdown system. A path in the transition system ending in a configuration $\langle p, n_1\ n_2 \cdots n_k\rangle$, where $n_i \in \Gamma$, is said to have a *stack trace* of $\langle n_1, \cdots n_k\rangle$: it describes a path in the CFG that is currently at n_1 and has unfinished calls corresponding to the return sites $n_2, \cdots n_k$. In this sense, a configuration stores an abstract run-time stack of the program, and the transition system describes valid changes that the program can make to it.

$$
\begin{array}{ll}
(a) & [e_{main}\ n_1\ n_2\ n_3\ e_p\ n_4\ n_5\ exit_p\ n_7] \\
(b) & \langle p, e_{main}\rangle \stackrel{r_1}{\Longrightarrow} \langle p, n_1\rangle \stackrel{r_2}{\Longrightarrow} \langle p, n_2\rangle \stackrel{r_3}{\Longrightarrow} \langle p, n_3\rangle \stackrel{r_4}{\Longrightarrow} \\
 & \langle p, e_p\ n_7\rangle \stackrel{r_9}{\Longrightarrow} \langle p, n_4\ n_7\rangle \stackrel{r_{10}}{\Longrightarrow} \langle p, n_5\ n_7\rangle \stackrel{r_{12}}{\Longrightarrow} \\
 & \langle p, exit_p\ n_7\rangle \stackrel{r_{14}}{\Longrightarrow} \langle p, n_7\rangle
\end{array}
$$

Figure 3: (a) A path in the CFG shown in Figure 1, and (b) the corresponding path in the pushdown system of the CFG. The superscripts on $\Rightarrow$ are the rules, from Figure 2, used to justify the particular transition.

Having created a pushdown system, we need to associate a value with each path that stores the set of bug predictors touched by the path. Along with this set, we also need to store the length of a path in order to select the shortest path in case two or more paths touch the same predictors. We accomplish this in Section 3 using weighted pushdown systems, which we describe in the next section.

2.2 Weighted Pushdown Systems

A weighted pushdown system (WPDS) is obtained by associating a *weight* with each pushdown rule. The weights must come from a set that satisfies bounded idempotent semiring properties [4, 23].

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 (i.e., for all $a \in D$, $a \oplus a = a$).*
2. *$(D, \otimes)$ is a monoid with $\overline{1}$ as its neutral element .*
3. *$\otimes$ distributes over $\oplus$, i.e., for all $a, b, c \in D$ we have*

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)\ .$$

4. *$\overline{0}$ is an annihilator with respect to $\otimes$, i.e., for all $a \in D$, $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$.*
5. *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 $f : \Delta \to D$ is a map that assigns a weight to each pushdown rule.*

The $\otimes$ operation is used to compute the weight of concatenating two paths and the $\oplus$ operation is used to compute the weight of merging parallel paths. If $\sigma = [r_1, r_2, \cdots, r_n] \in \Delta^*$ is a sequence of rules, then define the value of σ as $val(\sigma) = f(r_1) \otimes f(r_2) \otimes \cdots \otimes f(r_n)$. In Definition 3, item 3 is required by WPDSs to efficiently explore all paths, and item 5 is required for termination of the search for optimal path.

For pushdown configurations c and c', let $path(c, c')$ be the set of all rule sequences that transform c into c'. Let $n\Gamma^* \subseteq \Gamma^*$ denote the set of all stacks that start with n. Existing work on WPDSs allows us to solve the following problems [23]:

DEFINITION 5. *Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a weighted pushdown system, where $\mathcal{P} = (P, \Gamma, \Delta)$, and let $c' \in P \times \Gamma^*$ be a configuration. The **generalized pushdown predecessor ($GPP_{c'}$) problem** is to find for each $c \in P \times \Gamma^*$ and each $n \in \Gamma$:*

- $\delta(c) \stackrel{\text{def}}{=} \bigoplus\{\, val(\sigma) \mid \sigma \in path(c, c')\}$
- *a **witness set** of paths $\omega(c) \subseteq path(c, c')$ such that $\bigoplus_{\sigma \in \omega(c)} val(\sigma) = \delta(c)$.*
- $\delta(n\Gamma^*) \stackrel{\text{def}}{=} \bigoplus\{\, val(\sigma) \mid \sigma \in path(c, c'), c \in P \times (n\Gamma^*)\}$
- *a **witness set** of paths $\omega(n\Gamma^*) \subseteq \bigcup_{c \in P \times (n\Gamma^*)} path(c, c')$ such that $\bigoplus_{\sigma \in \omega(c)} val(\sigma) = \delta(c)$.*

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

- $\delta(c) \stackrel{\text{def}}{=} \bigoplus\{\, val(\sigma) \mid \sigma \in path(c', c)\}$
- *a **witness set** of paths $\omega(c) \subseteq path(c', c)$ such that $\bigoplus_{\sigma \in \omega(c)} val(\sigma) = \delta(c)$.*
- $\delta(n\Gamma^*) \stackrel{\text{def}}{=} \bigoplus\{\, val(\sigma) \mid \sigma \in path(c', c), c \in P \times (n\Gamma^*)\}$
- *a **witness set** of paths $\omega(n\Gamma^*) \subseteq \bigcup_{c \in P \times (n\Gamma^*)} path(c', c)$ such that $\bigoplus_{\sigma \in \omega(c)} val(\sigma) = \delta(c)$.*

The above problems can be considered as backward and forward reachability problems, respectively. Each aims to find the combine of values of all paths between a given pair of configurations ($\delta(c)$). Along with this value, we can also find a witness set of paths $\omega(c)$ that together justify the reported value for $\delta(c)$. This set of paths is always finite because of item 5 in Definition 3. Note that the reachability problems do not require finding the *smallest* witness set, but the WPDS algorithms always find a finite set.

We have only presented a restricted form of the reachability problems. In general, WPDS can find the value of $\delta(L)$ for any regular set of configurations $L \subseteq P \times \Gamma^*$. The more restricted form presented here is sufficient to capture our path finding problem; the next section presents this construction.

3 Finding an Optimal Path

Here we apply the general theory of Section 2 to the specific BTrace problem defined in Section 1. We begin by developing a solution to the basic path optimization problem without considering dataflow or ordering constraints. We then add these constraints back one by one and show how the basic solution can be extended to accommodate these additional features.

3.1 Creating a WPDS

Let (N, E) be a CFG and $\mathcal{P} = (P, \Gamma, \Delta)$ be a pushdown system representing its paths, constructed as described in Section 2.1. Let $B \subseteq N$ be the set of critical nodes. We will use this notation throughout this section. We now construct a WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ that can be solved to find the best path.

For each path, we need to keep track of its length and also the set of critical nodes it touches. Let $V = 2^B \times \mathbb{N}$ be a set whose elements each consist of a subset of B (the critical nodes touched) and a natural number (the length of the path). We want to associate each path with an element of V. Define the weight domain for $\mathcal{W}$ as follows:

Definition 6. *Let $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ be a bounded idempotent semiring where each component is defined as follows:*

- *The set of weights D is 2^V, the power set of V.*
- *For $w_1, w_2 \in D$, define $w_1 \oplus w_2$ as $reduce(w_1 \cup w_2)$, where*

$$reduce(A) = \{(b, v) \in A \mid \nexists (b, v') \in A \text{ with } v' < v\}$$

- *For $w_1, w_2 \in D$, define $w_1 \otimes w_2$ as*

$$reduce(\{(b_1 \cup b_2, v_1 + v_2) \mid (b_1, v_1) \in w_1, (b_2, v_2) \in w_2\})$$

- *The semiring constants $\overline{0}, \overline{1} \in D$ are*

$$\begin{aligned} \overline{0} &= \emptyset \\ \overline{1} &= \{(\emptyset, 0)\} \end{aligned}$$

The semiring weight domain needs to be able to represent the weight of a set of paths. This is accomplished by defining a weight as a set of elements from V. The combine operation simply takes a union of the weights, but eliminates an element if there is a better element around, i.e., if there are elements (b, v_1) and (b, v_2), the one with shorter path length is chosen. This drives the WPDS to only consider paths with shortest length. The extend operation takes a union of the critical nodes and sums up path lengths for each pair of elements from the two weights. This reflects the fact that when a path with length v_1 that touches the critical nodes in b_1 is extended with a path of length v_2 that touches

the critical nodes in b_2, we get a path of length $v_1 + v_2$ that touches the critical nodes in $b_1 \cup b_2$. The semiring constant $\overline{0}$ denotes an infeasible path, and the constant $\overline{1}$ denotes an empty path that touches no critical nodes and crosses zero graph edges.

To complete the description of the WPDS $\mathcal{W}$, we need to associate each pushdown rule with a weight. If $r = \langle p, n\rangle \hookrightarrow \langle p, u\rangle \in \Delta$, then associate it with the following weight:

$$f(r) = \begin{cases} \{(\emptyset, 1)\} & \text{if } n \notin B \\ \{(\{n\}, 1)\} & \text{if } n \in B \end{cases}$$

Whenever the rule r is used, the length of the path is increased by one and the set of critical nodes grows to include n if n is a critical node. It is easy to see that for a sequence of rules $\sigma \in \Delta^*$ that describes a path in the CFG, $val(\sigma) = \{(b, v)\}$ where b is the set of critical nodes touched by the path and v is its length. Note that by starting with the weights associated with pushdown rules and using the semiring operations $\otimes$ and $\oplus$, we can only create weights that contain at most one element for each subset of the set of all critical nodes. In particular, the *reduce* operation ensures that we only store a shortest path that touches a given set of critical nodes.

3.2 Solving the WPDS

An optimal path can be found by solving the generalized pushdown reachability problems on this WPDS. We consider two scenarios here: when we have the crash site but do not have the stack trace of the crash, and when both the crash site and stack trace are available. We start with just the crash site. Let $n_e \in N$ be the entry point of the program, and $n_f \in N$ the crash site.

THEOREM 1. *In $\mathcal{W}$, solving $GPS_{\langle p, n_e\rangle}$ gives us the following values: $\delta(n_f\Gamma^*) = \{(b, v) \in V \mid$ there is a path from n_e to n_f that touches exactly the critical nodes in b, and the shortest such path has length v $\}$. Moreover, $\omega(n_f\Gamma^*)$ is a set of paths from n_e to n_f such that there is at least one path for each $(b, v) \in \delta(n_f\Gamma^*)$ that touches exactly the critical nodes in b and has length v.*

The above theorem holds because $paths(\langle p, n_e\rangle, \langle p, n_f\Gamma^*\rangle)$ is nothing but the set of paths from n_e to n_f, which may or may not have unfinished calls. Taking a combine over the values of such paths selects, for some subsets $b \subseteq B$, a shortest path that touches exactly the critical nodes in b, and discards the longer ones. The witness set must record paths that justify the reported value of $\delta(n_f\Gamma^*)$. Since the value of a path is a singleton-set weight, it must have at least one path for each member of $\delta(n_f\Gamma^*)$.

When we have a stack trace available as some $s \in (n_f\Gamma^*)$, with n_f being the top-most element of s, we can use either *GPS* or *GPP*.

THEOREM 2. *In $\mathcal{W}$, solving $GPS_{\langle p, n_e\rangle}$ ($GPP_{\langle p, s\rangle}$) gives us the following values for $W_\delta = \delta(\langle p, s\rangle)$ ($\delta(\langle p, n_e\rangle)$) and $W_\omega = \omega(\langle p, s\rangle)$ ($\omega(\langle p, n_e\rangle)$): $W_\delta = \{(b, v) \in V \mid$ there is a valid path from n_e to n_f with stack trace s that touches all critical*

nodes in b, and the shortest such path has length v $\}$. W_ω = *a set of paths from* n_e *to* n_f*, each with stack trace s such that there is at least one path for each* $(b, v) \in W_\delta$ *that touches exactly the critical nodes in b and has length v.*

The above theorem allows us to find the required values using either *GPS* or *GPP*. The former uses forward reachability, starting from n_e and going forward in the program, and the latter uses backward reachability, starting from the stack trace s and going backwards. The next section compares the computational complexity of these two approaches.

Having obtained the above δ and ω values (W_δ and W_ω), we can find an optimal path easily. Let $\mu : B \to \mathbb{R}$ be a user-defined measure that associates a score with each critical node. We compute a score for each $(b, v) \in W_\delta$ by summing up the scores of all critical nodes in b and then choose the pair with highest score. Extracting the path corresponding to that pair in W_ω gives us an optimal path. Some advantages of having such a user-defined measure are the following:

- The user can specify bug predictor scores given by CBI, or make up his own scores.
- The user can give a negative score to critical nodes that should be *avoided* by the path.
- The user can add critical nodes with zero score and use them to specify ordering constraints (Section 3.4).

This lets our tool work interactively with the user to find a suitable path. More generally, we can allow the user to give a measure $\hat{\mu} : (2^B \times \mathbb{N}) \to \mathbb{R}$ that directly associates a score with a path. Using such a measure, the user can decide to choose shorter paths instead of paths that touch more critical nodes. In terms of using path optimization with CBI, this also allows for inserting predictors of multiple bugs and associating a zero score with paths that touch predictors of more than one bug. However, the exponential complexity related with the number of critical nodes, as shown in the next section, suggests that it is better to create multiple WPDSs, one for each bug, instead of inserting all bugs' predictors into the same WPDS.

3.3 Complexity of Solving the WPDS

Each of the methods outlined in Theorems 1 and 2 require solving either *GPS* or *GPP* and then reading the value of $\delta(c)$ for some configuration c. We do not consider the time required for reading the witness value as it can be factored into these two steps. Let $|\Delta|$ be the number of pushdown rules (or the size of the CFG), |Proc| the number of procedures in the program, n_e the entry point of the program, $|B|$ the number of critical nodes, L the length of a shortest path to the most distant CFG node from n_e. The height of the semiring (length of the longest descending chain) we use is $H = 2^{|B|}L$ and the time required to perform each semiring operation is $T = 2^{|B|}$.

To avoid requiring more WPDS terminology, we specialize the complexity results of solving reachability problems on WPDS [23] to our particular use. $GPS_{\langle p,n_e\rangle}$ can be solved in time $O(|\Delta|\ |\text{Proc}|\ H\ T)$ and $GPP_{\langle p,s\rangle}$ requires $O(|s|\ |\Delta|\ H\ T)$ time. Reading the value of $\delta(\langle p, n_e\rangle)$ is constant time and $\delta(\langle p, s\rangle)$ requires $O(|s|\ T)$ time. We can now put these results together.

When no stack trace is available, the only option is to use Theorem 1. Obtaining an optimal path in this case requires time $O(|\Delta|\ |\text{Proc}|\ 2^{2|B|}\ L)$. When a stack trace is available, Theorem 2 gives us two options. Suppose we have k stack traces available to us (corresponding to multiple failures caused by the same bug). In the first option, we solve $GPS_{\langle p,n_e\rangle}$, and then ask for the value of $\delta(\langle p, s\rangle)$ for each stack trace available. This has worst-case time complexity $O(|\Delta|\ |\text{Proc}|\ 2^{2|B|}\ L\ +\ k\ |s|\ 2^{|B|})$ where $|s|$ is the average length of the stack traces. The second option requires a stack trace s, solves $GPP_{\langle p,s\rangle}$ and then asks for the value of $\delta(\langle p, n_e\rangle)$. This has worst-case time complexity $O(k\ |s|\ |\Delta|\ 2^{2|B|}\ L)$. As is evident from these complexities, the second option should be faster, but its complexity grows faster with an increase in k. Note that these are only worst-case complexities, and comparisons based on them need not hold for the average case. In fact, in WPDS++ [11], the WPDS implementation that we use, solving *GPS* is usually faster than solving *GPP*.[1]

Let us present some intuition into complexity results stated above. Consider the CFG shown in Figure 4. If node n_2 is a critical node, then a path from n_1 to n_6 that takes the left branch at n_2 has length 5. The path that takes the right branch has length 4, and touches the same critical nodes as the first path. Therefore, at n_6, the first path can be discarded and we only need to remember the second path. In this way, branching in the program, which increases the total number of paths through the program, only increases the complexity linearly ($|\Delta|$). Now, if node n_3 is also a critical node, then at n_6 we need to remember both paths: one touches more critical nodes and the other has shorter length. (For a path that comes in at n_1, and has already touched n_3, it is better to take the shorter right branch at n_2.) In general, we need to remember a path for each subset of the set of all critical nodes. This is reflected in the design of our weight domain and is what contributes to the exponential complexity with respect to the number of critical nodes.

This exponential complexity is, unfortunately, unavoidable. The reason is that the path optimization problem we are trying to solve is a strict generalization of the traveling salesman problem: our objective is to find a shortest path between two points that touches a given set of nodes.

3.4 Adding Ordering Constraints

We now add ordering constraints to the path optimization problem. Suppose that we have a constraint "node n must be visited before node m," which says that we can only consider paths that do not visit m before visiting n. It is

[1] The implementation does not take advantage of the fact that the PDS has been obtained from a CFG. Backward reachability is easier on CFGs as there is at most one known predecessor of a return-site node.

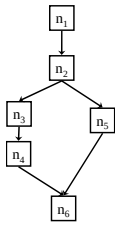


Figure 4: A simple control flow graph

relatively easy to add such constraints to the WPDS given above. The extend operation is used to compute the value of a path. We just change it to yield $\overline{0}$ for paths that do not satisfy the above ordering constraint. For $w_1, w_2 \in D$, redefine $w_1 \otimes w_2$ as *reduce*(A) where

$$A = \left\{ \begin{array}{l|l} (b_1 \cup b_2, v_1 + v_2) & (b_1, v_1) \in w_1, (b_2, v_2) \in w_2, \\ & \neg(m \in b_1, n \in b_2) \end{array} \right\}$$

If $\sigma \in \Delta^*$ is a sequence of rules that describes a path in the CFG, then $val(\sigma) = \{(b, v)\}$ where b is the set of critical nodes visited by the path and v is its length provided it does not visit m before n, and $val(\sigma) = \emptyset = \overline{0}$ if the path does visit m before visiting n. If we have more than one ordering constraint, then we simply add more clauses, one for each constraint, to the above definition of extend. For the use of BTrace as a debugging application, it is essential that we let the user interact with the tool. These constraints provide a way for the user to use his own intuition to guide the optimization problem solved by BTrace.

These constraints do not change the worst case asymptotic complexity of solving reachability problems in WPDS. However they do help prune down the paths that need to be explored, because each constraint cuts down on the size of weights produced by the extend operation.

3.5 Adding Dataflow

So far we have not considered interpreting the semantics of the program other than its control flow. This implies that the WPDS can find infeasible paths: ones that cannot occur in any execution of the program. An example is a path that assigns `x := 1` and then follows the true branch of the conditional `if (x == 0)`. In general, it is undecidable to restrict attention to paths that actually occur in some program execution, but if we can rule out many infeasible paths, we increase the chances of presenting a feasible or near-feasible path to the user. This can be done using dataflow analysis.

Dataflow analysis is carried out to approximate, for each program variable, the set of values that the variable can take at each point in the program. When a dataflow analysis satisfies certain conditions, it can be integrated into a WPDS

by designing an appropriate weight domain [13, 23]. Examples of such dataflow analyses include uninitialized variables, live variables, linear constant propagation [24], and affine relation analysis [21, 22]. In particular, we can use of any bounded idempotent semiring weight domain $\mathcal{S}_d = (D_d, \oplus_d, \otimes_d, \overline{0}_d, \overline{1}_d)$ provided that when given a function $f_d : \Delta \to D_d$ that associates each PDS rule (CFG edge) with a weight, it satisfies the following property: given any (possibly infinite) set $\Sigma \subseteq \Delta^*$ of paths between the same pair of program nodes, we have

$$\bigoplus_{\sigma \in \Sigma} val_d(\sigma) = \overline{0}_d \text{ only if all paths in } \Sigma \text{ are infeasible} \tag{1}$$

where $val_d(\ [r_1, \cdots, r_k]\ ) = f_d(r_1) \otimes_d \cdots \otimes_d f_d(r_k)$. In particular this means that $val_d(\sigma) = \overline{0}_d$ only if σ is an infeasible path, i.e., the path can never be executed in the program. This imposes a soundness guarantee on the dataflow analysis: it can only rule out infeasible paths. The computability of such a dataflow analysis comes from the fact that it can be encoded as a bounded idempotent semiring $\mathcal{S}_d$. We briefly describe how classical dataflow analysis frameworks [2] can be encoded as weight domains. More details can be found in Reps et al. [23].

In classical dataflow analysis, we have a meet semilattice $(\mathcal{D}, \sqcap)$ where

- $\mathcal{D}$ is a set of dataflow facts. Each element $e \in \mathcal{D}$ represents a set of possible program states or memory configurations.
- The meet operator $\sqcap$ is used to combine dataflow facts obtained along different paths.
- $\top \in \mathcal{D}$ is the greatest element in $\mathcal{D}$, i.e., $\top \sqcap e = e$ for all $e \in \mathcal{D}$. It represents the empty set of program states.

Each program statement is associated with a dataflow transformer $\tau : \mathcal{D} \to \mathcal{D}$ that represents the effect of executing the statement on the state of the program. The effect of executing a path σ in the program can then be computed by composing the dataflow transformers associated with each statement. Let pf_σ be the transformer obtained for path σ. Then the dataflow analysis problem is to compute, for each program point n, the "meet-over-all-paths" solution as follows:

$$\mathrm{MOP}_n = \bigsqcap_{\sigma \in paths(n_e, n)} \mathrm{pf}_\sigma(e_s)$$

where n_e is the starting point of the program and $e_s \in \mathcal{D}$ is the dataflow fact representing the set of states at the beginning of the program. MOP_n represents the set of states that can arise at n as it combines the values contributed by each path in the program that leads to n. This value is not always computable but a sufficient condition for computability is that the transfer functions associated with program statements be distributive, i.e., $\tau(e_1 \sqcap e_2) = \tau(e_1) \sqcap \tau(e_2)$ for all $e_1, e_2 \in \mathcal{D}$.

Dataflow analysis, as presented above, can be encoded as a weight domain provided the following conditions are met:

- The dataflow transformers associated with program statements must be chosen from a set $F \subseteq (\mathcal{D} \to \mathcal{D})$ that is closed under meet and composition, i.e., for all $\tau_1, \tau_2 \in F$, $\tau_1 \circ \tau_2 \in F$ and $\tau_1 \sqcap \tau_2 \in F$ where $\tau_1 \sqcap \tau_2 = \lambda e.\tau_1(e) \sqcap \tau_2(e)$.
- All transformers in F must be distributive.
- All transformers in F must be strict in $\top$, i.e., for all $\tau \in F$, $\tau(\top) = \top$.
- F has no infinite descending chains.

The weight domain is $(F, \sqcap, \odot, \lambda e.\top, \lambda e.e)$ where $\tau_1 \odot \tau_2 = \tau_2 \circ \tau_1$. Reps et al. give example of such an encoding for simple constant propagation [23, Section 4.2]. In such a weight domain, the value of a path is the dataflow transformer associated with the path: $val_d(\sigma) = \text{pf}_\sigma$ for any path $\sigma \in \Delta^*$. As dataflow analysis is concerned with computing an over-approximation of the set of states that can arise at a program point, $val_d(\sigma) = \lambda e.\top$ only when the path cannot contribute any set of program states, i.e., it is infeasible. The more general requirement of Equation 1 is also satisfied using a similar reasoning: the combine over the values of a set of paths corresponds to calculating the contribution of those sets of paths on program states, which is empty ($\top$) only when the paths are infeasible. If we consider all paths from the entry of the program to a particular node n, then the combine of values of these paths applied to e_s is simply MOP_n.

Such a translation from dataflow transformers to a weight domain allows us to talk about the meet-over-all-paths between configurations of a pushdown system. For example, solving $GPS_{\langle p, n_e \rangle}$ on this weight domain gives us $\delta(\langle p, n_1 n_2 \cdots n_k \rangle)$ as the combine (or meet) over the values of all paths from n_e to n_1 that have the stack trace $n_1 n_2 \cdots n_k$. This is a unique advantage that we gain over conventional dataflow analysis by using WPDSs.

Assume that $\mathcal{S}_d = (D_d, \oplus_d, \otimes_d, \overline{0}_d, \overline{1}_d)$ is a weight domain that satisfies Equation 1, and that $f_d : \Delta \to D_d$ is a function that associates a dataflow weight with each rule of our pushdown system. We change the weight domain of our WPDS as follows.

DEFINITION 7. *Let $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ be a bounded idempotent semiring where each component is defined as follows:*

- *The set of weights D is $2^{2^B \times \mathbb{N} \times D_d}$, the power set of the set $2^B \times \mathbb{N} \times D_d$.*
- *For $w_1, w_2 \in D$, define $w_1 \oplus w_2$ as $reduce_d(w_1 \cup w_2)$ where $reduce_d(A)$ is defined as*

$$\left\{ \begin{array}{ll} (b, & min\{v_1, \cdots v_n\}, \\ & d_1 \oplus_d \cdots \oplus_d d_n) \end{array} \;\middle|\; \begin{array}{l} (b, v_i, d_i) \in A, \\ 1 \le i \le n \end{array} \right\}$$

- *For $w_1, w_2 \in D$, define $w_1 \otimes w_2$ as $reduce_d(A)$ where A is the set*

$$\left\{ \begin{array}{l|l} (b_1 \cup b_2, v_1 + v_2, d_1 \otimes_d d_2) & \begin{array}{l} (b_1, v_1, d_1) \in w_1, \\ (b_2, v_2, d_2) \in w_2, \\ d_1 \otimes_d d_2 \neq \overline{0}_d, \\ (b_1, b_2) \textit{ satisfy all} \\ \textit{ordering constraints} \end{array} \end{array} \right\}$$

- *The semiring constants $\overline{0}, \overline{1} \in D$ are*

$$\begin{array}{rcl} \overline{0} & = & \emptyset \\ \overline{1} & = & \{(\emptyset, 0, \overline{1}_d)\} \end{array}$$

Here (b_1, b_2) satisfy all ordering constraints iff for each constraint "visit n before m," it is not the case that $m \in b_1$ and $n \in b_2$.

The weight associated with each rule $r = \langle p, n \rangle \hookrightarrow \langle p, u \rangle \in \Delta$ is given by

$$f(r) = \left\{ \begin{array}{ll} \{(\emptyset, 1, f_d(r))\} & \text{if } n \notin B \\ \{(\{n\}, 1, f_d(r))\} & \text{if } n \in B \end{array} \right.$$

Each path is now associated with the set of predictors it touches, its length, and its dataflow value. Infeasible paths are removed during the extend operation as weights with dataflow value $\overline{0}_d$ are discarded. More formally, for a path $\sigma \in \Delta^*$ in the CFG, $val(\sigma) = \{(b, v, w_d)\}$ if $w_d = val_d(\sigma) \neq \overline{0}_d$ is the dataflow weight associated with the path, v is the length of the path, b is the set of critical nodes touched by the path, and the path satisfies all ordering constraints. If σ does not satisfy the ordering constraints or if $val_d(\sigma) = \overline{0}_d$, then $val(\sigma) = \emptyset = \overline{0}$. Analysis using this weight domain is similar to the "property simulation" used in ESP [6], where a distinct dataflow value is maintained for each property-state. We maintain a distinct dataflow weight for each subset of critical nodes.

Instead of repeating Theorems 1 and 2, we just present the case of using *GPS* when the stack trace $s \in (n_f \Gamma^*)$ is available. Results for other cases can be obtained similarly.

THEOREM 3. *In the WPDS obtained from the weight domain of Definition 7, solving $GPS_{\langle p, n_e \rangle}$ gives us the following values:*

- *$\delta(\langle p, s \rangle) = \{(b, v, w_d) \mid$ there is a path from n_e to n_f with stack trace s that visits exactly the critical nodes in b, satisfies all ordering constraints, is not infeasible under the weight domain $\mathcal{S}_d$, and the shortest such path has length v $\}$.*

- *$\omega(\langle p, s \rangle)$ contains at least one path for each $(b, v, w_d) \in \delta(\langle p, s \rangle)$ that goes from n_e to n_f with stack trace s, visits exactly the predictors in b, satisfies all ordering constraints and has length v. More generally, for each $(b, v, w_d) \in \delta(\langle p, s \rangle)$ it will have paths $\sigma_i, 1 \leq i \leq k$ for some constant k such that $val(\sigma_i) = \{(b, v_i, w_i)\}$, $min\{v_1, \cdots, v_k\} = v$, and $w_1 \oplus_d \cdots \oplus_d w_k = w_d$.*

The worst case time complexity in the presence of dataflow increases by a factor of $H_d(C_d + E_d)$ where H_d is height of $\mathcal{S}_d$, C_d is time required for applying $\oplus_d$, and E_d is the time required for applying $\otimes_d$.

Theorem 3 completely solves the BTRACE problem mentioned in Section 1. The next section presents the dataflow weight domain that we used for our experiments along with some extensions that were necessary to increase the practical utility of dataflow analysis in our framework.

3.6 Example and Extensions for Using Dataflow Analysis

3.6.1 Copy Constant Propagation

We now give an example of a weight domain that can be used for dataflow analysis. We encode copy-constant propagation [26] as a weight domain. A similar encoding is used by Sagiv, Reps, and Horwitz [24]. Copy-constant propagation is concerned with determining if a variable has a fixed constant value at some point in the program. It interprets constant-to-variable assignments (`x := 1`) and variable-to-variable assignments (`x := y`) and abstracts all other assignments as `x := ⊥`, which says that `x` does not have a constant value. We ignore conditions on branches for now, i.e., we assume all branches to be non-deterministic. However, for this analysis to be useful in ruling out infeasible paths, we will need to put in conditionals later.

Let *Var* be the set of all global variables of a given program. Let $\mathbb{Z}^\top_\bot = \mathbb{Z} \cup \{\bot, \top\}$ and $(\mathbb{Z}^\top_\bot, \sqcap)$ be the standard constant propagation meet semilattice obtained from the partial order $\bot \sqsubseteq_{cp} c \sqsubseteq_{cp} \top$ for all $c \in \mathbb{Z}$. Then the set of weights of our weight domain is $D_d = Var \to (2^{Var} \times \mathbb{Z}^\top_\bot)$. Here, $\tau \in D_d$ represents a dataflow transfer function that can be used to summarize the effect of a path as follows: if $env : Var \to \mathbb{Z}$ is the state of the program before the path is executed and $\tau(\mathtt{x}) = (\{\mathtt{x}_1, \cdots, \mathtt{x}_n\}, c)$ for $c \in \mathbb{Z}^\top_\bot$, then the value of `x` after the path is executed is $env(\mathtt{x}_1) \sqcap env(\mathtt{x}_2) \cdots \sqcap env(\mathtt{x}_n) \sqcap c$. Let $\tau^{\mathtt{v}}(\mathtt{x})$ be the first component of $\tau(\mathtt{x})$ and $\tau^{\mathtt{c}}(\mathtt{x})$ be the second component. Then we can define the semiring operations as follows: for $\tau_1, \tau_2 \in D_d$,

$$\tau_1 \oplus_d \tau_2 = \lambda \mathtt{x}.(\tau_1^{\mathtt{v}}(\mathtt{x}) \cup \tau_2^{\mathtt{v}}(\mathtt{x}), \tau_1^{\mathtt{c}}(\mathtt{x}) \sqcap \tau_2^{\mathtt{c}}(\mathtt{x}))$$

$$\tau_1 \otimes_d \tau_2 = \lambda \mathtt{x}.(\bigcup_{\mathtt{y} \in \tau_2^{\mathtt{v}}(\mathtt{x})} \tau_1^{\mathtt{v}}(\mathtt{y}), \tau_2^{\mathtt{c}}(\mathtt{x}) \sqcap (\bigsqcap_{\mathtt{y} \in \tau_2^{\mathtt{v}}(\mathtt{x})} \tau_1^{\mathtt{c}}(\mathtt{y})))$$

The combine operation is simply a concatenation of expressions and the extend operation is substitution. For example, if $Var = \{\mathtt{x}, \mathtt{y}\}$ then the weight (or transformer) associated with the statement `x := 1` is $\tau_1 = [\mathtt{x} \mapsto (\emptyset, 1), \mathtt{y} \mapsto (\{\mathtt{y}\}, \top)]$ and the weight associated with the statement `y := x` is $\tau_2 = [\mathtt{x} \mapsto (\{\mathtt{x}\}, \top), \mathtt{y} \mapsto (\{\mathtt{x}\}, \top)]$. Taking their combine gives us $\tau_1 \oplus_d \tau_2 = [\mathtt{x} \mapsto (\{\mathtt{x}\}, 1), \mathtt{y} \mapsto (\{\mathtt{x}, \mathtt{y}\}, \top)]$, and their extend gives us $\tau_1 \otimes_d \tau_2 = [\mathtt{x} \mapsto (\emptyset, 1), \mathtt{y} \mapsto (\emptyset, 1)]$.

The semiring constants are given by $\overline{0}_d = \lambda \mathtt{x}.(\emptyset, \top)$ and $\overline{1}_d = \lambda \mathtt{x}.(\{\mathtt{x}\}, \top)$. This constructs a perfectly valid weight domain $\mathcal{S}_d = (D_d, \oplus_d, \otimes_d, \overline{0}_d, \overline{1}_d)$ for

copy-constant propagation, but it is not very useful to us. The reason is that this weight domain considers all paths to be feasible, which can be seen from the fact that the extend of any two non-zero weights is never $\overline{0}_d$. To remedy this, and disqualify some paths as infeasible, we need to add interpretation for branch conditions.

3.6.2 Handling Conditionals

Handling branch conditions is problematic because dataflow analysis in the presence of conditions is usually very hard. For example, finding whether a branch condition can ever evaluate to true, even for copy-constant propagation, is PSPACE-complete [20]. Therefore, we have to resort to approximate dataflow analysis i.e., we give up on computing meet-over-all-paths. This translates into relaxing the distributivity requirement on the weight domain $\mathcal{S}_d$. Fortunately, WPDSs can handle non-distributive weight domains [23] by relaxing Definition 3 item 3 to the following:

If D is the set of weights, then for all $d_1, d_2, d_3 \in D$,

$$\begin{array}{rcl} d_1 \otimes (d_2 \oplus d_3) & \sqsubseteq & (d_1 \otimes d_2) \oplus (d_1 \otimes d_3) \\ (d_1 \oplus d_2) \otimes d_3 & \sqsubseteq & (d_1 \otimes d_3) \oplus (d_2 \otimes d_3) \end{array}$$

where $\sqsubseteq$ is the partial order defined by $\oplus : d_1 \sqsubseteq d_2$ iff $d_1 \oplus d_2 = d_1$. Under this weaker property, the generalized reachability problems can only be solved approximately, i.e., instead of obtaining $\delta(c)$ for a configuration c, we only obtain a weight w such that $w \sqsubseteq \delta(c)$. For our path optimization problem, this inaccuracy will be limited to the dataflow analysis. We would only eliminate some of the paths that the dataflow can potentially find infeasible and might find a path σ such that $val_d(\sigma) = \overline{0}_d$. This is acceptable because it is not possible to rule out all infeasible paths anyway. Moreover, it allows us the flexibility of putting in a simple treatment for conditions in most dataflow analysis. The disadvantage is that we lose a strong characterization of the type of paths that will be eliminated.

For copy-constant propagation, we bring conditions into the picture by adding weights $\{\rho_e \mid e \text{ is an arithmetic condition}\}$. We associate weight ρ_e with the rule $\langle p, n\rangle \hookrightarrow \langle p, m\rangle$ $(n, m \in \Gamma)$ if the corresponding CFG edge can only execute when e evaluates to true on the program state at n. For example, we associate the weight $\rho_{\mathtt{x}=0}$ with the CFG edge corresponding to the true branch of the conditional `if (x == 0)` and weight $\rho_{\mathtt{x}\neq 0}$ with the false branch. The combine and extend operations are modified as follows: for all $\tau \in D_d$,

$$\begin{array}{rcl} \tau \oplus_d \rho_e & = & \tau \\ \rho_{e_1} \oplus_d \rho_{e_2} & = & \overline{1}_d \\ \tau \otimes_d \rho_e & = & \begin{cases} \overline{0}_d & \text{if } \tau(x_i) = (\emptyset, c_i), c_i \in \mathbb{Z} \text{ for all} \\ & x_i \text{ appearing in } e \text{ and } e[x_i = c_i] \\ & \text{evaluates to false} \\ \tau & \text{otherwise} \end{cases} \\ \rho_{e_1} \otimes_d \rho_{e_2} & = & \rho_{e_1} \\ \rho_e \otimes_d \tau & = & \tau \end{array}$$

This gives a very simple treatment for conditionals.

The extend operation, as defined here, is not associative. This implies a further loss in precision by not being able to evaluate conditions whose variables depend on input parameters of the procedure the condition is in because they only get instantiated at a call. One might choose a more powerful treatment of conditions based on weakest preconditions by associating each weight with a precondition. Then, for example, [`x := y`] $\otimes_d$ [`x == 0`] can be written as [`y == 0; x := y`]. This makes extend associative, at least for simple conditions. We have opted for the previous, simpler treatment for conditionals as it is easier to implement and the benefit from using preconditions in a practical setting is not directly evident. We leave judging the benefit of more precise dataflow analyses as future work.

3.6.3 Handling Local Variables

Another extension we make to our dataflow analysis is the treatment of local variables. A recent extension to WPDSs [13] shows how local variables can be handled by using *merge functions* that allow for local variables to be saved before a call and then merged with the information returned by the callee to compute the effect of the call. This treatment for local variables allows us to restrict each weight to just manage the local variables of only one procedure.

Let *Proc* be the set of all procedures in a given program, *Var* be the set of all global variables, and Var_{pr} be the set of all global variables together with local variables of procedure $pr \in Proc$. Then the set of weights is

$$\begin{aligned} D^l_d = \quad & Var \to (2^{Var} \times \mathbb{Z}^\top_\bot) \\ & \bigcup_{pr \in Proc} (\, Var_{pr} \to (2^{Var_{pr}} \times \mathbb{Z}^\top_\bot)) \\ & \bigcup \{\rho_e \mid e \text{ is an arithmetic condition}\} \end{aligned}$$

The extend and combine operations are the same as defined before, except that when they are applied on weights over different set of local variables, we first drop all local variables and then do the operation on global variables. For conditional weights, if there is a mismatch of variables, we assume that the condition evaluates to true (i.e., we ignore the condition). Next, we have to define the merge operation that will calculate the effect of a call. If τ_1 is over variables Var_{pr_1} and τ_2 is over variables Var_{pr_2}, then define $merge_d$ as follows:

$$\begin{aligned} merge_d(\tau_1, \tau_2) &= (\tau_1 \otimes_d (l^\bot_{pr_2} \otimes_d \tau_2)_{\text{global}}) \oplus_d (\tau_1)_{\text{local}} \\ merge_d(\rho_e, \tau_1) &= \tau_1 \\ merge_d(\tau_1, \rho_e) &= \tau_1 \\ merge_d(\rho_{e_1}, \rho_{e_2}) &= \rho_{e_1} \end{aligned}$$

where $l^\bot_{pr_2}$ is a weight over variables Var_{pr_2} and assigns $(\emptyset, \bot)$ to all local variables and $(\{g\}, \top)$ to all global variables g. It effectively discards local variables of τ_2. $(\tau)_{\text{local}}$ and $(\tau)_{\text{global}}$ remove the assignments for local and global variables from τ, respectively. This merge operation is needed, for example, to conclude that $\mathtt{l} = \mathtt{g}_2$ after a call when $\mathtt{l} = \mathtt{g}_1$ before the call and the callee assigns

$\mathtt{g}_2 = \mathtt{g}_1$. Note that we could not have simply created weights over all (local and global) variables of the program because that would not deal with recursive calls correctly.

The merge function carries over to the semiring of Definition 7 as follows: for $w_1, w_2 \in D$, $merge(w_1, w_2) = reduce(A)$ where A is the set

$$\left\{ (b_1 \cup b_2, v_1 + v_2, d_1 \otimes_d d_2) \;\middle|\; \begin{array}{l} (b_1, v_1, d_1) \in w_1, \\ (b_2, v_2, d_2) \in w_2, \\ merge_d(d_1, d_2) \neq \overline{0}_d, \\ (b_1, b_2) \text{ satisfy all} \\ \text{ordering constraints} \end{array} \right\}$$

Extending WPDSs in this manner does not change Theorem 3, except for the inaccuracies of dataflow analysis that we already had.

4 Finding Bug Predictors

The formalisms of Section 3 may be used for solving a variety of optimization problems concerned with touching key program points along some path. BTrace represents one application of these ideas: an enhancement to the statistical debugging analysis performed by the Cooperative Bug Isolation Project (CBI). In this section we briefly review the existing CBI system, paying particular attention to the kind of data its analysis yields and the ways in which this could be enhanced using path optimization.

4.1 Distributed Data Collection

The Cooperative Bug Isolation Project provides a low-overhead sampling infrastructure for gathering small amounts of information from every run of a program executed by its user community. It inserts instrumentation to test a large number of predicates on program values during execution. A CBI predicate can be thought of as a simple local assertion on program state at a specific code location. Predicates cast a very wide net to catch many different, potentially interesting program behaviors. For example, *branch predicates* record the direction of branches, with two such predicates (true, false) for each conditional in the program. *Return predicates* monitor the sign of function return values, with three such predicates (negative, zero, positive) at each function call site. Additional predicate schemes compare assigned variables to other nearby variables and constants, track the behavior of reference counts, test programmer-specified assertions, etc. A moderate-sized program can easily have hundreds of thousands or even millions of predicates, all injected automatically by the CBI instrumenting compiler [17].

CBI uses sparse random sampling of instrumentation sites, which keeps performance overhead low by spreading the performance cost thinly among a large number of end users. However, it is also fair in a statistically rigorous sense: in aggregate, the sampled behavior is representative of the true, complete program

behavior across all users and runs. In addition, only a few of the predicates collected on a given run have strong predictive power: a moderate-sized program might contain a million distinct predicates, only a dozen of which are actually associated with failure.

4.2 Debugging Support

CBI collects a feedback report from each run that identifies the run as successful or failed (e.g., crashed) and gives the number of times each predicate was observed to be true. Given a collection of such reports, CBI applies a set of *statistical debugging* techniques to generate a ranked list of *bug predictors*: those few predicates that are strongly associated with many program failures. Each bug predictor is assigned a numerical score in $\mathbb{R}^+$ that balances two key factors: (1) how much this predictor increases the probability of failure, and (2) how many failed runs this predictor accounts for. Thus, high-value predictors warrant close examination both because they are highly correlated with failure and because they account for a large portion of the overall failure rate seen by end users [18].

A bug predictor list helps a human programmer identify, perhaps reproduce, and ultimately fix a bug. To provide the most direct correspondence between a predictor list and a bug, the initial bug predictor list is further clustered into lists that contain distinct failure scenarios, by grouping predictors that behave similarly. The high-ranked predictors in a single bug predictor list are the most likely direct failure causes. Lower-ranked predictors in the same list are other program behaviors that are also strongly associated with the same kind of failure; these may help the programmer better understand the circumstances under which the bug appears.

4.3 A Need for Failure Paths

Because predicates are sampled throughout the entire dynamic lifetime of a run, they often reveal suspect behaviors that precede the actual point of failure. This is particularly common in the case of memory corruption bugs, where the bad operation that trashed the heap may have occurred long before an eventual crash inside `malloc()`. The ability to make observations throughout an entire run is a strength of CBI, but it can also make interpreting bug predictors challenging. The programmer must think both forward and backward in time: she must consider not only what earlier events could have caused the predictor to be true, but also what the later consequences may be once the predictor is already true. This contrasts with the task of interpreting a postmortem stack trace, where there are no future events and one only considers how past behavior could have led the program to its terminal state.

For this reason we believe that it is important to join isolated bug predictors into extended failure paths. The BTrace system lets us reconstruct a path from start to halt that hits several high-ranked bug predictors along the

way. When working with a single bug predictor, this path helps guide the programmer's attention forward and backward in time from that critical point. If several bug predictors all relate to the same failure scenario, a feasible path that touches them all can help the programmer draw connections between seemingly unrelated sections of code all of which act in concert to bring the program down.

5 Evaluation

We have implemented BTrace using the WPDS++ library [11]. To manage the exponential complexity in the number of bug predictors, we use abstract decision diagrams (ADDs) to efficiently encode weights. Appendix A contains additional details on how the semiring operations are implemented on ADDs. We use the CUDD library [25] for manipulating ADDs. The CBI instrumenting compiler provides a CFG for the instrumented program, though it does not currently handle indirect function calls and in general may represent only the instrumented portion of a multi-component system.

A BTrace debugging session starts with a list of related bug predictors, believed by CBI to represent a single bug. We designate this list (or some high-ranked prefix thereof) as the critical nodes and insert them at their corresponding locations in the CFG. Branch predictors, however, may be treated as a special case. These predictors associate the direction of a conditional with failure, and therefore can be repositioned on the appropriate branch. For example, if we have a conditional of the form "`if (x == 0)`", and CBI reports that "`x == 0` is false" is predictive of failure at this point, then this bug predictor is moved to the first node in the `else` branch of the conditional. This can be seen as one example of exploiting not just the location but also the semantic meaning of a bug predictor; branch predicates make this easy because their semantic meaning directly corresponds to control flow.

As an optimization, we compress the original program CFG into its basic blocks before converting it to a WPDS. Each basic block is a connected part of the CFG with unique entry and exit nodes, and all nodes inside it are connected in a single straight-line path. The weight on a pushdown rule $r = \langle p, n\rangle \hookrightarrow \langle p, u\rangle \in \Delta$ is now computed as follows: $f(r) = \{(\texttt{BP}(n), \texttt{Len}(n), \texttt{Df}(n))\}$, where $\texttt{BP}(n)$ is the set of bug predictors inside basic block n, $\texttt{Len}(n)$ is the number of nodes inside it, and $\texttt{Df}(n)$ is obtained by taking an extend of dataflow weights associated with nodes in n. This WPDS is still an accurate model of the CFG as a path that visits one node in a basic block is forced to visit all nodes in the basic block, provided it does not stop inside it. The failure site is kept in a basic block of its own and is not joined with other nodes. This, of course, requires that we identify the failure site before constructing the WPDS; we extract this information from CBI feedback reports.

For dataflow analysis, we track all integer- and pointer-valued variables and structure fields. We do not track the contents of memory and any write to memory via a pointer is replaced with non-deterministic ($\perp$) assignments to all variables whose address was ever taken. Direct structure assignments are

expanded into component-wise assignments to all fields of the structure.

Without dataflow, BTrace seeks the shortest available path through any procedure that has no bug predictors. In our experience this does not add to the usefulness of the path because a programmer who understands the code is better equipped to judge the effect of executing the procedure, rather than what the shortest path tells him. Therefore, we allow for the addition of extra edges in the CFG that skip over calls. In Figure 1, for example, extra edges would appear from node n_3 to node n_7 and from node n_8 to n_9, giving BTrace the option to bypass p entirely if no valuable bug predictors can be reached by entering it.

5.1 Case Studies: Siemens Suite

We have applied BTrace to three buggy programs from the Siemens test suite [9]: replace v8, tcas v37, and print_tokens2 v6. These programs do not crash; they merely produce incorrect output. Thus our analysis is performed without a stack trace, instead treating the exit from `main()` as the "failure" point. We find that BTrace can be useful even for non-fatal bugs.

tcas has an array index error in a one-line function that contains no CBI instrumentation and thus might easily be overlooked. Without bug predictors, BTrace produces the shortest possible path that exits `main()`, revealing nothing about the bug. After adding the top-ranked predictor, BTrace isolates lines with calls to the buggy function.

replace has an incorrect function return value. BTrace with the top two predictors yields a path through the faulty statement. Each predictor is located within one of two disjoint chains of function calls invoked from `main()`, and neither falls in the same function as the bug. Thus, while the isolated predictors do not directly reveal the bug, the BTrace failure path through these predictors does.

print_tokens2 has an off-by-one error. Again, two predictors suffice to steer BTrace to the faulty line. Repositioning of branch predictors is critical here. Even with all nineteen CBI-suggested predictors and dataflow analysis enabled, a correct failure path only results if branch predictors are repositioned to steer the path in the proper direction.

5.2 Case Studies: ccrypt and bc

We have also run BTrace on two small open source utilities: ccrypt v1.2 and bc v1.06. ccrypt is an encryption/decryption tool and bc is an arbitrary precision calculator. Both are written in C. Fatal bugs in each were first characterized in prior work by Liblit et al. [17]. Sections 5.2.1 and 5.2.2 illustrate how BTrace might be used in a typical debugging support role. Section 5.2.3 considers performance trends seen across both case studies.

5.2.1 ccrypt

CCRYPT has an input validation bug. In function `prompt()`, the `char *` pointer returned by `xreadline()` is dereferenced without being checked first. When `xreadline()` returns `NULL` to caller `prompt()`, failure is certain and immediate. CBI identifies two ranked lists of bug predictors, suggesting two distinct bugs. Each of the lists includes returning a `NULL` value from `xreadline()` as the eleventh strongest bug predictor, but have several other related behaviors in the same procedure as higher ranked predictors. Below we present our experience on using BTRACE on the first bug predictor list under two different scenarios. In each case we used a valid stack trace from CBI feedback reports.

In the first scenario, we turn off dataflow, i.e., we use the weight domain from Definition 6. The path returned by BTRACE, when given anywhere from the first 0 to 14 bug predictors, is the same: a `NULL` value is returned from `xreadline()` and then dereferenced `prompt()`. Adding higher ranked predictors does not change the path significantly. The interesting observation here is that even if we do not add any bug predictors, BTRACE still finds the same path. This suggests that the stack trace (actually just the crash site) is a fairly good indication of the bug and is enough for BTRACE to produce a path that illustrates the bug. This path returns `NULL` from `xreadline()` because it is the shortest path through that function.

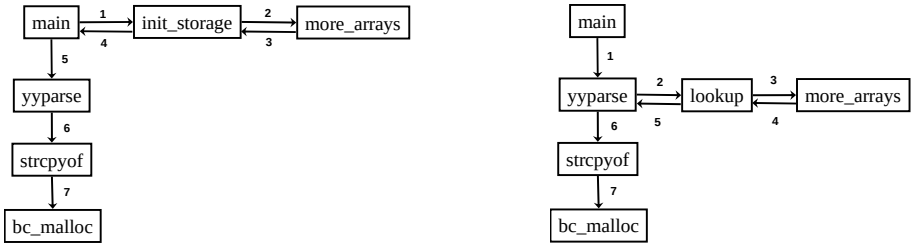
Even though this path indicates the bug, it is actually infeasible. The procedure `xreadline()` is also called from initialization routines in `main()` that check for the return value of `xreadline()` and gracefully terminate the program if the value is `NULL`. The path obtained from BTRACE returns `NULL` from each of these calls to `xreadline()`, and then incorrectly takes the wrong branch for the checks in `main()`. This suggests a need for dataflow analysis, but the user can use his own intuition to correct the path. As initialization routines rarely have a bug, the user can insert an ordering constraint that forces BTRACE to hit bug predictors in `xreadline()`, or any bug predictor for that matter, only after going through the initialization code. This corrects the infeasibility of the path.

In the second scenario, we turn on dataflow. However, we not obtain any change in the path till we include the eleventh predictor. The reason is that our dataflow analysis is not distributive. Consider the code for `xreadline()` shown in Figure 5. Our dataflow analysis summarizes the return value of `xreadline()` as $\perp$ or non-constant because `NULL` (line 46) and non-`NULL` (line 58) values get combined. When we insert the eleventh predictor, which is at line 45 (the first node of the `then` branch) the return value is summarized as $\perp$ when the eleventh predictor is not hit, and 0 or `NULL` when it is hit. Now the path correctly avoids returning `NULL` in `xreadline()` until the point when its return value is not checked i.e., when it is called from the procedure `prompt()`.[2]

[2]Some manual modifications are needed to the CCRYPT code to prevent the CBI instrumenting compiler from fooling our conditional-weight placement. When tracking branches, the instrumenting compiler rewrites even simple conditionals in a more complex form not modeled by our BTRACE implementation.

```
37: char *xreadline(FILE *fin, char *myname) {
38:   int buflen = INITSIZE;
39:
40:   char *buf = xalloc(buflen, myname);
41:   char *res, *nl;
42:
43:   res = fgets(buf, INITSIZE, fin);
44:   if (res==NULL) {
45:     free(buf);
46:     return NULL;
47:   }
48:   nl = strchr(buf, '\n');
      ...
58:   return buf;
59: }
```

Figure 5: Code for the procedure `xreadline()` used in CCRYPT



(a) With one predictor (b) With two predictors

Figure 6: Summaries of paths returned by BTRACE for BC. Boxes show the procedures that the path visits; edge labels show the order of calls and returns.

5.2.2 bc

BC has a buffer overrun in the function `more_arrays()`. The function allocates some memory to an array and then uses a wrong loop bound to walk through the array and `NULL` out its elements. The program eventually crashes in a subsequent call to `bc_malloc()`. The stack at the point of failure has unfinished calls to `main()`, `yyparse()`, `strcpyof()` and `bc_malloc()` in order. The presence of `bc_malloc()` suggests heap corruption but the stack provides no real clues as to when the corruption occurred or by what piece of code. Statistical debugging analysis by CBI identifies several suspect program behaviors. Predictors are scattered across several files and their relationship may not be clear on first examination.

Typical Debugging Session Suppose we wish to find the bug represented by one of CBI's two bug predictor lists. Some feedback reports include a valid stack trace, so we require that BTrace find paths ending in this stack configuration. We ask BTrace to hit the single top-ranked bug predictor, found in `more_arrays()`. Figure 6a summarizes the resulting path. This path shows a call from `main()` to `init_storage()` to `more_arrays()` that touches the desired bug predictor. Execution then continues through a set of additional calls (`yyparse()` to `strcpyof()` to `bc_malloc()`) that leave the stack in the desired final configuration.

However, this path is questionable. Cursory examination of the code, or data mining applied to CBI feedback reports, shows that `more_arrays()` is always called once at initialization time. If we believe that the initialization code is correct, we should disregard this call to `more_arrays()` and consider other ways of reaching that bug predictor. We therefore insert a zero-score bug predictor at the return-site of `init_storage()` along with the ordering constraint that the first bug predictor in `more_arrays()` must be visited after this new predictor in `init_storage()`. Under these new restrictions, BTrace gives the path shown in Figure 6b. BTrace bypasses `init_storage()` entirely, this time deeming it irrelevant to the bug. This revised path is more informative; indeed, while `more_arrays()` is always called once from `init_storage()`, it is the second or subsequent call from `lookup()` that spells trouble.

Additional Predictors If we show BTrace the top two predictors, it directly produces the path given in Figure 6b with no ordering constraints or other special intervention from the user. As before, the top predictor is in `more_arrays()`. The second predictor is in `lookup()`, which conditionally calls `more_arrays()`. BTrace hits both predictors in a single excursion from `yyparse()`. There is no benefit in hitting the same predictor twice, and therefore no reason to also include the call from `init_storage()` to `more_arrays()` as seen in Figure 6a. BTrace correctly bypasses that entire call, showing the more direct failure path given in Figure 6b while steering the programmer away from irrelevant code elsewhere.

The path remains unchanged as we increase the number of bug predictors from two to eight; these additional predictors are syntactically close to the first two, and are already touched by the path in Figure 6b. The ninth and later bug predictors have scores very near zero, suggesting they have low relevance and therefore are not useful to include in a reconstructed failure path.

Alternate Predictor Lists CBI actually produces two ranked lists of related bug predictors for bc, suggesting two distinct bugs. If we run BTrace using the single top predictor from the second list instead of the first, the reconstructed failure path is exactly the same. Up to twelve additional predictors leave the path unchanged, and even beyond that the path changes only slightly. The second predictor list is much more homogeneous, with `more_arrays()` code dominating the upper ranks, so most paths that hit one hit them all.

The tendency of BTrace to produce identical or similar paths from both lists suggests that they do correspond to a single bug, in spite of CBI's conclusions to the contrary. This is correct: the two lists do correspond to a single bug. CBI can be confused by sampling noise, statistical approximation, incompleteness of dynamic data, and other factors. BTrace may be a useful second check on CBI, letting us unify equivalent bug lists that CBI has incorrectly held apart. This finding also emphasizes the value of spreading predictors widely across an application: we see here that the greater heterogeneity of the CBI's first predictor list helps BTrace produce a correct path with fewer predictors than with the second list.

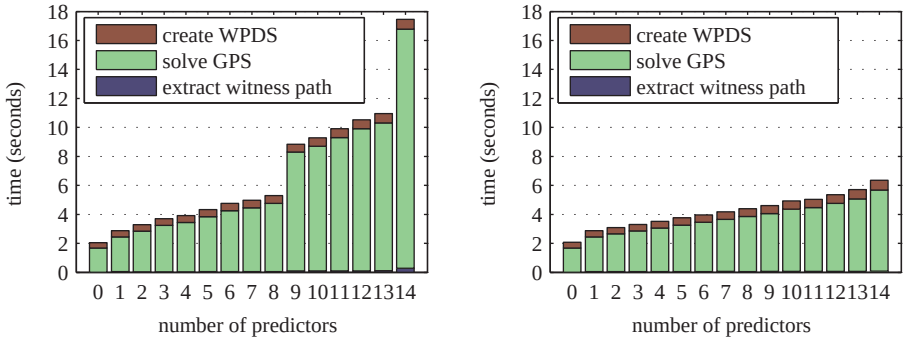
Unavailability of the Stack Trace The terminal state of the system, with outstanding calls from `main()` to `yyparse()` to `strcpyof()` to `bc_malloc()`, matches the stack trace reported by CBI and given to BTrace as path constraint. Critical bug predictors in `lookup()` and `more_arrays()` do not appear in the stack because those calls completed (and did their damage) long before the actual crash. The full BTrace path neatly integrates the terminal stack trace with the high-value bug predictors reported by CBI, and shows a short execution trace that is consistent with both of these key pieces of evidence.

However, a stack trace may not always be available. Buffer overruns and other memory corruption bugs can scramble the stack, leaving us with nothing more than the current program counter at the point of failure. If we give BTrace the failure location in `bc_malloc()` but no other information about the stack, the resultant paths change slightly from those we have already seen. The critical call sequence from `lookup()` to `more_arrays()` remains. However, after hitting the high-valued predictors in these two functions, BTrace simply takes the shortest path it can find into any call to `bc_malloc()`. This is an optimal path that is consistent with the available evidence, and the fact that the call from `lookup()` to `more_arrays()` is preserved means the path remains informative and useful.

5.2.3 Performance

Figure 7b, appearing on the right, shows analysis performance using bug predictors selected from bc's second predictor list. These and all following measurements were collected on a 1 GHz Athlon AMD processor with 1 GB RAM. Total time is split into the three key phases of the algorithm: creating the initial WPDS, solving the generalized pushdown successor (*GPS*) problem, and extracting a witness path from the solved system.

As expected, the *GPS* phase dominates. Using more predictors takes more time, but the increase is gradual. Recall that bc's second predictor list is fairly homogeneous and therefore yields paths which vary little as more predictors are used. The same path is produced for 0-12 predictors, then changes at 13 and again at 14. Thus, the performance slope in Figure 7b is primarily due to adding more bug predictors, with other factors held constant. Although Section 3.3 showed that solving the WPDS may require time exponential in the



(a) Using first predictor list

(b) Using second predictor list

Figure 7: BTrace performance on bc using varying numbers of predictors

number of bug predictors, we find that the actual slowdown is gradual and that the absolute performance of BTrace is good.

Figure 7a, appearing on the left, shows a similar performance profile. Gradual slowdown from 2-8 predictors represents the cost of adding more bug predictors with no change to the result path. The result path grows larger at nine predictors and again at fourteen, and analysis time increases accordingly. Thus, in practice the total length of the result path may be a more significant performance factor than the number of critical nodes visited along the way. Recall that the ninth and later predicates have CBI-assigned scores very near zero, which suggests that these longer paths are unlikely to even be requested by a BTrace user. For more realistic bug predictor counts of eight or below, the entire analysis completes in a few seconds.

bc has 45,234 CFG nodes, and a typical failure path produced by BTrace is about 3,000 nodes long. ccrypt is significantly smaller: 13,661 CFG nodes, with about 1,300 on a typical failure path. BTrace requires 0.10 seconds to find a path using zero ccrypt predictors, increasing gradually to 0.97 seconds with fifteen predictors. As with bc, more predictors slow the analysis gradually while longer failure paths have a larger effect.

Adding dataflow slows the analysis by a factor of between four and twelve, depending on the details of the configuration. Analysis with dataflow and realistic numbers of bug predictors takes about thirteen seconds for bc and less than two seconds for ccrypt.

6 Related Work

The Path Inspector tool [3, 7] makes similar use of weighted pushdown systems. It makes use of WPDSs for verification: to see if a program can drive an automaton, summarizing a program property, into a bad state. If this is possible,

it uses witnesses to produce the program path. It can also use dataflow analyses by encoding them as weights to rule out infeasible paths. We use WPDSs for optimization, which has not been previously explored.

Liblit and Aiken directly consider the problem of finding likely failure paths through a program [16]. They present a family of analysis techniques that exploit dynamic information such as failure sites, stack traces, and event logs to construct the set of possible paths that a program might have taken. They could not, however, optimize path length or the number of events touched when all of them might be unreachable in a single path. Our approach is, therefore, more general. BTrace incorporates these techniques, along with dataflow analysis, within the unifying framework of weighted pushdown systems. Another difference is that instead of using event logs, we use the output of CBI to guide the path-finding analysis. The theory presented in Section 3 can be extended to incorporate event logs by adding ordering constraints to appropriately restrict the order in which events must be visited by a path.

PSE is another practical tool for finding failing paths [19]. It requires a user-provided description of how the error could have occurred, e.g., "a pointer was assigned the value `NULL`, and then dereferenced." This description is in the form of a finite state automaton, and the problem of finding a failing run is reduced to finding a backward path that drives this automaton from its *error* state to its initial state. Their tool solves this in the presence of pointer-based data structures and aliasing. Our work does not require any user description of the bug that might have caused the crash, but we do not yet handle pointer-based structures.

Our work can also be compared with dynamic slicing [1, 12] that is concerned with identifying relevant program statements that affected the value of a variable at a certain point in the program's execution. Static information about the program can be used to reduce the runtime overhead of dynamic slicing [8, 10]. By using CBI, we only have to extract small amounts of dynamic information from each program run. We use static analysis to piece together this information and find a path in the program. Recent work by Zeller [27] uses a technique called Delta Debugging to find a cause-effect chain of program events that leads to failure. It exercises fine-grained control over a program's execution including going backwards in the execution to find the cause-effect chain.

In Definitions 6 and 7, we define semirings that are the power set of the values we want to associate with each path. This approach has been presented in a more general setting by Lengauer and Theune [14]. The power set operation is used to add distributivity to the semiring, and a reduction function, such as our *reduce*, ensures that we never form sets of more elements than necessary.

7 Conclusions and Future Work

We have presented a static analysis technique to build BTrace, a tool that can find an optimal path in a program under various constraints imposed by a user. Using bug predictors produced by CBI, BTrace can perform a postmortem

analysis of a program and reconstruct a program path that reveals the circumstances and causes of failure. The paths produced by BTrace might not be feasible, but we intend for them to help programmers understand the bug predictors produced by CBI and locate bugs more quickly. BTrace provides user options to supply additional constraints in the form of stack traces and ordering constraints, the latter of which allow the user to guide the tool interactively while locating a bug. In summary, more experiments are required to prove the utility of BTrace in debugging software, but initial results look promising.

For future work, we would like to evaluate the benefit of using better dataflow analysis in improving the quality of the path returned by BTrace. We would also like to interpret the dataflow information provided by the bug predictors themselves. If `x = 7` is a bug predictor, then we can use of this information to restrict the value of `x` when a path hits that predictor.

Another open area is to design incremental algorithms for weighted pushdown systems that do not recompute the solution each time the user makes a small change to the optimization problem. An example is adding bug predictors and ordering constraints on the fly. Currently, no such incremental algorithms exist, but their development could substantially improve the interactivity of an integrated BTrace bug hunting system.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[3] G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *Computer Aided Verfication*, 2005.

[4] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Symp. on Princ. of Prog. Lang.*, pages 62–73, 2003.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[6] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 57–68, 2002.

[7] GrammaTech, Inc. CodeSurfer Path Inspector, 2005. `http://www.grammatech.com/products/codesurfer/overview_pi.html`.

[8] R. Gupta, M. L. Soffa, and J. Howard. Hybrid slicing: Integrating dynamic information with static analysis. *ACM Transactions on Software Engineering and Methodology*, 6(4):370–397, Oct. 1997.

[9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proc. of the 16th Int. Conf. on Softw. Eng.*, pages 191–200. IEEE Computer Society Press, May 1994.

[10] K. Inoue, M. Jihira, A. Nishimatsu, and S. Kusumoto. Call-mark slicing: An efficient and economical way of reducing slices. In *Proc. of the 21st Int. Conf. on Softw. Eng.*, pages 422–431. ACM Press, May 1999.

[11] 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++`.

[12] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[13] A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *Computer Aided Verification*, pages 434–448, 2005.

[14] T. Lengauer and D. Theune. Unstructured path problems and the making of semirings (preliminary version). In *WADS*, volume 519 of *Lecture Notes in Computer Science*, pages 189–200. Springer, 1991.

[15] B. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[16] B. Liblit and A. Aiken. Building a better backtrace: Techniques for postmortem program analysis. Technical Report CSD-02-1203, University of California, Berkeley, Oct. 2002.

[17] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 2003.

[18] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 2005.

[19] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. In *Found. Softw. Eng.*, pages 63–72, 2004.

[20] M. Müller-Olm and O. Rüthing. On the complexity of constant propagation. In *European Symp. on Programming*, pages 190–205, 2001.

[21] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Symp. on Princ. of Prog. Lang.*, 2004.

[22] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *European Symp. on Programming*, 2005.

[23] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Sci. of Comp. Prog.*, volume 58, pages 206–263, 2005.

[24] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci.*, 167:131–170, 1996.

[25] F. Somenzi. Colorado University Decision Diagram package. Technical report, University of Colorado, Boulder, 1998.

[26] M. Wegman and F. Zadeck. Constant propagation with conditional branches. In *Symp. on Princ. of Prog. Lang.*, pages 291–299, 1985.

[27] A. Zeller. Isolating cause-effect chains from computer programs. In *Found. Softw. Eng.*, pages 1–10, 2002.

A Implementation Details

The worst-case running time of our analysis is exponential in the number of critical nodes. A part of this complexity is due to the expensive operations on weights as described in Definition 7. For this reason, we represent a weight from this semiring using an abstract decision diagram (ADD). An ADD is a variant of the well known binary decision diagrams [5] that are used to represent multi-valued functions on boolean variables. Let $n = |B|$ be the number of critical nodes, and $x_1, \cdots x_n$ be boolean variables. If $B = \{a_1, \cdots, a_n\}$, then let $\mathit{set}(x_1, \cdots, x_n) = \{a_i \mid x_i = 1\}$. A weight w can be represented by the following function.

$$f_w(x_1, \cdots, x_n) = \begin{cases} (v, d) & \text{if } (\mathit{set}(x_1, \cdots, x_n), v, d) \in w \\ (\infty, \overline{0}_d) & \text{otherwise} \end{cases}$$

Similar to BDDs, ADDs can also have canonical representations, i.e., each function has a unique representation. To take advantage of the compactness of such representations, the combine and extend operations on weights can be computed as follows. We write x_1 to denote the function $f(x_1, \cdots, x_n) = x_1$, and $\overline{x}_1$ for the function $f(x_1, \cdots, x_n) = \overline{x}_1$. Then

$$\begin{aligned} w_1 \oplus w_2 &= \overline{x}_1(w_1(x_1 = 0) \oplus w_2(x_1 = 0)) \oplus \\ &\quad\; x_1(w_1(x_1 = 1) \oplus w_2(x_1 = 1)) \\ w_1 \otimes w_2 &= (\overline{x}_1\ w_1(x_1 = 0) \oplus x_1\ w_1(x_1 = 1)) \otimes \\ &\quad\; (\overline{x}_1\ w_2(x_1 = 0) \oplus x_1\ w_2(x_1 = 1)) \\ &= \overline{x}_1(w_1(x_1 = 0) \otimes w_2(x_1 = 0)) \oplus \\ &\quad\; x_1((w_1(x_1 = 0) \otimes w_2(x_1 = 1)) \oplus \\ &\qquad (w_1(x_1 = 1) \otimes w_2(x_1 = 0)) \oplus \\ &\qquad (w_1(x_1 = 1) \otimes w_2(x_1 = 1))) \end{aligned}$$

This gives a recursive way of computing the combine and extend of two weights in time proportional to the product of the sizes of their ADD representations. The base case of this recursion is applying the operations on two constant-valued functions:

$$\begin{aligned}(v_1, d_1) \oplus (v_2, d_2) &= (min\{v_1, v_2\}, d_1 \oplus_d d_2) \\ (v_1, d_1) \otimes (v_2, d_2) &= (v_1 + v_2, d_1 \otimes_d d_2)\end{aligned}$$

In the presence of ordering constraints, the extend operation becomes a little more complicated. Suppose we have a list of constraints c, with each constraint being a pair (i, j) representing that node a_i must be visited before node a_j. Let $(i, j)\ c$ denote the list whose first constraint is (i, j). The extend of two weights under a list c of constraints is written as $w_1 \otimes_c w_2$. It can be recursively computed as follows:

$$\begin{aligned}w_1 \otimes_{(i,j)\ c} w_2 &= (w_1 \otimes_c w_2(x_i = 0)) \oplus \\ &\quad (w_1(x_j = 0) \otimes_c w_2)\end{aligned}$$