Algorithms for Automatically Computing the Causal Paths of Failures

William N. Sumner and Xiangyu Zhang

Department of Computer Science, Purdue University
{wsumner,xyzhang}@cs.purdue.edu

Abstract. We have proposed an automated debugging technique that explains a failure by computing its causal path leading from the root cause to the failure. Given a failing execution, the technique first searches for a dynamic patch. Fine-grained execution comparison between the failing run and the patched run is performed to isolate the causal path. The comparison is enabled by precisely aligning the two executions. We herein propose and study two algorithms aiming at efficiency. We also evaluate the effectiveness and cost of our technique on a set of real bugs, including requirement bugs in which no a single or small set of statements can be blamed as the root cause. In such cases, understanding a failure is more important.

Keywords: debugging, automated debugging, execution indexing

1 Introduction

During debugging, developers often have a "correct" oracle execution in mind with which they compare a failing execution to identify faulty state and then understand the failure. We have proposed an automated debugging technique that mimics such a procedure [1]. Our technique computes the causal path of a failure, a subsequence of the failing run that explains the failure. It first searches for a dynamic patch for the failing run. The patch mutates the values of certain variables or control flow at runtime to produce the desired output. If such a patch can be found, which is true in most cases, the technique aligns the failing run and the patched run by establishing a mapping between instruction instances in the two runs. The states at aligned points are compared to identify faulty variables, and causality testing is performed to isolate subsets of faulty variables that are essential. The sequence of essential faulty states explains the failure.

The technique follows the direction pioneered by Zeller et al. in [2, 3]. In their work, the idea is to isolate the state transitions that are critical for a failure by *comparing the faulty execution with a similar but correct execution.* Compared to their work, we made progress in the following directions. *First*, we found that state comparisons need to be performed at rigidly corresponding points in the two respective executions. Due to the differences between the two executions, construction of such correspondence is challenging. In [2, 3], it was carried out in an ad-hoc way such that points that are not compatible may be selected for comparison. As a result, the computed chain may not be relevant

to the failure. We proposed using *execution indexing* (EI) [4] to align the two executions before comparison. EI is a technique that constructs a hierarchical tree of an execution based on program structure. Executions can be aligned through their trees. *Second*, our experience shows that using a different execution as the reference oracle often fails to explain the causality of the failure, as the reference execution is semantically different from the failing execution. Thus, comparing these two executions often exposes their inherent semantic differences instead of causality information for the failure. Our solution is to use a patched run derived from the same failure inducing input for comparison.

In our prior work [1], we have built a formal model of the technique, proposed an objective evaluation metric, and evaluated the proposed technique using the SIR suite [5]. Results show that our technique can produce high quality causal paths that often capture the root cause at the beginning, the failure point at the end, and causality information between consecutive steps. Comparison with the technique in [3] shows that our approach provides much better failure explanations. Details can be found in [1].

In this paper, we make the following contributions.

- We propose and study two algorithms that focus on cost-effectiveness. Both algorithms compute the same causal paths but differ in their efforts for reducing the number of state comparisons and causality tests. The first algorithm relies on the execution index tree to compute causal paths in a hierarchical manner. The second algorithm speculatively takes shortcuts during causal path derivation based on program dependence information.
- We evaluate our approach on a set of real world bugs collected from the Internet. Results show that our technique is effective in explaining failures. The shortcutting algorithm is orders of magnitude faster than the hierarchical algorithm.
- We use concrete examples to explicitly explain the unique features of our technique, including using EI to align executions before comparison and using a patched execution rather than a different execution as the reference.

2 Background

The goal of our technique is to compute the causal path of a failure. Assume we have the corrected version of the faulty program. The *ideal* causal path of a failure is computed by comparing the failing execution and the execution of the corrected program with the same input. The comparison is done by differencing the states at corresponding points in the two respective runs, starting from the failure point and proceeding backwards. In particular, faulty variables at a step are determined by comparing their values in the failing run with those in the correct run. Not all faulty variables are essential to the failure. Thus, we define the *failure inducing state* (FIS) of a step in the failing run as the minimal set of faulty variables that cause the failure or the FIS of the next step. Consider the example in Fig. 1. There is a fault at line 10. Provided with the input `a=2` and `b=3`, the program produces a failure observed at line 18, i.e. printing the incorrect

value 0. The trace and the states at each execution step of the failing run are presented in the second and the third columns, respectively. The two columns on the right show the trace and the states for the correct execution. At 15_2 of the failing run, meaning the second instance of statement 15, the faulty variables are x and z, as they have different values at 15_2 in the correct run. However, x is not essential to the failure, as replacing its value with that of $(x \mapsto 10)$ in the correct run does not mask the failure. Therefore, the FIS of 15_2 contains only z. Observe that the technique hinges on correctly identifying corresponding points in the two executions. This is defined as the *execution alignment* problem. While it is clear that 18_1 and 15_2 in the failing run align with 18_1 and 15_2 in the correct run, the alignments of 14_1, 5_1, 14_2 and 5_2 of the failing run are less clear. A simple strategy, which was used in [3, 2], is to align two execution points in the two respective executions that have the same statement, calling context, and instance count. Following such a strategy, 14_2 and 5_2 in the failing run do not have aligned points in the correct run and the 14_1s and 5_1s in the two respective executions align as shown in the figure. Such alignments are undesirable because they result in $FIS(14_1)=FIS(5_1)=\{i \mapsto 0\}$, i.e., the failure will not occur if the value of i is replaced with that of i at 14_1 or 5_1 in the correct run. In other words, the benign state $\{i \mapsto 0\}$ is mistakenly identified as failure inducing state.

Code	Failing Execution (a=2, b=3)		Ideal Execution (a=2, b=3)	
	Trace	State	Trace	State
		a, b, x, y, z, i, (x>b)		**a, b, x, y, z, i, (x>b)**
1. int **F** (int v, int w) {	8_1 input (a,b);	2, 3, 0, 0, 0, 0, -	8_1 input (a,b);	2, 3, 0, 0, 0, 0, -
2. return v+w;	9_1 F (a,b);	2, 3, 0, 0, 0, 0, -	9_1 F (a,b);	2, 3, 0, 0, 0, 0, -
3. }	2_1 y=a+b;	2, 3, 0, 5, 0, 0, -	2_1 y=a+b;	2, 3, 0, 5, 0, 0, -
4. int **G** (int v, int w) {	10_1 x=a+4;	2, 3, 6, 5, 0, 0, -	10_1 x=a;	2, 3, 2, 5, 0, 0, -
5. return v-w;	11_1 z=10;	2, 3, 6, 5,10, 0, -	11_1 z=10;	2, 3, 2, 5,10, 0, -
6. }	12_1 for (i=...) {	2, 3, 6, 5,10, 0, -	12_1 for (i=...) {	2, 3, 2, 5,10, 0, -
7.	13_1 if (x>b)	2, 3, 6, 5,10, 0,T	13_1 if (x>b)	2, 3, 2, 5,10, 0, F
8. input (a,b);	14_1 G (z, y);	2, 3, 6, 5,10, $\boxed{0}$ -	15_1 x=x+4;	2, 3, 6, 5,10, 0, -
9. y= F (a,b);	5_1 z=z-y;	2, 3, 6, 5, 5, $\boxed{0}$ -	12_2 for (i=...) {	2, 3, 6, 5,10, 1, -
10. x=a+4; // should be x=a;	15_1 x=x+4;	2, 3,10,5, 5, 0, -	13_2 if (x>b)	2, 3, 6, 5,10, 1, T
11. z=10;	12_2 for (i=...) {	2, 3,10,5, 5, 1, -	14_1 G (z, y);	2, 3, 6, 5,10, 1, -
12. for (i=0;i<2; i++) {	13_2 if (x>b)	2, 3,10,5, 5, 1, T	5_1 z=z-y;	2, 3, 6, 5, 5, 1, -
13. if (x>b)	14_2 G (z, y);	2, 3,10,5, 5, 1, -	15_2 x=x+4;	2, 3,10, 5, 5, 1, -
14. z=G (z, y);	5_2 z=z-y;	2, 3,10,5, 0, 1, -	18_1 print(z);	2, 3,10, 5, 5, 2, -
15. x=x+4;	15_2 x=x+4;	2, 3,14,5, $\boxed{0}$ 1, -		
16. }	18_1 print(z);	2, 3,14,5, 0, 2, -		
18. print(z);				

Fig. 1. Misaligned failure inducing states. The ideal execution is from the corrected program with the same input. The dashed lines represent execution alignments. Boxes denote failure inducing states computed through comparisons. Note that function invocations in the traces are transformed to better reflect their semantics, e.g., variables such as v and w are omitted for brevity in the state columns.

In our prior work [1], we proposed using execution indexing [4] to align executions. This uses a tree, called the *index tree*, that represents the hierarchical structure of an execution so that executions can be aligned by aligning their trees. Let us first focus on the tree for the failing run presented on the left in

Fig. 2. The root node denotes the entire execution, which is the body of the main function. The main body comprises the execution of statements 8_1, 9_1, 10_1, 11_1, 12_1, and 18_1, which are the nodes at the first level. Observe that statement executions 9_1 and 12_1 have substructures, so substructure nodes are introduced at the second level. The procedure continues until all hierarchical substructures are exposed. Note that the second iteration of loop 12, denoted by the subtree rooted at 12_2, is considered as part of the first iteration, denoted by node 12_1. The reason is that the execution of the second iteration is determined by the fact that the first iteration gets executed. Similarly, the index tree of the correct run can be constructed. The two executions are aligned by aligning their index trees. As a result, 14_1 and 5_1 in the left tree do not align with any nodes in the right tree, and 14_2 and 5_2 in the left tree align with 14_1 and 5_1 in the right tree. Thus, $FIS(5_2) = \{z \mapsto (0, 5_2)\}$, with the state of z being a pair comprising its value and the definition point of the value, because it is the minimal subset of the faulty variables at 5_2 that causes the FIS of its next step, 15_2. Recall that $FIS(15_2) = \{z \mapsto (0, 5_2)\}$. $FIS(14_2) = \{z \mapsto (5, 5_1)\}$, as it induces $FIS(5_2)$. The FISs of all comparable execution points are annotated on the nodes in the left tree. The definition points in these FISs constitute the causal path as highlighted in the left tree, which is $10_1 \to 13_1 \to 5_1 \to 5_2 \to 18_1$. One can see it starts with the root cause and clearly explains how the fault leads to the failure. In comparison, the dynamic slice [6] of the failure point 18_1 contains all the executed statements, including the causal path.
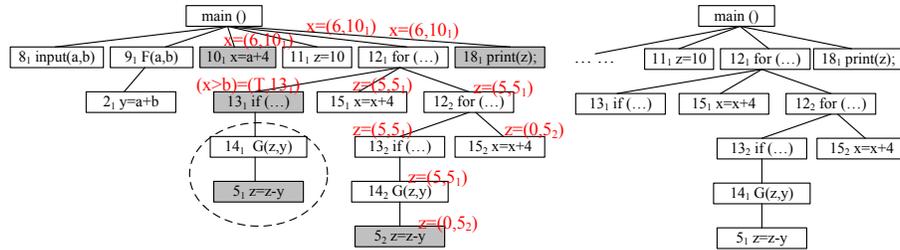

Fig. 2. Index trees for the two executions in Fig. 1. Execution alignment is achieved by aligning the two trees. The circled portion does not align with any part in the other tree. FISs for aligned nodes are annotated. The state of a variable is a pair *(value, definition point)*. The causal path is highlighted in the index tree of the failing run.

So far, we have discussed how to compute the causal path in the ideal case where the corrected program is available. In realistic debugging, however, only the buggy program is available. It was proposed in [3, 2] to use a similar but passing run of the buggy program as the reference run to perform the comparison. Unfortunately, since the two executions are derived from two different inputs, the semantic differences often significantly compromise the resulting causal path. Consider the case in Fig. 3. The failing run is the same as before, but since the corrected program is not available, a similar but passing run of the buggy pro-

gram is used, derived from the input *a=2* and *b=1*. Note that although the run on the right is from the buggy program, it produces the expected output, i.e., the same output as that produced by the corrected program, because whether x has the value of 6 or 2 at 10_1 does not affect the final output. The two executions have identical control flows and their inputs differ by only one value. The two executions can be trivially aligned. Comparing the states at the aligned steps as mentioned earlier produces the FISs boxed in the figure. Notice that $FIS(15_2) = \{z \mapsto (0, 5_2)\}$, as it is the minimal state difference that induces the failure. $FIS(14_2) = \{y \mapsto (5, 2_1), z \mapsto (5, 5_1)\}$ because it is the minimal state difference that induces $FIS(5_2) = \{z \mapsto (0, 5_2)\}$, even though we know y has a benign value. The definition points of the faulty states at each step constitute the causal path as highlighted on the left. Observe that although it has causality between steps, the path does not explain the failure but rather the difference between the two executions. Particularly, $\{b \mapsto (3, 8_1)\}$ is computed for $FIS(8_1)$ although b has a completely benign value at 8_1. Similarly, $FIS(2_1) = \{y \mapsto (5, 2_1)\}$ is due to the semantic differences between the two executions. Even worse, x is not part of $FIS(10_1)$ as it has the same value in both executions. In other words, the real faulty state is mistakenly considered as being benign.

Failing Execution (a=2, b=3)		A Similar but Passing Execution (a=2, b=1)	
Trace	State	Trace	State
	a, b, x, y, z, i, (x>b)		**a, b, x, y, z, i, (x>b)**
8_1 input (a,b);	2, 3, 0, 0, 0, 0, -	8_1 input (a,b);	2, 1, 0, 0, 0, 0, -
9_1 F (a,b);	2, 3, 0, 0, 0, 0, -	9_1 F (a,b);	2, 1, 0, 0, 0, 0, -
2_1 y=a+b;	2, 3, 0, 5, 0, 0, -	2_1 y=a+b;	2, 1, 0, 3, 0, 0, -
10_1 x=a+4;	2, 3, 6, 5, 0, 0, -	10_1 x=a+4;	2, 1, 6, 3, 0, 0, -
11_1 z=10;	2, 3, 6, 5, 10, 0, -	11_1 z=10;	2, 1, 6, 3, 10, 0, -
12_1 for (i=…) {	2, 3, 6, 5, 10, 0, -	12_1 for (i=…) {	2, 1, 6, 3, 10, 0, -
13_1 if (x>b)	2, 3, 6, 5, 10, 0,T	13_1 if (x>b)	2, 1, 6, 3, 10, 0,T
14_1 G (z, y);	2, 3, 6, 5, 10, 0, -	14_1 G (z, y);	2, 1, 6, 3, 10, 0, -
5_1 z=z-y;	2, 3, 6, 5, 5, 0, -	5_1 z=z-y;	2, 1, 6, 3, 7, 0, -
15_1 x=x+4;	2, 3,10, 5, 5, 0, -	15_1 x=x+4;	2, 1,10, 3, 7, 0, -
12_2 for (i=…) {	2, 3,10, 5, 5, 1, -	12_2 for (i=…) {	2, 1,10, 3, 7, 1, -
13_2 if (x>b)	2, 3,10, 5, 5, 1, T	13_2 if (x>b)	2, 1,10, 3, 7, 1, T
14_2 G (z, y);	2, 3,10, 5, 5, 1, -	14_2 G (z, y);	2, 1,10, 3, 7, 1, -
5_2 z=z-y;	2, 3,10, 5, 0, 1, -	5_2 z=z-y;	2, 1,10, 3, 4, 1, -
15_2 x=x+4;	2, 3,14, 5, 0, 1, -	15_2 x=x+4;	2, 1,14, 3, 4, 1, -
18_1 print(z);	2, 3,14, 5, 0, 2, -	18_1 print(z);	2, 1,14, 3, 4, 2, -

Fig. 3. Comparing the failing run with a similar but passing run with different input.

In our technique, we first construct a dynamic patch to correct the failing execution and then use the patched execution as the reference run for comparison [1]. A failing execution is *patched* if mutating part of its state at one or multiple execution points leads to the correct output. Since the patched execution is derived from the same input, it precludes state differences caused by the input differences. Predicate switching [7] is our prior work on patching a failing run. It works by systematically changing the branch outcome of a predicate instance and then observing if the mutated execution produces the expected

output. It was used as a fault localization technique because, if such a predicate instance exists, called the *critical predicate*, it discloses a wealth of information on the fault. Our study [1] showed that 80% of all the failing test cases in the SIR [5] suite and 8 out of 12 real bugs collected from internet can be patched by predicate switching. A patched run serves as an approximation of the ideal run to carry out execution comparison. For instance, if the predicate instance 13_1 of the failing run in Fig. 1 is switched so that 14_1 and 5_1 are not executed, the resulting z value at 18_1 becomes the desired 5. The patched run is very similar to the *ideal* run on the right of Fig. 1. The only difference is that variable x has different values from 10_1 to 13_1 in the two respective runs. The causal path is computed as $13_1 \rightarrow 5_1 \rightarrow 5_2 \rightarrow 18_1$, which captures most of the ideal causal path. Note that the root cause 10_1 is not caught, as the patched run has identical state as the failing run till the switched predicate instance 13_1. Although the root cause is not captured in this example, it can often be captured by our technique if the switched predicate determines if the root cause gets executed. Our prior study [1] shows that about 45% of the causal paths computed by our technique capture the root cause. Moreover, we argue that presenting the causal path is more informative than pointing at root cause candidates. It is reported in [8] that requirement bugs are the most frequently occurring kind of bug in the field, which often do not have a single or a small set of statements to be blamed as the root cause. For such cases, understanding failure causality is preferable.

3 Algorithms

The major contribution of this paper is a detailed study of two algorithms for the aforementioned causal path computation. The two algorithms produce the same causal paths but achieve efficiency with different approaches. A naïve approach is to first align the two executions by aligning their index trees, then compute the FISs for each aligned step backwards, starting from the failure point. Our experience shows that such an algorithm is extremely expensive due to the large number of aligned execution points. For the failures collected from Linux utilities, listed in Section 4, the algorithm failed to terminate after 8 hours of computation. Note that all these algorithms require the same basic FIS computation that compares states of two aligned steps in the two respective executions and minimizes the state differences using the delta debugging algorithm [9, 3, 2].

3.1 A Hierarchical Algorithm

We have proven in [1] that FISs have a stability property, which states that if *the FISs computed at any two aligned points are the same, there is no need to compute FISs between these two points because they will be identical.* For example, in Fig. 3, as the two aligned steps 5_1 and 14_2 in the failing run have the same FIS, i.e., $\{y \mapsto (5, 2_1), z \mapsto (5, 5_1)\}$, all aligned steps in between have the same FIS, too. Formally, this property requires a more advanced notion of FIS that captures the definition points and values from both the faulty and correct

executions, but we elide these details in presentation for simplicity. Based on this property, a hierarchical algorithm can be designed to compute FISs in a demand-driven fashion. The idea is to carry out state comparison and causality testing top-down along the index tree of the failing execution until the right granularity is reached. Each FIS is computed to induce the successive one except for the last FIS, which induces the observed failure.

Algorithm 1 Hierarchical computation for causal paths of failures.

Primitives:
- ALIGNEDCHILDREN()- Finds the children that align with some nodes in the reference run.
- FIS()- Computes the FIS of the given aligned node.

Important Variables:
- `target` - The FIS to be induced when computing the preceding FIS.
- `defs` - The sequence of definition points that constitute the causal path.
- `sets`- A set of FISs.

CALCULATECAUSALPATH()
1   target ← failure
2   defs ← {target}
3   (sets,target) ← SETSINREGION(executionRoot,target)
4   defs ← defs $\cup(\cup_{\text{fis}\in\text{sets}})$
5   **return** TEMPORALSORT(defs)

SETSINREGION(node, target)
INPUT:*A node in the index tree and the first FIS that must be induced.*
OUTPUT:*The set of FISs in the region and the single FIS that induces them.*
1   sets ← ∅
2   kids ← ALIGNEDCHILDREN(node)\{children executing with or after target}.
3   **while** |kids| > 0 **do**
4   **while** |kids| > 1 **do**
5   mid ← |kids| / 2
6   newFIS ← FIS(kids$_{mid}$) inducing target
7   **if** newFIS = target **then**
8   kids ← kids$_{0...mid-1}$
9   **else**
10   kids ← kids$_{mid...|kids|-1}$
11   **if** FIS(kids$_0$)≠target **then**
12   (subsets,target) = SETSINREGION(kids$_0$, target)
13   sets ← sets ∪ subsets
14   kids ← ALIGNEDCHILDREN(node)\{children executing with or after kids$_0$}.
15 newFIS ← FIS(node) inducing target
16 **return** (sets ∪ newFIS, newFIS)

The algorithm is shown in Algorithm 1. Using the failing execution and the reference execution, CALCULATECAUSALPATH() generates the complete causal path. This procedure first initializes variables `target`, which is the FIS or the failure to be induced by the next computed FIS, and `defs`, which stores the sequence of definitions constituting the causal path. It then calls SETSINREGION() with the root of the index tree and `target` to compute the set of FISs, stored in `sets`, in a top-down manner. Once all FISs have been aggregated, the definitions they contain are sorted by their temporal position and returned at line 5. This sequence of definitions comprises the complete causal path.

SETSINREGION() generates the FISs for the portion of the indexing tree rooted at `node` such that the temporally last generated FIS induces the FIS

`target`. At line 2, the algorithm extracts the ordered list of all the child nodes that have alignments in the reference execution and stores them in `kids`, excluding those executed after or with `target`. If there are no such children, the function skips the loop in lines 3-14 and computes the FIS for the `node` that induces `target`. The loop computes the set of FISs of the subtree. The inner loop in lines 4-10 performs a classic binary search on `kids` list to locate the first child from the end that has an FIS different from `target`. Observe that if the FIS of the midpoint is identical to `target`, as checked at line 7, the algorithm safely skips computing FISs for the right half of the `kids` list and its subtrees. If such a child is found, the function recursively calls itself to compute the FISs in the subtree rooted at the child on line 12. On line 14, the `kids` list updates to reflect a new FIS, so the next round of binary search will be performed on a reduced list. The computation terminates if the list becomes empty.

Consider the example in Fig. 2. The computation starts from the top of the tree on the left, and the failure at 18_1, i.e., $\{z \mapsto (0, 5_2)\}$, is the `target` to induce. In the first invocation of SETSINREGION(), the `kids` list is initialized to contain $\{8_1, 9_1, 10_1, 11_1, 12_1\}$. The binary search in lines 4-10 identifies the first child with an FIS different than `target` to be 12_1. It recursively calls itself on 12_1 to compute the FISs in the subtree rooted at 12_1. This time, the `kids` list is initialized to have $\{13_1, 15_1, 12_2\}$. The algorithm descends along 12_2, 13_2, and then 14_2 because their FISs are different than the failure until the closest different FIS inducing the failure, namely 14_2, is identified. The `target` is updated to be 14_2. At some point, the recursive call for 12_2 returns with $FIS(12_2) = \{z \mapsto (5, 5_1)\}$ being the `target`. The `kids` list is updated to $\{13_1, 15_1\}$ at line 14 to start a new round of the binary search. This time, the search identifies $FIS(15_1) = FIS(12_2)$, so if 15_1 were the root for a subtree, the algorithm would not descend into the subtree. Computation over the remainder of the tree can be similarly derived.

3.2 A Shortcutting Algorithm

We introduce here another algorithm that exploits the stability property in a different way. That is, given an FIS to induce, we try to identify the earliest aligned point that is likely to have the same FIS and jump directly to that point without computing any FISs in between. The intuition is that the earliest point that has the same FIS is very likely *the last definition point, before the current point, that occurs in any of the previously computed FISs* because all faulty values in previously computed FISs remain intact between their definition points and FISs. For example, in Fig. 3, at the aligned step 14_2, the set of definitions in all the previously computed FISs is $\{2_1, 5_1, 5_2\}$. Recall that causal path computation proceeds backwards, starting from the failure, with the first FIS inducing the failure and the remaining FISs each inducing the successive FIS. The last definition point that happens before 14_2 is 5_1. Observe that $FIS(5_1)$ is still $\{y \mapsto (5, 2_1),\ z \mapsto (5, 5_1)\}$. That implies we are taking the shortcut, jumping directly from 14_2 to 5_1. Further note that the FIS immediately before is $\{y \mapsto (5, 2_1)\}$. This reflects the change in FIS that the definition at 5_1 causes. By taking dependence shortcuts, fewer searches for computing an FIS are needed.

Pseudocode for this approach is presented in Algorithm 2. The main procedure CALCULATECAUSALPATH() derives the causal path for the failing execution, computing backwards starting from the failure. In lines 4-17, the algorithm traverses backwards by taking shortcuts if possible, accumulating FISs in `defs` along the way, until there is no more relevant state, as checked at 4. On line 5, the last definition that was previously caught in the causal path and occurred before the current computation point is stored in `closest`. In lines 6-9, the algorithm handles cases in which no shortcut is available as the captured definition points happen after or at the current traversal point. The algorithm conservatively moves one step backwards. Lines 10-17 handle cases in which a shortcut is possible. Line 13 checks if taking the shortcut is valid by comparing the FIS of `closest` with `target`. If they are identical, the shortcut is valid and then at line 17, the algorithm updates `currentNode`. Note, however, the shortcut may not always be valid, i.e., a different FIS may be computed at the destination of the shortcut. For example, in Fig. 4 the failing run is different from the reference run because it omits the execution of the true branch. Thus, variable x is not updated. Assume $FIS(7_1)$ is computed as $\{x \rightarrow (2, 2_1)\}$. If the algorithm takes the shortcut to 2_1, the FIS at 2_1 is empty as all variables have the right value. More important, following the shortcut misses the real FIS alteration point 3_1. This results from the failing run omitting execution that it should have gone through. Note that similar effects can be observed if program state is updated by the OS and thus not visible to our analysis. Fortunately, such cases are reflected in the FIS at the shortcut being different from `target`. If they occur, the algorithm falls back to a revised hierarchical search at line 14.

Code	Failing Run			Ref. Run		
		State		Trace	State	
		x	p		x	p
1. ...						
2. x=2;	1_1 ...			1_1 ...		
3. if (p) {	2_1 x=2;	$(2, 2_1)$	-	2_1 x=2;	$(2, 2_1)$	-
4. ...	3_1 if (p)	$(2, 2_1)$	(F,3_1)	3_1 if (p) {	$(2, 2_1)$	(T,3_1)
5. x= x-1;				4_1 ...		
6. }				5_1 x=x-1;		
7. ...;	7_1 ...;	$(2, 2_1)$	-	7_1 ...;	$(1, 5_1)$	-

Fig. 4. Taking the shortcut is not successful due to execution omission in the failing run. The horizontal lines denote execution alignment.

CLOSESTADAPTIVE() performs a revised hierarchical derivation of a single FIS given the `target`, i.e., the FIS to be induced, and the bound on how early the FIS transition could occur. Line 1 finds the common ancestor in the index tree of both the early bound and the index at which the target FIS was derived. Intuitively, execution within the provided bounds must be in the subtree rooted at the ancestor. Lines 2-14 of CLOSESTADAPTIVE() simply perform a hierarchical binary search for the resulting FIS. The procedure is very similar to SETSINREGION in Algorithm 1. At line 3, the children of the `start` node are retrieved. Those that happen before `closest` and after or at the point where

target is computed are filtered out. The loop in lines 4-10 is a classic binary search that looks for the first kid with an FIS different from target, and then the algorithm traverses the index tree one level down to the kid at line 14.

Algorithm 2 Shortcutting computation for causal paths of failures.

Important Variables:
- target - The FIS to be induced when computing the preceding FIS.
- defs - The sequence of definition points that constitute the causal path.
- currentNode - The current FIS computation point.

CALCULATECAUSALPATH()
1   currentNode ← the failure point
2   target ← failure
3   defs ← {target}
4   **while** target ≠ ∅ **do**
5   closest ← the temporally last definition in defs that happens before currentNode
6   **if** closest = ⊥ **then**
7   currentNode ← the aligned point immediately preceding currentNode.
8   target ← FIS(currentNode)
9   defs ← defs ∪ target
10   **else**
11   **if** closest is not aligned **then**
12   closest ← the aligned point immediately preceding closest.
13   **if** FIS(closest) ≠ target **then**
14   target ← CLOSESTADAPTIVE(target, closest)
15   defs ← defs ∪ target
16   **else**
17   currentnode ← closest
18   **return** TEMPORALSORT(defs)

CLOSESTADAPTIVE(target, closest)
INPUT: *A target FIS to induce and a bound on the earliest point it may be induced.*
OUTPUT: *The FIS that immediately precedes the target in the sequence of FISs.*
1   start ← the common ancestor in the indexing tree of closest and the index of target.
2   **loop**
3   kids ← ALIGNEDCHILDREN(start)\{children before closest, after target, and target itself}.
4   **while** |kids| > 1 **do**
5   mid ← |kids| / 2
6   newFIS ← FIS($kids_{mid}$) inducing target
7   **if** newFIS = target **then**
8   kids ← $kids_{0...mid-1}$
9   **else**
10   kids ← $kids_{mid...|kids|-1}$
11   **if** kids = ∅ **then**
12   currentnode ← start
13   **return** FIS(start) inducing target
14   start ← $kids_0$

Consider the example in Fig. 2. To start, target=failure=$\{z \mapsto (0, 5_2)\}$, so the shortcut jumps to 5_2. The shortcut is valid, as $FIS(15_2)$=target. In the next round of the main loop in CACULATECAUSALPATH(), the algorithm traverses one step backwards since no shortcut is available because all captured definitions happen after or at 5_2. The next FIS computation is for 14_2, which has the result $\{z \mapsto (5, 5_1)\}$. Due to the newly caught definition 5_1, a shortcut is available to reach 5_1. However, 5_1 is not aligned and thus its immediate aligned predecessor 13_1 is used to compute the FIS instead, as in line 12 of CACULATE-CAUSALPATH(). The resulting $FIS(13_1) = \{(x > b) \mapsto (T, 13_1)\}$ differs from

target$=FIS(14_2)$. That is, the shortcut is not valid and the algorithm must fall back and call CLOSESTADAPTIVE(). The binary search eventually identifies 13_1 as the closest point at which a different FIS is computed that induces $FIS(14_2)$. Now, since there is not a captured definition occurring before 13_1, the algorithm moves one step backwards and computes $FIS(12_1) = \{x \mapsto (6, 10_1)\}$. A shortcut is available leading to the root cause 10_1.

4 Evaluation

In order to evaluate the presented algorithms, we employed them against several real world bugs found in the GNU utilities `grep`, `gzip`, `bc`, `find`, `diff`, and `tar`. The debugging infrastructure comprises source to source transformation via CIL, Python, and the public Python and GDB infrastructure from [3]. The two algorithms were implemented with CIL and Python. The tests were run on a 2GHz dual core machine with 2GB of RAM. For each analyzed bug, Table 1 presents the program and version the bug applies to, a bug report if available, the number of definitions the causal path comprises, and the time in seconds required for each algorithm to derive the causal path. *Time 1* shows to the time taken by Algorithm 1 and *Time 2* shows the time taken by Algorithm 2.

Table 1. Examined bugs and their causal path properties. Time n is the time in seconds taken to derive the causal path using Algorithm n. 'Bug' is the Internet address of a bug report, if applicable.

Program	Version	Bug	Path Length	Time 1	Time 2
bc	1.06	bugs.gentoo.org/51525	2	1130	135
diff	2.8	....gnu....utils/2002-12/msg00067.html	8	2320	368
find	4.3.0	savannah.gnu.org/bugs/?18222	5	2906	48
grep	2.5.1	savannah.gnu.org/bugs/?11579	5	1220	159
grep	2.5.1	savannah.gnu.org/bugs/?9519	7	>4 hours	250
grep	2.5.1	savannah.gnu.org/bugs/?13920	7	6865	217
grep	2.5.1	savannah.gnu.org/bugs/?9768	5	6167	221
grep	2.5.1	savannah.gnu.org/bugs/?19491	9	>4 hours	244
grep	2.5.3	savannah.gnu.org/bugs/?15620	4	>4 hours	56
grep	2.5.3	`-h -H` with a single file	4	>4 hours	55
gzip	1.3.9	....gnu....gzip/2007-05/msg00003.html	15	7583	651
tar	1.13.25	....gmane....comp.gnu.tar.bugs/491	7	8823	531

Observe that the time taken by Algorithm 2, using shortcutting, is consistently and substantially less than the time required by Algorithm 1, 7% as long or less on average. The most significant factors in the runtime of the algorithm are the number of causality tests and the size and complexity of the program states that must be analyzed during causality tests. Causality testing is part of the FIS computation. It determines if a subset of state differences induce the

successive FIS. It is done through re-executing the program with the state differences applied. The efficiency difference in Table 1 results predominantly from decreasing the number of causality tests via shortcutting. In practice, execution omission and unobserved state force the shortcutting approach to degenerate into a hierarchical binary search for some individual elements of the path, but most shortcutting efforts are successful. Both algorithms are significantly faster than a naïve algorithm that computes FISs linearly by traversing backwards step by step. Note that all these algorithms compute the same causal path.

In the second experiment, we evaluate the effectiveness of our technique on a set of real bugs from Linux utility programs. We collected these bugs by looking into their CVS repositories and on-line bug reports. Some of these cases are explained in detail below.

Grep. Version 2.5.3 of the `grep` regular expression matching utility incorrectly handles command-line options `-h` and `-H`, which respectively disable and enable printing out the filename of a file containing a matched expression. If both options are given, only the last should be obeyed, but both options are independently enabled, yielding inconsistent results. Interestingly, there are multiple ways that this fault can manifest, but they have the same causal path, which identifies them as stemming from the same fault. If options `-H -h` are used when searching multiple files, file names prefix all resulting output with matches in those files, but they should not. If `-H -h` are used when searching one file containing short lines of text, some of the resulting output lines have the filename prefix, while others do not. It is not immediately apparent that the same fault affects both executions. The causal paths and faulty code for these executions, however, are the same, as shown in Fig. 5(a). The switched predicate on line 5 becomes false, preventing the flag for printing filename prefixes from becoming enabled. The resulting causal path shows that in the failing runs, the predicate on line 5 enables the flag `out_file` for printing filename prefixes on line 6. Much later in the execution, this causes the predicate on line 7 to be true, which then results in failure when the filenames are printed at 8. The switched predicate and causal path reveal that the predicate should not evaluate to true, but the `-H` command forces it to via the variable `with_names`, as it is mistakenly not disabled by the later option `-h`. The applied fix was then to disable the opposing commands on lines 2 and 3.

Find. Find is a tool that locates all files matching provided criteria and also performs an action at all such files. Version 4.3.0 contains a bug when multiple directories to search are specified. If the given action is `-printf '%H %P\n'`, which prints out the specified directory name before each file found in the directory, then every directory name printed is no longer than the first. For instance, if the directories `dir1` and `directory` are specified, the contents of `dir1` will be printed with the prefix 'dir1', and the contents of `directory` will have the prefix 'dire'. The causal path is presented in Fig. 5 (b). At the first step of the causal path, the variable `s.starting_length` is set to 4, recording the length of the first specified directory. The critical predicate is on line 1, which corresponds to the second invocation of the function `consider_visiting()`, handling

the second specified directory. In the failing run, it evaluates to false, so when it gets switched, `s.starting_length` is updated to 9 and thus leads to the correct output. The causal path also captures that `s.starting_length` is used at line 6 to cut off the second directory name and eventually produce the wrong output. The assignment to `cc` at line 5 is in the causal path, even though it may not seem needed for failure induction, because the wrong directory name 'dire' was printed multiple times, and the assignment at line 5 is critical to continually printing the wrong directory name. This case also shows how a causal path does not necessarily start with the critical predicate. Namely, the definition at line 2 is the first step of the causal path because it is in $FIS(5)$ and $FIS(6)$.

Code Snippet:
```
main(argc,argv):
1  switch (options) {
2    case 'H': with_names = true;break;
3    case 'h': no_names = true;break;
4
5  if ((num_files>1 && !no_names) ||
         with_names)
6      out_file = true;

print_line_head(beg, lim, sep):
7  if (out_file)
8      print_filename();
```

Causal Path:

At 5, (. . . || with_names) is true
At 6, out_file is given true
At 7, (out_file) is true
Thus the filename is printed at 8.

(a)

Code Snippet:
```
consider_visiting(p,ent):
1  if (0 == s.starting_length)
2      s.starting_length = ent->fts_pathlen;

pred_fprintf(pathname,stat_buf,pred_ptr):
3  switch (kind & 0xff) {
4    case 'H':
5      cc = pathname[s.starting_length];
6      pathname[s.starting_length] = '\0';
7      printf(pathname);
8      pathname[s.starting_length] = cc;
9      break;
```

Causal Path:

At 2, s.starting_length is given 4
At 1, (0 == s.starting_length) is false
At 5, cc is given 'c'
At 6, pathname is given "dire"
So "dire" is printed at 7.
... ...
"dire" is printed at 7 again.

(b)

Fig. 5. Causal paths for (a) grep and (b) find

Diff. The `diff` tool compares two files or the contents of two directories and reports differences. In version 2.8, an option to ignore the case within filenames works incorrectly when comparing directory contents. Thus, if one directory contains the file `bar` while another contains `BAR`, comparing the directories with the `--ignore-file-name-case` option should print nothing, but it reports the above files as being different. The causal path and code in Fig. 6 for this bug show that this is due to an incorrect name comparison algorithm. When comparing directories, the list of files from one directory is compared to the list in another using the `diff_dirs` function. This uses `compare_names()` to compare individual files. The critical predicate on line 2 shows that when two file names are case-insensitively equal, that equality is immediately disregarded because `(r)` is false. Thus, the `compare_names()` function continues and returns that the names are case-sensitively unequal via `-7` at line 4 and into `order` at 6. This is used to determine that one of the files is unique when `(order<0)` is true at line 8 causes `*fname1` to be zeroed out at 9. Thus, when the `compare_files()` function is called to show the results, the argument `name1` is given 0, and `(!(name0 && name1))` evaluates to true, forcing one of the equivalent filenames, `name0`, to be printed as a difference. The causal path indicates that the right

patch should be to change the algorithm in `compare_names()` to make the comparison at line 4 case insensitive when the option is set.

Code Snippet:

```
compare_names(name1,name2):
1  r = strcasecmp(name1,name2);
2  if (r)
3      return r
4  return strcoll(name1,name2);

diff_dirs(cmp,handle_file):
5  while (fname0 || fname1) {
6      order = compare_names(*fname0,*fname1);
7 ...
8      if (order < 0)
9          *fname1 = 0;
10     v1 = compare_files(cmp,*fname0,*fname1);

compare_files(parent,name0,name1):
11 if (!(name0 && name1))
12     print(name0 == 0 ? name1 : name0);
```

Causal Path:

At 2, (r) is false
At 4, compare_names returns -7
At 6, order is given -7
At 8, (order < 0) is true
At 9, *fname1 is given 0
At 10, name1 is given 0
At 11, (!(name0 && name1)) is true
So the filename is shown at 12

Fig. 6. Causal path for diff

5 Related Work

Delta Debugging. The work most relevant to ours is that by Zeller et al. [9, 2, 3]. The project in [2] is the first one to propose comparing two similar executions using delta debugging [9] to compute cause-effect chains, which is a concept similar to causal paths of failures. Later in [3], the technique is extended to link cause transitions to a faulty statement. Compared to these works, we make significant progress on the following: we identify execution indexing as a key technique, use a patched execution instead of a different execution to reduce noise from semantic differences, and develop efficient algorithms. Our prior evaluation [1] using the SIR suite [5] showed that our technique is superior.
Fault Localization. Fault localization computes fault candidates by looking at many executions, both passing and failing, as exemplified by [10, 11, 12, 13, 14, 15, 16, 17]. Details of these techniques cannot be presented due to space limits. Compared to our technique, fault localization techniques are less effective in explaining failures. They produce a ranked candidate set, usually containing static statements. Reasoning about the candidates and the failure often falls onto the programmer.
Dynamic Slicing. Dynamic slicing was introduced as an aid to debugging [6]. Compared to fault localization, slicing features the capability of capturing causality through program dependencies. However, slicing tends to produce fat slices containing not only the failure inducing dependencies but also benign dependencies. Although various techniques have been proposed to prune dynamic slices [18, 19], without using a reference execution to exclude benign chains, inspecting pruned slices still requires non-trivial human effort. Dicing [20] aggregates slices from multiple executions. However, the simple set manipulations in dicing undermine causality and make resulting slices hard to understand. Furthermore, it does not handle cases in which a faulty statement occurs in both the benign and faulty slices. In comparison, our work does not rely strictly on program

dependence but rather on semantic causality. The use of a reference execution effectively excludes benign state.

References

[1] Sumner, W.N., Zhang, X.,"Automatic failure inducing chain computation through aligned execution comparison," Tech. Rep. 08-023, Purdue University, http://www.cs.purdue.edu/homes/wsumner/CSD_TR_08-023.pdf (2008)

[2] Zeller, A., "Isolating cause-effect chains from computer programs," in *FSE* (2002)

[3] Cleve, H., Zeller, A., "Locating causes of program failures," in *ICSE* (2005)

[4] Xin, B., Sumner, W.N., Zhang, X., "Efficient program execution indexing," in *PLDI* (2008)

[5] Do, H., Elbaum, S., Rothermel, G., "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.," *Empirical Software Engineering: An International Journal*, vol. 10, no. 4.

[6] Korel, B., Laski, J.,"Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3 (1988)

[7] Zhang, X., Gupta, N., Gupta, R., "Locating faults through automated predicate switching," in *ICSE* (2006)

[8] Jackson, D., Thomas, M., Millett, L.I., *Software for Dependable Systems:Sufficient Evidence?* THE NATIONAL ACADEMIES PRESS.

[9] Zeller, A., Hildebrandt, R., "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2 (2002)

[10] Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., Yi, L., "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, vol. 10, no. 3.

[11] Jones, J.A., Harrold, M.J., Stasko, J., "Visualization of test information to assist fault localization," in *ICSE* (2002)

[12] Renieris, M., Reiss, S., "Fault localization with nearest neighbor queries," in *ASE* (2003)

[13] Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I., "Bug isolation via remote program sampling," in *PLDI* (2003)

[14] Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S., "Sober: statistical model-based bug localization," in *FSE* (2005)

[15] Brun, Y., Ernst, M.D., "Finding latent code errors via machine learning over program executions," in *ICSE* (2004)

[16] Chesley, O.C., Ren, X., Ryder, B.G., Tip, F., "Crisp–a fault localization tool for java programs," in *ICSE* (2007)

[17] Wang, T., Roychoudhury, A., "Automated path generation for software fault localization," in *ASE* (2005)

[18] Gupta, N., He, H., Zhang, X., Gupta, R.,"Locating faulty code using failure-inducing chops," in *ASE* (2005)

[19] Zhang, X., Gupta, N., Gupta, R., "Pruning dynamic slices with confidence," *SIGPLAN Not.*, vol. 41, no. 6 (2006)

[20] Chen, T.Y., Cheung, Y.Y., "Dynamic program dicing," in *ICSM* (1993)