

Comparison-based Computation of Causal Paths for Software Failures

William N. Sumner Xiangyu Zhang Suresh Jagannathan
Department of Computer Science, Purdue University
West Lafayette, Indiana 47907
{wsumner,xyzhang,suresh}@cs.purdue.edu

ABSTRACT

In this paper, we present an automated debugging technique that explains a failure by computing its causal path leading from the root cause to the failure. The technique assumes two executions, one passing and the other failing. The two executions could be directly available if the failure is caused by a regression fault. Otherwise, the passing run is automatically generated by searching for a dynamic patch. Fine-grained program state comparison between the two runs is performed to isolate the causal path. We identify that a key step is to align two executions before comparison. We propose to use a technique called execution indexing, whose goal is to establish a mapping between equivalent points in two different executions based on execution structure so that comparison can be meaningfully performed. We formally define the technique, identify and prove properties that are crucial to efficiency and efficacy. Experiments show that the proposed approach is able to compute high quality causal paths that explain failures, with reasonable cost.

1. INTRODUCTION

Debugging identifies faults in software by diagnosing runtime failures. It is an indispensable process for software dependability and productivity. It would be ideal to be able to prevent mistakes in the first place or identify faults statically without letting them manifest as runtime failures. However, due to the limitations of the enabling techniques (e.g., formal verification, model checking, and static program analysis), and the fact that error-prone humans play an important role in employing these techniques, it is unlikely runtime failures can be fully prevented in the near future. Therefore, debugging remains a key challenge.

Over the years, software has evolved from small sequential programs to concurrent or distributed programs with millions of lines of code. The common practice of debugging is still that upon observing a failure, the programmer sets breakpoints, restarts the execution, changes values at breakpoints, and inspects the result of the perturbation. Such a

practice is tedious and sometimes becomes unmanageable for large software.

Automated debugging has been increasingly studied lately [22, 23, 3, 18, 7]. Existing techniques have the following limitations. *First*, the outcome of most automated techniques is a ranked list of statements that are fault candidates. The onus is on the programmer to understand the failure and correlate these fault candidates to the failure in order to decide which one is the real fault. *Second*, these techniques handle only coding errors, i.e. assume faults reside in code. It is reported in [17] that, in reality, many failures cannot be attributed to coding errors. One of the largest classes of problems arises from errors made in the eliciting, recording, and analysis of requirements. The resulting failures cannot be simply blamed on specific program statements. Understanding the causality of these failures is highly desirable. *Third*, many techniques assume a large test suite, which may not be easily satisfiable.

Recently, Zeller et al. proposed a technique to isolate state transitions that are critical for a failure [37, 7]. The idea is to compare the failing execution with *a similar but correct execution generated from a different input*. The sequence of state difference transitions can serve the goal of failure explanation to some extent. However, using an execution from a different input for comparison introduces misleading noise into the results due to the underlying semantic differences between the two executions. Moreover, the points in the two executions where comparisons are conducted have to be selected manually or by ad-hoc methods such as using timestamps. The incurred inaccuracy often leads to state transitions that are irrelevant to the failure being isolated.

In this paper, we propose a technique to compute failure explanations automatically. Given 1) a defective program, 2) input that causes the defect to be observed, and 3) a non-regression version of the faulty program or the desired output if such a version is not available, the system automatically derives a failure explanation. The explanation is a minimal sequence of *causally related*, executed statements that *produce faulty values* in the failing run, starting from the root cause and leading to the observable defective behavior. Informally, two executed statements are causally related with respect to sets of variables if the values of one set at the earlier statement decide the values of another set at the later statement. These explanations capture both the exact source code relevant for fixing the defect and the behavior that encompasses how the faulty code interacts with the program to produce a defective result. Hence, they can substantially facilitate the process of understanding and then

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

correcting defects.

Motivating Example. To better illustrate the capability of the proposed technique, one of the case studies for real world applications is presented as follows.

Version 2.5.1 of the `grep` utility has a flaw which causes `grep` to fail to find strings that match the user provided pattern if both `-F` and `-w` options are specified [39]. These options require that the pattern be a string and a word respectively, but those cases are not mutually exclusive and should not preclude matching. This is a design bug.

Code Snippet

```
Fexecute(buf, size, match_size, exact):
1  if (match_words) {
2      /*verify the match is a word*/
3      return -1;
grepbuf (char const *beg, char const *lim):
4  match_offset = Fexecute(p, lim-p, &match_size, 0);
5  while (match_offset != -1) {
6      /*Print results & Keep scanning*/
main(argc, argv):
7  while (opt != -1) { ...
8      case 'F': setmatcher ("fgrep");
9      case 'w': match_words = 1;
10 n += grepbuf (beg, lim);*
```

Causal Path

At 7, `opt` is given 1
 At 9, `match_words` is given 1
 At 1, (`match_words`) is true
 At 3, `Fexecute` returns -1
 At 4, `match_offset` is given -1
 At 5, (`match_offset != -1`) is false
 Thus the output ... differs from ...

Figure 1: Causal Path for `grep 2.5.1`

The resulting causal path and relevant code is reproduced in Fig. 1. First, the causal path reveals that the `-w` option is required for the error and that it requires word patterns by setting the `match_words` flag. The third step says this flag is checked in `Fexecute`, whose execution is dictated by the option `-F`. Due to the erroneous design, `Fexecute` immediately and prematurely returns -1 to `match_offset`, signaling that the buffer is done scanning. Finally, at statement 5, `match_offset` is checked and the program ends without printing any matches.

For this case, there is no single statement to blame for the failure. Understanding the causal path of the failure becomes much more important. In `grep 2.5.3`, the bug is fixed by placing a multi-line patch at statement 2. Observe that this is where the explanation deviates from the programmer’s intent and returns prematurely. The fact that the causal path provides an explanation and points the programmer to the faulty location clearly demonstrates the power of proposed technique. Note that the explanation contains only 6 steps while the program has over 9000 lines of code.

The overarching design of the system involves first generating a reference execution modeling the correct behavior, either from a non-regressing version or by dynamically patching the failing execution, driven by the given expected output. It then aligns the failing and reference runs. This is necessary because the two executions may differ in control flow as well as in program state, so comparison may be carried out at incomparable places if executions are not carefully aligned. Program states at aligned points are compared and state differences are minimized. The chain of minimal state differences that are critical to the manifestation of the

failure is reported as the causal path of the failure.

In summary, this paper makes the following major contributions.

- We propose a technique that can automatically explain software failures by computing their causal paths. We develop a formal model in which the causal path computation is defined.
- We present two ideas that are key to efficacy. One is to use a technique called execution indexing (EI) to align executions before they are compared. The other is to use a reference execution originating from the same input instead of a different input for comparison.
- We identify and prove that the causal path computation must terminate at the root cause if the root cause is a single line fault.
- We propose fast pass optimizations that can avoid redundant computation. We also prove these optimizations are safe, meaning that they do not alter the final outcome.
- We implement a prototype and evaluate it on a few hundred synthetic bugs in the Siemens test suite [16] and a set of real world bugs from medium-sized open source programs. The evaluation demonstrates the effectiveness and efficiency aspects of the technique. In 80% of the synthetic cases that are not caused by code omissions (the fault is that a piece of code missing) and 83% of the real cases, it precisely captures the perfect failure explanations. These causal paths start exactly at the root cause, end at the point of failure. Most of them have a length less than 10 steps, requiring very little effort for a human to understand the failures.

2. OVERVIEW

The central idea of the technique is to compare two executions generated from the same input, with one passing and the other failing. The comparison is done by contrasting the states at corresponding points in the two respective runs, starting from the point of failure and proceeding backwards. In particular, the faulty variables at a step are derived by comparing their values in the failing run with those in the correct run. Not all faulty variables are relevant to the failure. Our technique minimizes the set of relevant faulty variables at each step. The process terminates when there are no relevant variables, often implying the faulty statement has not yet polluted program state at that point in the failing execution. The causal path is essentially the chain of definitions that produce the relevant faulty variables.

Consider the example in Fig. 2. The code on the left computes the sum of values stored in a linked list. There is a regression fault at line 5, which alters the result of the predicate variable `x`. The predicate controls whether or not the current element should be skipped. The table presents the two executions. Columns `Execution` present the statement traces and columns $\sigma(\dots)$ present the program state *before each statement execution*. A state comprises the values of the listed variables. Pointer values are represented by linked list states on the bottom. For example, at point \textcircled{A} of the failing run, the state “ $S_1, 1, -$ ” means that before executing the statement at \textcircled{A} , the pointer `p` is pointing to the head of the list, `sum` has the value of 1 and `x` has not been defined.

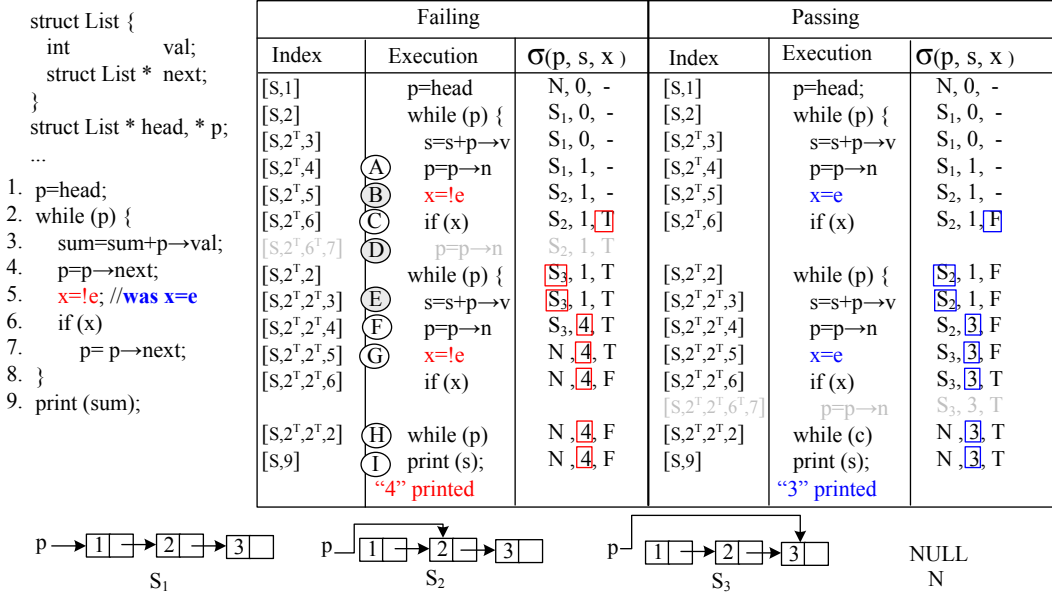


Figure 2: Example for causal path computation. Code is on the left; column Index represents an abstraction for executions points that allows execution alignment; Execution presents the trace; σ shows program state; symbol s is the abbreviation for variable `sum`; value differences that are relevant to the failure are boxed; pointer values S_1 , S_2 , S_3 , and N are explained on the bottom.

The two executions are first aligned, using the information in the **Index** column. Explanation of execution indexing will be presented in Section 3. In the table, the two executions have already been aligned, i.e., corresponding points are in the same row. The computation is conducted by stepping backwards at the aligned points. As the first step of the causal path computation, it is observed that the two outputs are different at the end of the two executions. Going backwards, at the aligned point **Ⓘ**, it is computed that variable s is the relevant faulty variable. This is decided by testing if replacing the variables' values in the passing run and continuing the execution leads to the relevant faulty variables in the next aligned point having the faulty values. Here, replacing 3 with 4 (the value in the failing run) at **Ⓘ** (before executing the `print` statement) in the passing run manifests the failure. Note that although variable x also represents a value difference, it is not relevant to the failure at this point. Stepping backwards, at **Ⓔ**, the same relevant variable s is computed. When the computation proceeds backward to **Ⓔ**, p becomes the relevant variable because replacing its value S_2 in the passing run with S_3 is sufficient to produce $s=4$ at **Ⓔ**. Similarly, at **Ⓒ**, the relevant variable becomes x . At **Ⓒ**, there is no relevant variable as all variables have the same values across the two runs; thus, the computation terminates. The definition points of all the computed relevant variables constitute the causal path. Here, the path comprises the shaded points **Ⓔ** → **Ⓓ** → **Ⓔ** → failure. The textual explanation is that x is evaluated to the wrong branch outcome, leading to p advancing one extra step, which further results in the wrong value being aggregated to `sum` and hence the failure. One can see the path clearly explains the causality of the failure.

Note that points **Ⓔ** and **Ⓒ** produce different values across two runs and hence will be reported by a trace differencing based technique [14]. The traditional dynamic slicing technique [20] will identify all the statements that have depen-

dence with the failure and hence include points such as **Ⓐ** (because the failure is transitively dependent on it through the edge from **Ⓓ** to **Ⓐ**). In contrast, our causal path is very precise and often captures the root cause as the head of the path; individual steps in the path contain only faulty values and are causally related; and the path ends at the failure.

3. EXECUTION INDEXING

The success of the technique hinges on correctly identifying corresponding points in the two executions. Comparisons are only performed at such *aligned points*. If points for comparison are not carefully chosen, the resulting state difference may very likely be due to the intrinsic semantic differences between the two points instead of the fault. As a consequence, the resulting causal path may fail to explain the failure. A simple strategy is to align two execution points (across executions) that have the same statement, calling context, and instance count. However, such an idea falls short in practice. Following this strategy, in Fig. 2, the first instances of statement 7 (the grey lines in columns **Execution**) in the two runs align. Comparing their states identifies that the values of x concur, which implies x has the correct value. However, x has a faulty value in the failing run. This is masked by the fact that the comparison is performed between undesirable points.

In this paper, we use *execution indexing* (EI) developed in our prior work [36] to facilitate alignment. Informally, an *index tree* is constructed for an execution, representing the nesting structure of execution points. In particular, a leaf node is created for each execution point. Internal nodes are created to represent control structures such as branches of conditionals, loop bodies, and method invocations. The index trees for the two executions in Fig. 2 are shown in Fig. 3. Consider the left tree. The root node represents the whole execution, which consists of three statement executions pre-

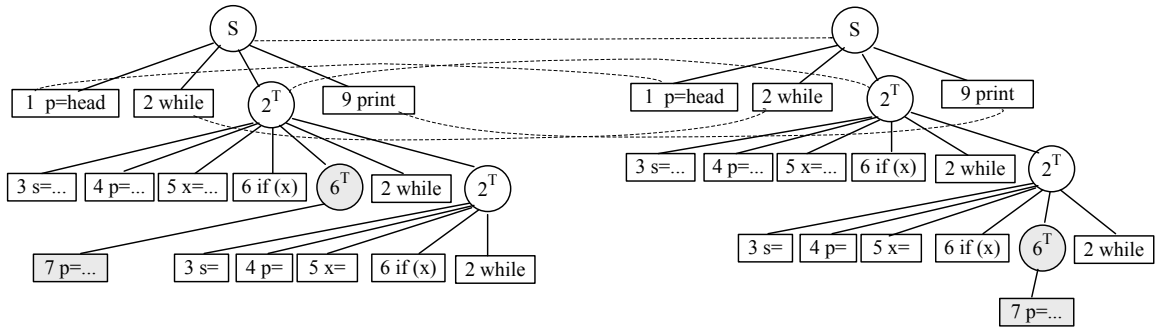


Figure 3: Structural indexing trees for the executions in Fig. 2 with boxes representing leaf nodes and circles representing internal nodes. Dotted lines show part of the alignment.

sented as the leaf children of the root. Since the `while` statement has substructure, an internal node 2^T , pronounced as “the true branch of statement 2”, is created to represent the loop body. The remainder is constructed similarly. Execution alignment is achieved by aligning the index trees. The intuition is that at the highest level, the two roots should align because the two executions as a whole should align. The terminal children of the roots, namely, the three statements at the first level, should align as represented by the dotted lines because their executions are dictated by their parent. The internal nodes may or may not align, depending on their branch outcomes. If they do align, the recursive alignment of their children should be further performed. It is easy to tell that all the nodes in the left tree have corresponding instances in the right tree except for the shaded 6^T and 7. That clearly indicates the branch was taken in the failing run but not in the passing run.

The index of an execution point is defined as the path leading from the root of the index tree to the terminal node representing the point. For instance, the index of the shaded 7 in the corner of the left tree is $[S, 2^T, 6^T, 7]$, implying the statement execution is inside the first iteration of the loop and the true branch of the `if` statement at 6. It has been proved in [36] that an index uniquely represents an execution point. Optimizations and handling non-structural control flow caused by `break/continue/longjmp/setjmp` are also discussed in [36]. EI has been used in [36, 19] to identify corresponding dynamic points across concurrent executions.

In this paper, we compute indices on the fly and emit them to a trace. This index trace is used to align the two executions and drive causal path computation.

4. FORMAL MODEL

4.1 The Language

KERNEL-LANGUAGE \mathcal{L}

```

 $e \in \text{Expression} ::= x \mid v \mid e + e \mid \dots$ 
 $s \in \text{Statement} ::= x =^{\ell} e \mid \text{while } x^{\ell} \{s\} \mid$ 
 $\quad \text{if } x^{\ell} \text{ then } s_1 \text{ else } s_2; \text{ endif} \mid$ 
 $\quad s_1; s_2 \mid \text{Skip}$ 
 $v \in \text{Constant} ::= \text{true} \mid \text{false} \mid 0 \mid 1 \mid \dots$ 

```

Figure 4: Syntax

We first introduce a simple imperative language. The syn-

tax is shown in Fig. 4. We will discuss extensions to handle methods, pointers, etc. in Section 5. In the language, statements are uniquely labeled to denote static program points. Only variables are allowed as predicates of conditional and loop statements. If an expression is intended to be used as an assignment to a variable followed by a predicate on the variable. We also use a special statement `endif` to explicitly denote the end of the conditional statement. Note that the grammar dictates that a `endif` uniquely matches with a `if` statement. `Skip` represents an empty statement. A value could be a boolean or an integer.

The operational semantics for the core language is presented in Fig. 5. Relevant definitions are given on the top. Symbol σ is used to represent the store, which is a mapping from variables to constants. *Index* is a sequence of labels, which may be annotated with boolean values, to denote the index of the current evaluation point.

4.1.1 Evaluation Rules

The evaluation rules along with instrumentation semantics to compute indices are presented in Fig. 5. Expression evaluation is of the form $\langle e, \sigma \rangle \rightarrow v$, meaning an expression e is evaluated to a value v with the store σ .

The configuration of statement evaluation is of the form $\langle s, \sigma, \mathcal{I} \rangle$, in which s represents a statement and \mathcal{I} represents the nesting structure of the current evaluation. The current index is the concatenation of \mathcal{I} and the label of the statement under evaluation. Rule *Assignment* evaluates the right hand side expression and updates the store accordingly. In conditional statements (*If-True* and *If-False*) evaluation, the branch is taken according to the value of the predicate variable, and the index is updated by pushing the label of the conditional statement (annotated with the branch outcome to distinguish the two branches), indicating the subsequent statement evaluations are nested in the branch of the conditional. At the end of the taken branch, Rule *Endif* is applied to evaluate the `endif` statement, which pops the most recently pushed label from \mathcal{I} , indicating the following statement evaluations are no longer nested in the branch. The index manipulations are analogous to call stack manipulations, both following the LIFO rule. The difference is that index computation requires pushing at predicates and popping in the end of branches. Rule *While* expands a loop statement to a conditional statement, whose evaluation will update \mathcal{I} . Note that in the expanded form, the `if` statement

DEFINITIONS	
$\sigma \in Store ::= Variable \mapsto Constant$	
$\mathcal{I} \in Index ::= \overline{\ell^{T/F/\epsilon}}$	
RIGHT HAND SIDE RULES	
$\frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)} (Variable)$	$\frac{}{\langle v, \sigma \rangle \rightarrow v} (Constant)$
$\frac{\langle e_1, \sigma \rangle \rightarrow v_1 \quad \langle e_2, \sigma \rangle \rightarrow v_2}{\langle e_1 + e_2, \sigma \rangle \rightarrow v_1 + v_2} (Addition)$	
STATEMENT RULES	
$\frac{\langle e, \sigma \rangle \rightarrow v}{\langle x =^\ell e, \sigma, \mathcal{I} \rangle \rightarrow \langle \text{Skip}, \{x \mapsto v\} \sigma, \mathcal{I} \rangle} (Assignment)$	
$\frac{\mathcal{I} = \mathcal{I}' \cdot \ell^b}{\langle \text{endif}, \sigma, \mathcal{I} \rangle \rightarrow \langle \text{Skip}, \sigma, \mathcal{I}' \rangle} (Endif)$	
$\frac{\langle x, \sigma \rangle \rightarrow \text{true}}{\langle \text{if } x^\ell \text{ then } s_1 \text{ else } s_2; \text{ endif}, \sigma, \mathcal{I} \rangle \rightarrow \langle s_1; \text{ endif}, \sigma, \mathcal{I} \cdot \ell^T \rangle} (If - True)$	
$\frac{\langle x, \sigma \rangle \rightarrow \text{false}}{\langle \text{if } x^\ell \text{ then } s_1 \text{ else } s_2; \text{ endif}, \sigma, \mathcal{I} \rangle \rightarrow \langle s_2; \text{ endif}, \sigma, \mathcal{I} \cdot \ell^F \rangle} (If - False)$	
$\frac{}{\langle \text{while } x^\ell \{s\}, \sigma, \mathcal{I} \rangle \rightarrow \langle \text{if } x^\ell \text{ then } s; \text{ while } x^\ell \{s\} \text{ else Skip}; \text{ endif}, \sigma, \mathcal{I} \rangle} (While)$	
$\frac{\langle s_1, \sigma, \mathcal{I} \rangle \rightarrow \langle r, \sigma', \mathcal{I}' \rangle}{\langle s_1; s_2, \sigma, \mathcal{I} \rangle \rightarrow \langle r; s_2, \sigma', \mathcal{I}' \rangle} (Sequence)$	
$\frac{}{\langle \text{Skip}; s, \sigma, \mathcal{I} \rangle \rightarrow \langle s, \sigma, \mathcal{I} \rangle} (Skip)$	
Figure 5: Evaluation Rules.	

and the **while** statement have the same label, ensuring that iterations can be correctly represented by having the same label repeated on \mathcal{I} . Rule *Sequence* handles the evaluation of a statement sequence. Rule *Skip* represents a no-op and reduces the statement sequence. The evaluation terminates if the program evaluates to a **Skip** statement.

Example. The following example demonstrates how \mathcal{I} is updated in loop evaluation. For brevity, the stores are not presented and an expression is used as the loop predicate. Variable $i = 0$ initially.

$$\begin{aligned}
&\langle \text{while } i < 2^1 \{i =^2 i + 1\}, [S] \rangle \rightarrow \\
&\langle \text{if } i < 2^1 \text{ then } i =^2 i + 1; \text{while } \dots; \text{endif}, [S] \rangle \rightarrow \\
&\langle i =^2 i + 1; \text{while } \dots; \text{endif}, [S, 2^1] \rangle \rightarrow \\
&\langle \text{while } i < 2^1 \{i =^2 i + 1\}; \text{endif}, [S, 2^1] \rangle \rightarrow \\
&\langle \text{if } i < 2^1 \text{ then } \dots \text{while } \dots; \text{endif}; \text{endif}, [S, 2^1] \rangle \rightarrow \\
&\langle i =^2 i + 1; \text{while } \dots; \text{endif}; \text{endif}, [S, 2^1, 2^1] \rangle \rightarrow \\
&\dots \\
&\langle \text{endif}; \text{endif}, [S, 2^1, 2^1] \rangle \rightarrow \\
&\langle \text{endif}, [S, 2^1] \rangle \rightarrow \dots
\end{aligned}$$

The **while** statement is expanded to a **if** statement. The **if** statement evaluation at each iteration pushes an anno-

tated label 2 to \mathcal{I} , which is popped when the corresponding **endif** statement is encountered later. A more complicated example on index computation can be found in Fig. 2.

DEFINITIONS	
$\mathcal{T} \in Trace ::= \overline{\text{entry}(\mathcal{I}, \sigma)}$	
TRACING RULES	
$\frac{\langle s^\ell, \sigma, \mathcal{I} \rangle \rightarrow \langle r, \sigma', \mathcal{I}' \rangle}{\langle s^\ell, \sigma, \mathcal{I}, \mathcal{T} \rangle \xrightarrow{t} \langle r, \sigma', \mathcal{I}', \mathcal{T} \cdot \text{entry}(\mathcal{I} \cdot \ell, \sigma) \rangle} \begin{array}{l} (T - Assign) \\ (T - IfTrue) \\ (T - IfFalse) \end{array}$	
$\frac{\langle s, \sigma, \mathcal{I} \rangle \rightarrow \langle r, \sigma', \mathcal{I}' \rangle}{\langle s, \sigma, \mathcal{I}, \mathcal{T} \rangle \xrightarrow{t} \langle r, \sigma', \mathcal{I}', \mathcal{T} \rangle} \begin{array}{l} (T - Endif) \\ (T - While) \\ (T - Skip) \\ (T - Sequence) \end{array}$	
Figure 6: Tracing Rules.	

In order to drive causal path computation, we introduce the tracing semantics in Fig. 6. These evaluation rules yield a sequence of index and store pairs. If the statement being evaluated is an assignment or a conditional, the rules are of the format on the top. More particularly, a trace entry is emitted, consisting of the index of the evaluation and the store right before the evaluation. The index is the concatenation of \mathcal{I} and the label of the statement. For statements whose evaluations do not have any effect on data or control flow, namely, do not change the store or the index, the tracing rules have the format on the bottom. Loop statements and skip statements fall into this category. Note that in our implementation, we do not collect stores in the trace. Instead, we only collect the index trace and since we consider only deterministic executions, the stores can be reproduced through re-execution.

4.2 Causal Path Computation

In our formal model, we consider two program versions: $P^{\mathcal{P}}$ represents the passing version and $P^{\mathcal{F}}$ represents the failing version. From now on, we use superscripts \mathcal{F}/\mathcal{P} to distinguish terms or operations in the failing evaluation and the passing evaluation, respectively. Let Δ be a sequence of code edits, including statement changes, additions, and eliminations.

$$P^{\mathcal{F}} = \overline{\Delta} P^{\mathcal{P}}$$

We further assume a unique label is associated with each distinct statement across versions. In other words, the statements that have not been changed must have the same label. This ensures our execution indices are able to correlate execution points across runs.

Next, we present evaluation rules for causal path computation. The evaluation configuration has the form of $\langle \Lambda, \Gamma, \mathcal{CP} \rangle$. The definitions of the terms are presented in Fig. 7. In particular, Λ is a sequence of alignment entries, representing the join ($\bowtie_{\mathcal{I}}$) of the passing trace and the failing trace on the index field. The join operator is defined in Fig. 7. Intuitively, it realizes trace alignment by finding entries that have identical indices in both traces and composing them into alignment entries. An alignment entry, denoted as **align**, consists of an index that is present in both traces and the stores for the index in the two traces. Term Γ represents the set of

DEFINITIONS

$$\Lambda \in \text{Alignment} ::= \overline{\text{align}(\mathcal{I}, \sigma^{\mathcal{P}}, \sigma^{\mathcal{F}})} \quad \Gamma \in \text{RelevantVar} ::= \{x\} \quad \mathcal{CP} \in \text{CausalPath} ::= \{\mathcal{I}\}$$

$\mathcal{T}^{\mathcal{P}}$ and $\mathcal{T}^{\mathcal{F}}$ are the generated traces when the passing and failing runs terminate;

$\sigma_0^{\mathcal{P}}$ and $\sigma_0^{\mathcal{F}}$ are the stores when the two runs terminate;

align_0 is a shorthand for $\text{align}(\mathcal{I}_0 \cdot \ell_0, \sigma_0^{\mathcal{P}}, \sigma_0^{\mathcal{F}})$; align_1 is a shorthand for $\text{align}(\mathcal{I}_1 \cdot \ell_1, \sigma_1^{\mathcal{P}}, \sigma_1^{\mathcal{F}})$;

$\text{outputVar}()$ identifies the set of output variables;

$\text{defpoint}(x, \mathcal{I})$ identifies the latest definition to x that precedes the index of \mathcal{I} .

$$\text{entry}(\mathcal{I}_1, \sigma_1) \cdot \mathcal{T}_1 \bowtie_{\mathcal{I}} \text{entry}(\mathcal{I}_2, \sigma_2) \cdot \mathcal{T}_2 = \begin{cases} \text{align}(\mathcal{I}_1, \sigma_1, \sigma_2) \cdot \mathcal{T}_1 \bowtie_{\mathcal{I}} \mathcal{T}_2 & \mathcal{I}_1 = \mathcal{I}_2; \\ \mathcal{T}_1 \bowtie_{\mathcal{I}} \mathcal{T}_2 & \text{Otherwise} \end{cases}$$

CAUSAL PATH COMPUTATION RULES

$$\frac{\Gamma_0 \subseteq \{x \mid \sigma_0^{\mathcal{P}}(x) \neq \sigma_0^{\mathcal{F}}(x)\} \quad \langle s^{\ell_0}; S_0, \{\Gamma_0 \mapsto \sigma_0^{\mathcal{F}}(\Gamma_0)\} \sigma_0^{\mathcal{P}}, \mathcal{I}_0 \rangle \rightarrow \dots \rightarrow^{\mathcal{P}} \langle s^{\ell_1}; S_1, \sigma', \mathcal{I}_1 \rangle \quad \sigma'(\Gamma_1) = \sigma_1^{\mathcal{F}}(\Gamma_1)}{\langle \Gamma_0, \text{align}(\mathcal{I}_0 \cdot \ell_0, \sigma_0^{\mathcal{P}}, \sigma_0^{\mathcal{F}}) \rangle \rightsquigarrow \langle \Gamma_1, \text{align}(\mathcal{I}_1 \cdot \ell_1, \sigma_1^{\mathcal{P}}, \sigma_1^{\mathcal{F}}) \rangle} \text{ (Induction)}$$

$$\frac{\Lambda = \mathcal{T}^{\mathcal{P}} \bowtie_{\mathcal{I}} \mathcal{T}^{\mathcal{F}} \quad \Gamma = \{x \mid \sigma_0^{\mathcal{P}}(x) \neq \sigma_0^{\mathcal{F}}(x)\} \cap \text{outputVar}()}{\langle \Lambda, \Gamma, \phi \rangle} \text{ (Initialization)}$$

$$\frac{\langle \Gamma_0, \text{align}_0 \rangle \rightsquigarrow \langle \Gamma_1, \text{align}_1 \rangle \quad |\Gamma_0| > 0 \quad |\Gamma_0| \text{ is minimal}}{\langle \Lambda \cdot \text{align}_0 \cdot \text{align}_1, \Gamma_1, \mathcal{CP} \rangle \xrightarrow{\text{cp}} \langle \Lambda \cdot \text{align}_0, \Gamma_0, \mathcal{CP} \cup \text{defpoint}(\Gamma_0, \mathcal{I}_0) \rangle} \text{ (Single - Step)}$$

Figure 7: Causal Path Computation.

relevant variables at an alignment entry. Informally, relevant variables contain different values in the two runs and they transitively cause the failure. A variable in a relevant set is also called a *value difference*. Term \mathcal{CP} represents the causal path, it is essentially a set of causally related definition points denoted by their indices. It provides reasoning about the failure generation.

Rule *Induction* defines a helper relation. A set of variables Γ_0 at an earlier alignment entry align_0 induces a given relevant set Γ_1 at a later alignment entry align_1 , if all variables in Γ_0 have different values in the earlier stores, and replacing their values in the passing evaluation with their values in the failing evaluation leads to the later evaluation at $\mathcal{I}_1 \cdot \ell_1$, and the variables in Γ_1 have the same faulty values. Note that symbol $\rightarrow^{\mathcal{P}}$ represents program evaluation with the passing program version. The causal path evaluation is driven by Λ in a reverse order. Intuitively, the process evaluates each entry in Λ from the end to the beginning. Relevant variables are computed at each step, definition points of relevant variables are inserted into \mathcal{CP} to construct the causal path. In Fig. 7, Rule *Initialization* initiates the evaluation. The alignment is the join of the passing trace and the failing trace. The relevant set is initialized to the set of output variables that have different values at the end of the two evaluations. The causal path is initialized to be empty. Rule *Single-Step* explains how a single step causal path evaluation is conducted. It concerns two consecutive alignment entries identified by $\mathcal{I}_0 \cdot \ell_0$ and $\mathcal{I}_1 \cdot \ell_1$, i.e. align_0 and align_1 , with $\mathcal{I}_0 \cdot \ell_0$ being the predecessor in the program evaluation order. Symbols ℓ_0 and ℓ_1 represent the labels of the statements in the two entries. We use s^ℓ to denote statement s with a label ℓ . Given the relevant variables Γ_1 at align_1 , the rule generates the relevant variables at the preceding entry align_0 . The evaluation can progress as long as there is a minimal non-empty relevant set at the preceding entry that induces Γ_1 at the later entry. Minimality is simply set cardinality minimality. The causal path is updated with the definition

points of the variables in Γ_0 in the failing run, i.e. the points generate the faulty values.

Example. Consider our running example. The application of the *Initialization* rule sets Λ to

$$\begin{aligned} & \text{align}([\text{S}, 1], \{p \mapsto N, s \mapsto 0, \dots\}, \{p \mapsto N, s \mapsto 0, \dots\}) \cdot \\ & \text{align}([\text{S}, 1], \{p \mapsto S_1, s \mapsto 0, \dots\}, \{p \mapsto S_1, s \mapsto 0, \dots\}) \cdot \\ & \dots \\ & \text{align}([\text{S}, 9], \{p \mapsto N, s \mapsto 4, x \mapsto F\}, \{p \mapsto N, s \mapsto 3, x \mapsto T\}) \end{aligned}$$

The shaded trace entries such as that at \textcircled{D} are not part of Λ .

The relevant set Γ is set to $\{s\}$ as s is an output variable that has different values. For brevity, we use shorthands, e.g., align_I is the shorthand for the alignment entry at \textcircled{I} . Since

$$\begin{aligned} \{s\} & \subseteq \{y \mid \sigma_H^{\mathcal{P}}(y) \neq \sigma_H^{\mathcal{F}}(y)\} \\ \langle \text{while}^H; \text{print}^I, \{s \mapsto 4\} \sigma_H^{\mathcal{P}}, [\text{S}] \rangle & \rightarrow \\ \langle \text{print}^I, \sigma_I = \{p \mapsto N, s \mapsto 4, x \mapsto T\}, [\text{S}] \rangle & \\ \sigma_I(s) = \sigma_I^{\mathcal{F}}(s), & \end{aligned}$$

Rule *Induction* applies and

$$\langle \{s\}, \text{align}_H \rangle \rightsquigarrow \langle \{s\}, \text{align}_I \rangle.$$

We can also easily infer that

$$\langle \{s, x\}, \text{align}_H \rangle \rightsquigarrow \langle \{s\}, \text{align}_I \rangle.$$

However, $\{s, x\}$ is not the minimal inducing set. Therefore, Rule *Single-Step* applies and we have

$$\langle \Lambda_x \cdot \text{align}_H \cdot \text{align}_I, \{s\}, \phi \rangle \xrightarrow{\text{cp}} \langle \Lambda_x \cdot \text{align}_H, \{s\}, \{\mathcal{I}_E\} \rangle$$

Note that $\text{defpoint}(s, \mathcal{I}_H) = \mathcal{I}_E$. Other steps are similarly conducted.

4.3 Root Cause Acquisition

We can prove that the causal path always starts with the root cause when certain conditions are satisfied. In other

words, such paths are rooted at the faulty statement, end at the failure manifestation point, and have causally related intermediate steps, thus serving as an explanation for the failure. According to Rule *Single-Step* in Fig. 7, the causal path computation is a backward process that updates the relevant variable set Γ , driven by the alignment Λ . We say the process terminates if the evaluation rule cannot proceed, namely, one of the three premises of Rule *Single-Step* is not satisfied.

THEOREM 1. *If the fault is a single definition mutation as follows.*

$$x =^{\ell_0} e_1; \dots \Rightarrow x =^{\ell_0} e_2; \dots$$

, and the faulty statement is evaluated once, then

$$\langle \Lambda, \Gamma, \phi \rangle \xrightarrow{cp} \dots \xrightarrow{cp} \langle \dots \cdot \mathbf{align}_0 \cdot \mathbf{align}_1, \{x\}, \mathcal{CP} \rangle \xrightarrow{sp}$$

holds. In other words, the causal path computation must terminate at entry \mathbf{align}_0 with the index $\mathcal{I}_0 \cdot \ell_0$ and the relevant set at the following alignment entry \mathbf{align}_1 must be a singleton containing x .

PROOF. Let $\mathbf{entry}_{0/\mathcal{I}_1}^{\mathcal{F}}$ be the shorthand for $\mathbf{entry}(\mathcal{I}_{0/\mathcal{I}_1} \cdot \ell_{0/\mathcal{I}_1}, \sigma_{0/\mathcal{I}_1}^{\mathcal{F}})$. Let the causal path computation terminate at \mathbf{align}_0 , which is the shorthand for $\langle \mathcal{I}_0 \cdot \ell_0, \sigma_0^{\mathcal{P}}, \sigma_0^{\mathcal{F}} \rangle$. Let \mathbf{align}_1 be the following alignment entry. Operator \subset is used to represent the sub-sequence relation. Operator $\xrightarrow{t^{\mathcal{F}}}$ represents tracing rule application in the failing evaluation. Operator $\rightarrow^{\mathcal{F}}$ represents semantics rule application in the failing evaluation.

$$\begin{aligned} \mathbf{align}_0 \cdot \mathbf{align}_1 \subset \Lambda = \mathcal{T}^{\mathcal{P}} \bowtie_{\mathcal{I}} \mathcal{T}^{\mathcal{F}} &\Rightarrow \\ \mathcal{T}^{\mathcal{F}} = \mathcal{T}_0 \cdot \mathbf{entry}_0^{\mathcal{F}} \cdot \dots \cdot \mathbf{entry}_1^{\mathcal{F}} \cdot \dots &\Rightarrow \\ \langle s^{\ell_0}; S_0, \sigma_0^{\mathcal{F}}, \mathcal{I}_0, \mathcal{T}_0 \rangle \xrightarrow{t^{\mathcal{F}}} \langle r_0; S_0, \sigma_0^{\mathcal{F}'}, \mathcal{I}'_0, \mathcal{T}_0 \cdot \mathbf{entry}_0^{\mathcal{F}} \rangle \xrightarrow{t^{\mathcal{F}}} \dots \xrightarrow{t^{\mathcal{F}}} \\ \langle s^{\ell_1}; S_1, \sigma_1^{\mathcal{F}}, \mathcal{I}_1, \mathcal{T}_0 \cdot \mathbf{entry}_0^{\mathcal{F}} \cdot \dots \rangle \xrightarrow{t^{\mathcal{F}}} \\ \langle r_1; S_1, \sigma_2^{\mathcal{F}}, \mathcal{I}_2^{\mathcal{F}}, \mathcal{T}_0 \cdot \mathbf{entry}_0^{\mathcal{F}} \cdot \dots \cdot \mathbf{entry}_1^{\mathcal{F}} \rangle &\Rightarrow \\ \langle s^{\ell_0}; S_0, \sigma_0^{\mathcal{F}}, \mathcal{I}_0 \rangle \rightarrow \dots \rightarrow^{\mathcal{F}} \langle s^{\ell_1}; S_1, \sigma_1^{\mathcal{F}}, \mathcal{I}_1 \rangle & \quad (1) \end{aligned}$$

Similarly, we have

$$\langle s^{\ell_0}; S_0, \sigma_0^{\mathcal{P}}, \mathcal{I}_0 \rangle \rightarrow \dots \rightarrow^{\mathcal{P}} \langle s^{\ell_1}; S_1, \sigma_1^{\mathcal{P}}, \mathcal{I}_1 \rangle \quad (2)$$

Assume

$$\Delta = \{x \mid \sigma_0^{\mathcal{P}}(x) \neq \sigma_0^{\mathcal{F}}(x)\} \neq \phi, \quad (3)$$

meaning there are value differences at index $\mathcal{I}_0 \cdot \ell_0$. It can be inferred that $\{\Delta \mapsto \sigma_0^{\mathcal{F}}(\Delta)\} \sigma_0^{\mathcal{P}} = \sigma_0^{\mathcal{F}}$. Together with (1), we can infer that

$$\langle s^{\ell_0}; S_0, \{\Delta \mapsto \sigma_0^{\mathcal{F}}(\Delta)\} \sigma_0^{\mathcal{P}}, \mathcal{I}_0 \rangle \rightarrow \dots \rightarrow^{\mathcal{F}} \langle s^{\ell_1}; S_1, \sigma_1^{\mathcal{F}}, \mathcal{I}_1 \rangle. \quad (4)$$

Since there are value differences at $\mathcal{I}_0 \cdot \ell_0$, it is implied that the faulty statement has been evaluated, leading to program state being contaminated. According to the premise that the faulty statement is evaluated only once and (4), we can infer that

$$\langle s^{\ell_0}; S_0, \{\Delta \mapsto \sigma_0^{\mathcal{F}}(\Delta)\} \sigma_0^{\mathcal{P}}, \mathcal{I}_0 \rangle \rightarrow \dots \rightarrow^{\mathcal{P}} \langle s^{\ell_1}; S_1, \sigma_1^{\mathcal{F}}, \mathcal{I}_1 \rangle. \quad (5)$$

Intuitively, it says since the evaluation from $s^{\ell_0}; S_0$ to $s^{\ell_1}; S_1$ with the faulty program does not involve the faulty statement, the evaluation must be identical to the evaluation with the correct program.

Due to (3), (5), and the tautology $\sigma_1^{\mathcal{F}}(\Gamma_1) = \sigma_1^{\mathcal{F}}(\Gamma_1)$, the *Induction Rule* can be applied to conclude that

$$\langle \Delta, \mathbf{align}_0 \rangle \rightsquigarrow \langle \Gamma_1, \mathbf{align}_1 \rangle \quad (6).$$

Intuitively, it says that replacing the variables in Δ at $\mathcal{I}_0 \cdot \ell_0$ essentially mutates the passing evaluation to the failing evaluation such that it dictates that the same faulty state Γ_1 is observed at $\mathcal{I}_1 \cdot \ell_1$ in the mutated run, we say Δ induces Γ_1 .

Next, we want to prove that there exists a non-empty minimal subset $\Gamma_0 \subseteq \Delta$ that satisfies $\langle \Gamma_0, \mathbf{align}_0 \rangle \rightsquigarrow \langle \Gamma_1, \mathbf{align}_1 \rangle$. Assume the opposite is true, the minimal set is empty such that

$$\langle \phi, \mathbf{align}_0 \rangle \rightsquigarrow \langle \Gamma_1, \mathbf{align}_1 \rangle \quad (7)$$

From Rule *Induction*, we infer that

$$\begin{aligned} \langle s^{\ell_0}; S_0, \{\} \sigma_0^{\mathcal{P}}, \mathcal{I}_0 \rangle \rightarrow \dots \rightarrow^{\mathcal{P}} \langle s^{\ell_1}; S_1, \sigma', \mathcal{I}_1 \rangle & \quad (8) \\ \sigma'(\Gamma_1) = \sigma_1^{\mathcal{F}}(\Gamma_1) & \quad (9) \end{aligned}$$

From (2) and (8), we can infer that $\sigma' = \sigma_1^{\mathcal{P}}$. Since $\Gamma_1 \neq \phi$, it must be true that $\sigma_1^{\mathcal{P}}(\Gamma_1) \neq \sigma_1^{\mathcal{F}}(\Gamma_1)$, which is a contradiction to (9). Hence, the assumption (7) is not true, there must be a non-empty minimal subset of Δ that induces Γ_1 . According to Rule *Single-Step*, the causal path computation can progress, which contradicts to the premise that the computation cannot progress. Hence, the assumption (3) cannot be true. Intuitively, so far we have proved that the causal path computation cannot terminate at a point that has value differences in the two runs.

Hence, when the causal path computation cannot progress from \mathbf{align}_1 to \mathbf{align}_0 , $\Delta = \phi$. Assume

the terminating statement s^{ℓ_0} is not the faulty statement. (10)

In the failing evaluation, we have the following steps that generate two consecutive entries in $\mathcal{T}^{\mathcal{F}}$ with the first one being the terminating entry \mathbf{entry}_0 .

$$\begin{aligned} \langle s^{\ell_0}; S_0, \sigma_0, \mathcal{I}_0, \mathcal{T}_0 \rangle \xrightarrow{t^{\mathcal{F}}} \langle r_0^{\mathcal{F}}; S_0^{\mathcal{F}}, \sigma_1^{\mathcal{F}}, \mathcal{I}_1^{\mathcal{F}}, \mathcal{T}_0 \cdot \mathbf{entry}_0 \rangle \xrightarrow{t^{\mathcal{F}}} \dots \xrightarrow{t^{\mathcal{F}}} \\ \langle s^{\ell_1^{\mathcal{F}}}; S_1^{\mathcal{F}}, \sigma_1^{\mathcal{F}}, \mathcal{I}_1^{\mathcal{F}}, \mathcal{T}_0 \cdot \mathbf{entry}_0 \rangle \xrightarrow{t^{\mathcal{F}}} \\ \langle r_1^{\mathcal{F}}; S_1^{\mathcal{F}}, \sigma_2^{\mathcal{F}}, \mathcal{I}_2^{\mathcal{F}}, \mathcal{T}_0 \cdot \mathbf{entry}_0 \cdot \mathbf{entry}(\mathcal{I}_1^{\mathcal{F}} \cdot \ell_1^{\mathcal{F}}, \sigma_1^{\mathcal{F}}) \rangle \end{aligned}$$

Similarly, in the passing evaluation, we have the following steps that generate two consecutive entries in $\mathcal{T}^{\mathcal{P}}$.

$$\begin{aligned} \langle s^{\ell_0}; S_0, \sigma_0, \mathcal{I}_0, \mathcal{T}_0 \rangle \xrightarrow{t^{\mathcal{P}}} \langle r_0^{\mathcal{P}}; S_0^{\mathcal{P}}, \sigma_1^{\mathcal{P}}, \mathcal{I}_1^{\mathcal{P}}, \mathcal{T}_0 \cdot \mathbf{entry}_0 \rangle \xrightarrow{t^{\mathcal{P}}} \dots \xrightarrow{t^{\mathcal{P}}} \\ \langle s^{\ell_1^{\mathcal{P}}}; S_1^{\mathcal{P}}, \sigma_1^{\mathcal{P}}, \mathcal{I}_1^{\mathcal{P}}, \mathcal{T}_0 \cdot \mathbf{entry}_0 \rangle \xrightarrow{t^{\mathcal{P}}} \\ \langle r_1^{\mathcal{P}}; S_1^{\mathcal{P}}, \sigma_2^{\mathcal{P}}, \mathcal{I}_2^{\mathcal{P}}, \mathcal{T}_0 \cdot \mathbf{entry}_0 \cdot \mathbf{entry}(\mathcal{I}_1^{\mathcal{P}} \cdot \ell_1^{\mathcal{P}}, \sigma_1^{\mathcal{P}}) \rangle \end{aligned}$$

Note that since the above passing and failing evaluation steps start from the same configuration, reflecting they align at $\mathcal{I}_0 \cdot \ell_0$ and do not have value differences. According to the assumption (10), it must be true that $\ell_1^{\mathcal{P}} = \ell_1^{\mathcal{F}}$, $\sigma_1^{\mathcal{P}} = \sigma_1^{\mathcal{F}}$, and $\mathcal{I}_1^{\mathcal{P}} = \mathcal{I}_1^{\mathcal{F}}$. Hence, $\mathbf{entry}_1^{\mathcal{P}} = \mathbf{entry}_1^{\mathcal{F}}$. In other words, $\mathbf{align}_0 \cdot \mathbf{align}_1 \subset \mathcal{T}^{\mathcal{P}} \bowtie_{\mathcal{I}} \mathcal{T}^{\mathcal{F}}$. Since $\sigma_1^{\mathcal{P}} = \sigma_1^{\mathcal{F}}$, $\sigma_1^{\mathcal{P}}(\Gamma_1) = \sigma_1^{\mathcal{F}}(\Gamma_1)$, which is a contradiction to the premise of Γ_1 being a non-empty relevant set. Intuitively, it says a benign statement cannot produce a faulty value from correct values. Hence, assumption (10) does not hold, s^{ℓ_0} must be the faulty statement. Finally, it is easy to infer that Γ_1 is a singleton as the faulty statement can only define one variable. \square

We haven't proven the property in the presence of multi-line faults and multiple occurrences of the single-line fault. However, the experiments in Section 6 show that in practice, our technique is still highly effectively in these cases.

4.4 Fast Pass Optimization

The rules in Fig. 7 demand step-by-step backward evaluation. If there are n entries in the alignment Λ , the *Single-Step* rule will be applied n times, which implies the *Induction* rule will be applied n times, entailing n program re-evaluations and value difference minimizations. We observe that computations at many of these steps are redundant and can be optimized. The modified rules are presented in Fig. 8.

The goal of the fast pass rules is to avoid the expensive program re-evaluation and value difference minimization. Both rules require that the two consecutive alignment entries during causal path computation also denote consecutive entries in both the passing and the failing traces, expressed as the first two premises in both rules. Rule *Def-Free* specifies that if none of the relevant variables in Γ is defined at the statement with label ℓ_0 in either evaluation, the same relevant set and the same causal path are resulted in the earlier alignment entry without computation. Note that the statements with label ℓ_0 in the two evaluations may define different variables as in the two program versions are different. Rule *Definition* specifies that when the statement at ℓ_0 in the two evaluations use the same set of source variables to define the same destination variable and a variable in Γ is defined at ℓ_0 , the right hand side variables of the definition are added to the relevant set if they represent minimal value differences. Before we prove the two rules are safe, i.e. producing the same results as Rule *Single-Step*, let us revisit our running example to demonstrate how these rules are used.

Example. In Fig. 2, Γ is initialized to $\{s\}$ at the end of the program execution. The causal path computation is carried out in a backward order on the alignment entries. The first two entries, denoted by $\textcircled{1}$ and $\textcircled{2}$, are consecutive in both runs and they do not define s , hence Rule *Def-Free* applies and the expensive computation on these steps can be skipped. As $\textcircled{3}$ does not immediately follow its preceding alignment entry with index $[S, 2^T, 2^T, 6]$ in the passing run, fast passes cannot be taken. The expensive *Single-Step* and hence *Induction* rules are applied. After this, fast passes can be taken again. At $\textcircled{4}$, Rule *Definition* can be applied such that s is removed from the relevant set and the operand of the assignment that is a value difference, pointer p , is directly inserted into the relevant set.

THEOREM 2. *Fast pass rules are safe, meaning they produce the same results as Rule Single-Step.*

PROOF. Due to space limitation, we present a proof sketch as follows.

Consider Rule *Def-Free* first. Since none of the variables in Γ are defined at $\mathcal{I}_0 \cdot \ell_0$ in either evaluation, Γ still represents value differences at $\mathcal{I}_0 \cdot \ell_0$ and hence satisfies the first condition in Rule *Induction*. Since $\mathcal{I}_0 \cdot \ell_0$ and $\mathcal{I}_1 \cdot \ell_1$ denote consecutive entries in both traces, s^{ℓ_0} must not be a predicate using a variable that has different values in the two evaluations, otherwise, different branches would have been taken, leading to unmatched next trace entries. Replacing the variables in Γ in the passing evaluation at $\mathcal{I}_0 \cdot \ell_0$ with their faulty values follows the same evaluation rule in the next step (i.e. control flow is not changed). Moreover, it results in the variables in Γ having the same faulty values at $\mathcal{I}_1 \cdot \ell_1$ because

none of them are re-defined, and hence the second and third conditions are satisfied. Hence, $\langle \Gamma, \text{align}_0 \rangle \rightsquigarrow \langle \Gamma, \text{align}_1 \rangle$ according to Rule *Induction*. The set Γ is also minimal, thus the *Def-Free* rule produces the same result as *Single-Step*.

Now consider Rule *Definition*. The premises specify that one of the value differences in Γ is defined at $\mathcal{I}_0 \cdot \ell_0$. It is easy to tell that all variables in Γ' must be value differences and hence the first premise of Rule *Induction* is satisfied. Following the same reasoning as described earlier, none of the variables in Γ' is used as a predicate variable at $\mathcal{I}_0 \cdot \ell_0$. Hence, replacing the variables in Γ' does not change the next evaluation rule application. Furthermore, it leads to the value differences in Γ . Hence, Rule *Induction* can be applied. Since Γ' is also minimal, the *Def-Free* rule produces the same result as the *Single-Step* rule. \square

5. PRACTICAL CHALLENGES

In this section, we discuss how we handle a number of practical issues such that our technique can scale to real world C programs.

5.1 Handling Methods

Methods can be handled with easy extensions to the core language. At method invocations, the *Assignment* rule is applied to assign actual arguments to the formal arguments. \mathcal{I} is updated by pushing the label of the invocation, indicating entry into another level of nesting (the method body). The label is popped when the method returns. The aforementioned theorems still hold in the presence of methods. The rules and proofs are elided.

5.2 Handling Complex Types

$$\frac{\text{ispointer}(p) \quad \text{ispointer}(p') \quad \text{type}(p) = \text{type}(p')}{\bar{f} = \text{type}(p) \quad p \rightarrow \bar{f} = p' \rightarrow \bar{f}}{p = p'} \quad (\text{Equivalence})$$

$$\frac{\text{ispointer}(p) \quad \text{ispointer}(p') \quad \text{type}(p) = \text{type}(p')}{p \rightarrow \bar{f} \neq p' \rightarrow \bar{f} \quad \bar{f} \subseteq \text{type}(p) \quad p \rightarrow \bar{f} := p' \rightarrow \bar{f}}{p := p'} \quad (\text{Val - Replacement})$$

ispointer() decides if a variable contains a pointer. *type()* returns the type of data structure pointed by a pointer. *fields()* returns the fields of a data structure. Symbol $:=$ represents value replacement.

Figure 9: Pointer Comparison and Replacement.

Arrays and structs can be handled by treating each array element or each field as an individual variable. The main challenge lies in handling pointers.

Pointers, especially heap pointers, may contain different address values in different runs even though they point to the same data structure in practice. To address this issue, we propose the pointer comparison and value replacement rules in Fig. 9. Note that these rules are no longer syntactically bounded by our simple language. Rule *Equivalence* compares two pointers by recursively comparing their fields. Rule *Val-Replacement* specifies copying a pointer to another

DEFINITIONS

$lhs(\ell)$ returns the variable that is defined at statement s^ℓ ;
 $rhs(\ell)$ returns the variables that are used at statement s^ℓ ;
 $\text{entry}_0^{\mathcal{P}}$ is a shorthand for $\text{entry}(\mathcal{I}_0 \cdot \ell_0, \sigma_0^{\mathcal{P}})$; Other entry shorthands are similarly defined.

FAST PASS RULES

$$\frac{\text{entry}_0^{\mathcal{P}} \cdot \text{entry}_1^{\mathcal{P}} \subset \mathcal{T}^{\mathcal{P}} \quad \text{entry}_0^{\mathcal{F}} \cdot \text{entry}_1^{\mathcal{F}} \subset \mathcal{T}^{\mathcal{F}} \quad lhs^{\mathcal{P}}(\ell_0) \cap \Gamma = \phi \quad lhs^{\mathcal{F}}(\ell_0) \cap \Gamma = \phi}{\langle \Lambda \cdot \text{align}_0 \cdot \text{align}_1, \Gamma, \mathcal{CP} \rangle \xrightarrow{cp} \langle \Lambda \cdot \text{align}_0, \Gamma, \mathcal{CP} \rangle} \quad (Def - Free)$$

$$\frac{\text{entry}_0^{\mathcal{P}} \cdot \text{entry}_1^{\mathcal{P}} \subset \mathcal{T}^{\mathcal{P}} \quad \text{entry}_0^{\mathcal{F}} \cdot \text{entry}_1^{\mathcal{F}} \subset \mathcal{T}^{\mathcal{F}} \quad lhs^{\mathcal{P}}(\ell_0) = lhs^{\mathcal{F}}(\ell_0) \quad rhs^{\mathcal{P}}(\ell_0) = rhs^{\mathcal{F}}(\ell_0)}{lhs^{\mathcal{P}}(\ell_0) \cap \Gamma \neq \phi \quad \Gamma' = (\Gamma - lhs^{\mathcal{P}}(\ell_0)) \cup \{x \mid x \in rhs^{\mathcal{P}}(\ell_0) \text{ AND } \sigma^{\mathcal{P}}(x) \neq \sigma^{\mathcal{F}}(x)\}} \langle \Lambda \cdot \text{align}_0 \cdot \text{align}_1, \Gamma, \mathcal{CP} \rangle \xrightarrow{cp} \langle \Lambda \cdot \text{align}_0, \Gamma', \mathcal{CP} \cup \text{defpoint}(\Gamma') \rangle \quad (Definition)$$

Figure 8: Fast Passes.

is carried out by recursively copying the fields that have different values, which may entail allocating new space. Our technique currently considers all void pointers as holding the same value.

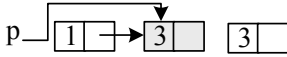


Figure 10: Pointer replacement at \textcircled{E} in Fig. 2.

Consider the example in Fig. 2. At \textcircled{E} , in order to decide if p is a relevant variable, the algorithm replaces S_2 with S_3 in the passing run and tests if $s=4$ can result at \textcircled{F} . According to Rule *Val-Replacement*, the second element of the linked list is set to have the values of the third element, including the NULL value for the `next` field. The resulting linked list state is presented in Fig. 10. It leads to $s=4$ at \textcircled{F} . Note that the replacement also changes the state of the linked list due to aliasing. However, the perturbation does not affect the causal path computation, which is a local computation, meaning it only concerns consecutive alignment entries.

5.3 Value Difference Minimization

In Rule *Single-Step*, the Γ_0 of the preceding entry is required to be minimal. Our implementation first computes the set of value differences and then tries to find the minimal subset that still satisfies the *Induction* rule. Theoretically, the minimality requirement has an exponential complexity, which is often too expensive. We make use of the *Delta Debugging* algorithm [38], which is a binary search like algorithm. It has much better efficiency but finds the subsets with 1-minimality, meaning any one-element-less subsets of the 1-minimal subset do not satisfy the *Induction* rule while the 1-minimal subset itself does. In practice, we have found 1-minimality is sufficient in most cases.

5.4 Static Version Alignment

When a regression bug is analyzed, the prior, non-regressing version of the program may be used as the correct reference execution with which to compare. However, because this static program is itself different from the buggy one, we need to first assign labels to matching statements in the two versions. This static statement alignment is formed by finding the maximal tree correspondence between the control dependence trees of each function in the two program

versions [2]. Significant differences in variables and types are simply viewed as incomparable differences between the two programs.

5.5 Composing Reference Executions

If the non-regressing version of the program is not available, which is the common case in the real world, we have to find a reference execution that represents the correct behavior. One simple idea is to use a passing execution of the faulty program that is similar to the failing run but generated from a different input [7, 37, 9, 28, 4]. Unfortunately, since the two executions are derived from two different inputs, the computed causal path often explains the semantic differences resulting from input differences instead of the faulty statements. We have detailed and exemplified discussion regarding this problem in our technical report [31].

In contrast, we propose to *construct a reference execution out of the failing execution*. More particularly, we leverage the technique called predicate switching [39] that can patch a failing execution by switching the branch outcome of a dynamic predicate. It was reported that for a set of Unix utility programs, only 0.26%-8.33% of an execution does not converge on a predicate. This implies that, in a failing run, faulty state caused by faulty non-predicate instruction executions converges and drives a predicate to take the wrong branch. Therefore, patching the predicate has the same effect as patching these arbitrary faulty non-predicate executions. Note that predicates have a boolean domain in which searching for a patch is much more tractable than in alternative domains. In [39], it was shown that by switching the branch outcome of a single predicate instance, 9 out of 12 real bugs for Unix utility programs can be patched. In order to further validate the observation, in this project we have conducted a larger scale study on the Siemens [8] debugging suite. We found that 80% of the over 9000 failing runs can be patched. The maximum time required to patch a failure is under 5 minutes. More details can be found in our technical report [31].

6. EVALUATION

A proof-of-concept system has been developed to analyze C programs. Program instrumentation is implemented using CIL [25]. Compared to binary instrumentation, the source-to-source translation feature of CIL provides better symbol table manipulation and avoids the technique being ex-

posed to very low level execution details such as registers and stack operations. State differencing and minimization algorithms are implemented using Python, reusing part of the open source delta-debugging package [7].

Benchmark	# Failures	# \neg CO	Path Length	Roots
schedule	28	20	8.1	16
schedule2	28	12	7.9	4
print_tokens	24	16	4.21	4
print_tokens2	32	20	12.4	20
replace	38	38	57	34
tcas	40	40	9	40

Table 1: Results for Siemens programs.

In the first experiment, we used the Siemens programs [8]. For each program, Siemens provides the correct version of the program, a set of mutants with faults injected, and a test suite. The injected faults include multiple line faults and code omissions (the fault is that a piece of code is eliminated). For the first four programs, we used all the available mutants that terminate with failures. For programs **replace** and **tcas**, the first ten mutants were selected, as there are too many mutants for our study. For each mutant, we strictly used the first four failing unit tests for reproducibility. Note that an injected fault may fail in multiple unit tests. A failing test case of the mutant and the corresponding passing run of the correct version are subject to our technique. Table 1 presents the results. In the table, path length refers to the number of elements in the causal path. Number of failures denotes how many failing test cases have gone through our technique. Two of the failing cases in **replace** crashed our system and the results cannot be presented. In some other case, the mutants did not exhibit any failure or did not terminate. **Roots** refers to how many of the causal paths start with the root cause. Note that in many cases, the faulty statement gets executed multiple times. We consider a path to start with the root cause if it starts with an instance of the faulty statement. **# \neg CO** lists how many of the failures are not due to code omission. The root causes for omission faults are not captured by definition because they are absent from failing executions.

The following observations can be made from the results.

Observation One. The causal paths generally capture the root causes as their starting points, except for code omission errors. For the last three benchmarks, most of the root causes are captured. Manual inspection of these cases show that the causal paths precisely explain the failures. Let’s use one example to illustrate this observation. The **replace** program in the Siemens suite substitutes a pattern in an input string with an alternative user provided pattern. In the tenth mutant, the injected fault is that **esc()** returns ‘\0’ instead of an escape pattern. This is propagated as seen in the causal path in Fig. 11 until it is assigned to a position in a substitution string, truncating it and causing several characters to be omitted from the output. Observe that the root cause is perfectly captured at the start of the causal path, and each step causes or collaborates with successive steps to create the failure. For code omission cases, the causal paths root at definitions that should have been overwritten by the root causes, and capture how such omitted definitions pollute the program state and finally lead to the failures. We argue that such information is also very useful.

Code Snippet

```

esc(s, i):
1  result = '\0'; //was '@'
2  return result
addstr(c, outset, j, maxset):
3  outset[*j] = c;
makesub(arg, from, delim, sub):
4  escjunk = esc(arg, &i);
5  junk = addstr(escjunk, sub, &j, 100);
putsb(lin, s1, s2, sub):
6  while (sub[i] != '\0') {
7    fputc(lin[j], stdout);
main(argc, argv):
8  makeres = makesub(arg, 0, '\0', sub);*
9  putsb(lin, i, m, sub);*

```

Ideal Causal Path:

```

At 1, result is given '\0'
At 4, escjunk is given '\0'
At 5, arg c is given '\0'
At 3, sub/outset[59] is given '\0'
At 6, (sub[i] != '\0') is false
Thus the output ... differs from ...

```

Figure 11: The causal path for a replace case

Observation Two. Most causal paths are succinct, meaning most failures can be explained in a few steps. The large average length for **replace** is caused by a few cases whose causal paths contain loop iterations, i.e. relevant faulty variables are loop related. Recall that our causal paths capture statement executions instead of static statements.

Observation Three. In some cases, our technique does not capture the root causes, especially for benchmarks **schedule2** and **print_tokens**. The causal path computation terminates prematurely for these cases. The main reason is that these programs have external states, namely, data is stored on files. External states are invisible to our current system. Since faulty states unfortunately reside externally and the internal states are identical, our technique does not find any value differences and hence terminates. In a few cases, our current solution to pointer value comparison and replacement is not sufficient in the presence of substantial aliasing. We leave it to our future work. However, these partial paths still largely explain the failures.

In the second experiment, we applied the prototype to 6 real bugs for 3 Unix utility programs, including **bc**, **cmp**, and **grep**. Additional programs and bugs can be found in our technical report[31]. The tests were performed on a 2 GHz dual core machine with 2GB of RAM. For each bug, we first used predicate switching to isolate a critical predicate that could be negated to generate the reference execution. Using this reference execution, we extracted the causal path that explains the bug. Each phase was timed independently, and results from the experiment can be found in Table 2. For each program, we list the number of inducing definitions in the causal path in *Path Length*, time taken to derive the causal path without optimization in *Time w/o FP* and the time taken using the fast pass optimization in *Time w/ FP*. The predicate switching phase required fewer than 20 seconds in every case. Note that the naïve algorithm to derive causal paths, computing the relevant state at every alignment, timed out after 1 hour in several cases, as denoted by ‘-’. As seen in Table 2, the fast pass approach generally takes from a few seconds to several minutes on these medium sized programs, due to the very frequent applications of the

Code Snippet:

```
main(argc, argv):
1 while (option != -1) {
2   switch (option) {
3     case 'm':
4       max_count = value; break;
5     ...
6   }
7   grep(desc, file, stats);
8   grep(fd, file, stats):
9   outfile = max_count;
10  grepbuff(beg, lim);
11  grepbuff():
12  if (!out_left) {
13    outfile--;
14  }
15  prpending(lim):
16  if (!out_left)
17    EGexecute(last, nldiff, size, 0);
```

Causal Path:

At 1, (option != -1) is true
At 4, max_count is given 1
At 7, outfile is given 1
At 10, outfile is given 0
At 11, (!out_left) is true
So EGexecute() crashes at 12

Figure 12: Causal path for segfault in grep 2.5.1

fast pass rules. Finally, the *Root* column denotes whether or not the causal path captures the faulty region of code that was actually patched to fix the bug. Observe that all but one of the chains included the faulty region.

Looking closely at one particular case, `grep` has the abilities of printing out only the first few occurrences of a pattern in a file, using option ‘-m’, and of printing out lines before and after any matching pattern to provide context. In version 2.5.1, using these together leads to a segfault. Interestingly, the failure is not explained and no fix was given in the bug report. However, the causal path of the segfault, in Fig. 12, both explains the failure and leads to potential fixes. The critical predicate of the path, at line 1, is true in the buggy run, which enables the option for printing only the first n matches of a pattern in a file. As a result, this value n is stored in `max_count` on line 4 and from there into `outfile` on line 7. As a result, when only one match should be printed, i.e. `value=1` at line 4, `outfile` receives the value 0 on line 10. Thus, `(!out_left)` is true and `EGexecute()` is called at line 12. Calling `EGexecute()` at this point immediately and always crashes the program by violating the function’s preconditions. The potential fixes are to prevent `EGexecute()` being called at line 12 with the aforementioned options or to alter or follow the preconditions of the function. Observe that the causal path narrowly captures all behavior that is relevant to explaining the failure. In this case, there is no single possible fix, but a developer would be able to use the explanation to understand the failure and choose an appropriate one.

7. RELATED WORK

Delta Debugging. The most relevant related work is delta debugging [38, 37, 7, 13]. In particular, [7, 37] try to compute a notion of causal paths of failures by comparing a failure with a similar but passing execution. We have conducted experiments to compare our technique with theirs regarding failure explanation. The results show that our technique supersedes delta debugging largely because our approach composes an execution from the failure, instead of using a different run and aligns executions before compar-

son. Details can be found in [31].

Large Scale Comparison based Fault Localization Many fault localization techniques [12, 15, 28, 21, 23, 11, 3, 22, 27, 26, 29, 42, 18] compute fault candidates by looking at many executions, both passing and failing. Compared to the proposed work, these techniques require a large number of runs. They lack or are limited in the ability to explain failures. They usually produce a ranked candidate set containing static statements. Inspecting such a set is labor-intensive. In [18], control flow paths that connect fault candidates are also reported to facilitate understanding. In [6], the cost of producing such paths is reduced.

Other Fault Localization. Some fault localization techniques compare execution profiles of a failure with a small number of correct runs (usually one) [28, 34, 9, 4]. They only compare simple profiles such as control flow paths and code coverage, and hence are not as effective. They don’t perform causality testing. Like other fault localization techniques, they produce a static fault candidate list without explanations. Crisp [5] is a technique that helps developers in regression testing, allowing developers to selectively apply a set of code edits and then observe the correlation between code edits and regression failures. In [14], traces of regression failures are compared with passing runs through multiple views. The technique has similar limitations. While this paper lays down the formal foundation for causal path computation and presents the comprehensive design, an empirical comparison of the efficiency of various algorithms can be found in [32].

Program Slicing. Program slicing [35, 20, 1, 33] was introduced as an aid to debugging. Compared to fault localization, slicing features the ability to capture causality through program dependencies. However, slicing tends to produce very fat slices that contain all possible paths that lead to the failure, including those that do not originate from the faulty statement. Although various techniques have been proposed to prune slices [10, 40, 30], without using a reference execution to exclude benign dependences, inspecting pruned slices still requires non-trivial human effort. Dicing [4] is a slicing technique that aggregates slices from multiple executions; however, the involved simple set manipulations undermine causality in slices and make the resulting dices hard to understand. Furthermore, it does not handle cases in which the faulty statement occurs in both the benign and faulty slices. PSE [24] can be considered as a special technique that tries to explain a failure by performing state-aware static slicing. It only works on memory errors.

8. CONCLUSIONS

We propose a highly effective technique to automatically compute the causal path of a failure. The idea is to compare the failing execution with a reference execution, which could be generated from the non-regression version on the same input or by patching the failing execution if the non-regression version is not available. The key is to align executions before they are compared. A formal model is proposed, which allows us to formally describe the technique and prove important properties. Safe optimizations are also proposed to improve scalability. Our results on synthetic bugs and real bugs show that the proposed technique can precisely explain failures with reasonable cost.

9. REFERENCES

Table 2: Examined bugs and their causal path properties. ‘Bug’ is an identifier for a bug report, if applicable. ‘Path length’ is the number of inducing definitions in the causal path. ‘Time w/o FP’ is the time in seconds taken to derive the causal path without the fast pass optimization. ‘Time w/ FP’ is the time in seconds taken to derive the causal path using the optimization.

Program	Failing Ver.	Bug	Path Length	Time w/o FP	Time w/ FP	Roots
bc	1.06	savannah.gnu.org/bugs/?21898	4	-	634	no
cmp	11 Dec 2001	savannah.gnu.org...diffutils...cmp.c...rev1.26	3	43	16	yes
cmp	9 Mar 2006	debian bug 356083	3	198	21	yes
cmp	7 May 2006	...gnu.utils.bug/2006-05/msg00018.html	7	515	48	yes
grep	2.5.1	http://savannah.gnu.org/bugs/?13920	7	-	109	yes
grep	2.5.1	http://savannah.gnu.org/bugs/?19491	9	-	384	yes

- [1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, 1990.
- [2] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337:217–239, 2005.
- [3] Y. Brun and M. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, pages 480–490, 2004.
- [4] T. Y. Chen and Y. Y. Cheung. Dynamic program dicing. In *ICSM*, pages 378–385, 1993.
- [5] O. Chesley, X. Ren, B. Ryder, and F. Tip. Crisp—a fault localization tool for java programs. In *ICSE*, pages 775–779, 2007.
- [6] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. *ICSE*, 2009.
- [7] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.
- [8] H. Do, S.G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4).
- [9] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *CC*, pages 80–95, 2006.
- [10] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE’05*.
- [11] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE’02*.
- [12] M. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.
- [13] R. Hildebrandt and A. Zeller. Simplifying failure-inducing input. In *ISSTA*, 2000.
- [14] K. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *PLDI’09*, 2009.
- [15] T. Huang, P. Chou, C. Tsai, and H. Chen. Automated fault localization with statistically suspicious program states. In *LCTES*, 2007.
- [16] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE ’94*, pages 191–200, 1994.
- [17] D. Jackson, M. Thomas, and L. Millett. *Software for Dependable Systems: Sufficient Evidence?* Committee on Certifiably Dependable Software Systems, National Research Council, 2007.
- [18] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE ’07*, pages 184–193, 2007.
- [19] P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, 2009.
- [20] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.
- [21] B. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, 2005.
- [22] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.
- [23] C. Liu, X. Yan, L. Fei, J. Han, and S.P. Midkiff. Sober: statistical model-based bug localization. In *ESEC/FSE*, 2005.
- [24] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. In *FSE*, 2004.
- [25] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC ’02*, 2002.
- [26] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. Reiss. Automated fault localization using potential invariants. In *AADEBUG’03*, pages 287–296, 2003.
- [27] E. Renieris. *A Research Framework for Software-Fault Localization Tools*. PhD thesis, Brown University, Providence, Rhode Island, 2005.
- [28] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.
- [29] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *SIGSOFT Software Engineering Notes*, 22, 1997.
- [30] M. Sridharan, S.J. Fink, and R. Bodik. Thin slicing. In *PLDI ’07*, pages 112–122, 2007.
- [31] W. N. Sumner and X. Zhang. Automatic failure inducing chain computation through aligned execution comparison. In *Technical Report 08-023*, Purdue U.
- [32] W. N. Sumner and X. Zhang. Algorithms for automatically computing the causal paths of failures. In *FASE*, 2009.
- [33] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3.
- [34] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *ASE*, pages 347–351, 2005.
- [35] M. Weiser. *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, Ann Arbor, Michigan, 1979.
- [36] B. Xin, N. Sumner, and X. Zhang. Efficient program execution indexing. In *PLDI*, 2008.
- [37] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, 2002.
- [38] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), 2002.
- [39] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE’06*.
- [40] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. *SIGPLAN Not.*, 41(6):169–180, 2006.
- [41] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *MICRO*, 2004.