Explaining Software Failures by Cascade Fault Localization

QIUPING YI, Chinese Academy of Sciences
ZIJIANG YANG, Western Michigan University
JIAN LIU and CHEN ZHAO, Chinese Academy of Sciences
CHAO WANG, Virginia Polytechnic Institute and State University

During software debugging, a significant amount of effort is required for programmers to identify the root cause of a manifested failure. In this article, we propose a cascade fault localization method to help speed up this labor-intensive process via a combination of weakest precondition computation and constraint solving. Our approach produces a cause tree, where each node is a potential cause of the failure and each edge represents a casual relationship between two causes. There are two main contributions of this article that differentiate our approach from existing methods. First, our method systematically computes all potential causes of a failure and augments each cause with a proper context for ease of comprehension by the user. Second, our method organizes the potential causes in a tree structure to enable *on-the-fly* pruning based on domain knowledge and feedback from the user. We have implemented our new method in a software tool called CaFL, which builds upon the LLVM compiler and KLEE symbolic virtual machine. We have conducted experiments on a large set of public benchmarks, including real applications from GNU Coreutils and Busybox. Our results show that in most cases the user has to examine only a small fraction of the execution trace before identifying the root cause of the failure.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Verification, Reliability, Algorithms

Additional Key Words and Phrases: fault localization, weakest precondition, constraint, satisfiability modulo theory (SMT)

ACM Reference Format:
Qiuping Yi, Zijiang Yang, Jian Liu, Chen Zhao, and Chao Wang. 2015. Explaining software failures by cascade fault localization. ACM Trans. Des. Autom. Electron. Syst. 20, 3, Article 41 (June 2015), 28 pages.
DOI: http://dx.doi.org/10.1145/2738038

1. INTRODUCTION

Testing and debugging are considered the most expensive phase in the entire software development cycle [Beizer 1990]. One of the main reasons for such high cost is that fault localization, the process of tracing propagation of faults and identifying

This work was supported in part by the National Science Foundation of China (NSFC) under grant 61472318, the National Science and Technology Major Project of China under grant 2012ZX01039-004, and the National Science Foundation (NSF) under grants CCF-1149454, CCF-1500365, and CCF-1500024. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.
Authors' addresses: Q. Yi, National Engineering Research Center for Fundamental Software, Institute of Software, and University of Chinese Academy of Sciences, Beijing, China; Z. Yang, Department of Computer Science, Western Michigan University, Kalamazoo, MI; School of Computer Science and Engineering, Xi'an University of Technology, Xi'an, China; J. Liu, C. Zhao, National Engineering Research Center for Fundamental Software, Institute of Software, Beijing, China; Chao Wang, Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA. Corresponding authors: Zijiang Yang; email: zijiang.yang@wmich.edu, J. Liu; email: liujian@iscas.ac.cn.
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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2015 ACM 1084-4309/2015/06-ART41 $15.00
DOI: http://dx.doi.org/10.1145/2738038

the location of erroneous program statements, is labor intensive and time consuming. Although there exists a large body of work in the software engineering literature on developing automated methods for fault localization, many of these existing methods require a comprehensive test suite to provide sufficiently many passing and failing executions [Groce and Visser 2003; Ball et al. 2003; Renieris and Reiss 2003; Groce et al. 2006]. In this context, fault localization is conducted by comparing and contrasting these passing and failing executions. While they are useful in certain applications, such as version upgrade [Banerjee et al. 2010], such approaches do not match the common software development practice, where programmers tend to focus on a single faulty execution rather than accumulate a large number of executions for comparison. In addition, although it is possible to generate more executions from a faulty execution for the purpose of comparison, ensuring that these additional passing and failing executions are closely related to the same fault is a challenging task.

We believe that, in general, the process of identifying program statements responsible for a given failure cannot be fully automated. In other words, the user's involvement cannot be completely eliminated except for rare occasions, for instance, when a formal specification of the intended program behavior or a golden model exists. The reason is that there will always be more than one possible ways to make the failure manifested in a faulty execution trace *go away*—some are valid fixes but some are merely making the erroneous code unreachable under certain program inputs. For example, in the following code snippet `x=0; y=0; if(x>0) {y=1;} assert(y!=0)`, there are two potential *causes* of the assertion failure. One is `y=0` and the other is `if(x>=0)`, since we can avoid the failure either by changing `y=0` to `y=1`, or by changing `if(x>0)` to `if(x>=0)`. However, merely analyzing the code snippet itself is not enough to determine which repair is desired by the programmer. Without knowing the programmer's design intent, in general, it is not possible for a software tool to completely automate the debugging process.

Therefore, instead of proposing new heuristics to replace the user in the aforementioned decision making process, we focus on identifying all potential causes and organizing them in a way that facilitates comprehension and efficient pruning. Specifically, given the program source code and a single execution that leads to an observable failure, our method traverses the error trace backwardly and systematically computes all possible causes. Each cause is augmented with a proper context to help the user understand why it may be responsible for the failure. Furthermore, all causes are organized in a tree structure, where nodes represent the causes and edges represent their causal relationships. Nodes in this *cause tree* are computed *on demand* in a cascade fashion. If the user finds that a certain cause is suspicious and wants to investigate further, the tree structure allows her/him to easily navigate to related causes at a upper level.

We have implemented our method in a software tool called CaFL, which stands for *Cascade Fault Localization*. The tool builds upon the popular LLVM [Lattner 2002] compiler platform and the KLEE [Cadar et al. 2008] symbolic virtual machine. We use LLVM/KLEE as the front-end to handle real-world C/C++ applications and replay the faulty executions. Our method first converts a faulty execution into a set of logic formulas based on a weakest precondition computation, and then solves these formulas using an off-the-shelf satisfiability modulo theory (SMT) solver. Based on the unsatisfiability proofs generated by the SMT solver, our method will be able to obtain the potential causes, together with the causal relationships between these causes. We have conducted experiments on a large set of public benchmarks, including programs from the Siemens Suite, which have been widely used in previous software testing studies, as well as real applications from GNU Coreutils and Busybox.

Our method differs from the recent works on using constraint solving in fault localization, such as BugAssist [Jose and Majumdar 2011a, 2011b], error invariants [Ermis et al. 2012; Christ et al. 2013], and inductive interpolant labelings [Murali et al. 2014].

Although these existing methods share some elements of the underlying analysis methods with ours, such as the use of SAT/SMT solvers and unsatisfiability proofs, their focus is different. In particular, these existing methods focus on heuristically identifying the root cause of a given failure, that is, replacing the programmer with fully automated algorithms during the decision making, whereas our method focuses on computing all potential causes and highlighting their causal relationships, to help the user decide which is the root cause.

To sum up, we have made the following contribution.

- We propose a new method for computing all potential causes of a manifested failure and organizing these causes in a tree structure to facilitate efficient navigation and pruning by the user.
- We implement our method in a practical software tool, which builds on the popular LLVM compiler and KLEE symbolic virtual machine, to handle real C/C++ applications.
- We demonstrate the effectiveness of our method on a large set of public benchmarks, including programs from the Siemens suite and real applications from GNU Coreutils and Busybox.

The remainder of the article is organized as follows. First, we illustrate our method using a motivating example in Section 2. Then, we present the detailed algorithm of our cascade analysis in Section 3. This is followed by a description of several heuristic optimizations in Section 4. We present our experimental results in Section 5. We discuss the limitations of our approach in Section 6. We review related work in Section 7, and finally give our conclusions in Section 8.

2. MOTIVATING EXAMPLES

In this section we use a running example given in Figure 1 to provide a high-level description of our new method. In addition, we use several simpler examples to illustrate the key ideas in case readers are not familiar with concepts such as weakest precondition computation. Figure 1 shows a program that takes as input the lengths of three sides of a triangle in decreasing order. Depending on whether the triangle is equilateral, isosceles, right, or scalene, the program uses different methods to compute the area. Using automated code transformation, we insert a statement at Line 4 to explicitly assign symbolic arguments to the parameters. During the concrete execution, the arguments `t1, t2, t3` are replaced by their actual values passed to the function.

There is a defect at Line 9, where the condition `(a*a == b*b+c*c)` is accidentally written as `(a*a != b*b+c*c)`. Under the test input vector `⟨ t1≡6,t2≡5,t3≡4⟩`, the execution produces the following trace $\langle 2-7, 9-11, 13-14, 22\rangle$. The bug is manifested as an assertion failure at Line 22, where `ORACLE` represents the exact correct value (about 9.92) for the variable `area`. Ideally, a fault localization tool should be able to inform the programmer that the defect at Line 9 causes the assertion failure at Line 22. However, in general, a fully automated method cannot achieve such accuracy.

Our approach starts from the failed assertion at Line 22 and traverses the execution trace in reverse order in order to construct a quantifier-free, first order logic formula. The formula captures the weakest precondition of the assertion predicate `(area==ORACLE)`, which is the minimal requirement for the assertion to hold along this trace. We stop this backward trace traversal as soon as the logic formula of the weakest precondition becomes unsatisfiable, which is guaranteed to happen by the time we reach the starting point of the trace. This is because the backward traversal assumes that `(area==ORACLE)` holds at Line 22, but in reality, we have `(area!=ORACLE)`. We check for unsatisfiability by using an off-the-shelf satisfiability modulo theory (SMT) solver [YIC].

```
1  int area(int a, int b, int c) {
2    int class;
3    double s,area;
4    a = t1, b = t2, c = t3; //automatically inserted symbols: t1, t2, t3
5    if (a>=b && b>=c) {
6      class = SCALENE;
7      if (a==b || b==c)
8        class = ISOSCELES;
9      if (a*a != b*b+c*c)   //the condition should be (a*a==b*b+c*c)
10       class = RIGHT;
11     if (a==b && b==c)
12       class = EQUILATERAL;
       switch(class) {
13       case RIGHT:
14         area = b*c/2; break;
15       case EQUILATERAL:
16        area = a*a*sqrt(3)/4; break;
17       default:
18        s = (a+b+c)/2;
19        area = sqrt(s*(s-a)*(s-b)*(s-c));
       }
     }else {
20     class = ILLEGAL;
21     area = 0;
     }
22   assert (area==ORACLE);  //the failure
23   return area;
24 }
```

Fig. 1. A program that computes the area of a triangle: there is a bug at Line 9 causing the assertion failure at Line 22.

```
1  x = t;
2  y = x+1;
3  assert (y==1);
```

Fig. 2. Code snippet that illustrates weakest precondition computation.

```
1  x = 0;
2  y = 1;
3  if (y == 1)
4      x = x+1;
5  assert (x==0);
```

Fig. 3. Code snippet that illustrates critical conditions.

In order to explain the procedure further we use a simpler code snippet shown in Figure 2, where there is an assertion failure under input ⟨t≡1⟩. Starting from the assertion failure at Line 3 we create the initial formula `(y==1)`. At Line 2 the weakest precondition computation substitutes `y` with `x+1`, which produces new formula `(x+1==1)`. Finally the computation terminates at Line 1 because the replacement of `x` with `t` leads to a unsatisfiable formula `(t+1==1)`. In the running example, our backward traversal stops at Line 4 and produces a more complicated unsatisfiable formula shown here.

```
(t2*t3/2==ORACLE) && (class==RIGHT) && (t1 != t2) && (t1*t1 != t2*t2+t3*t3)
 && (t2 != t3) && (t1 != t2) && (t2 >= t3) && (t1 >= t2) && (t1 == 6)
 && (t2 == 5) && (t3 == 4)
```

Here, the values of `t1`, `t2`, and `t3` are 6, 5, and 4, respectively. For the purpose of localizing the cause of a failure, our observation is that not all of the constraints in the unsatisfiable weakest precondition formula contribute to the unsatisfiability. For example, removing `(class==RIGHT)` from the above formula would not make it satisfiable, which means that the constraint is irrelevant to the proof of its unsatisfiability. If we keep removing such irrelevant constraints, in the end, we would

be able to obtain an unsatisfiable (UNSAT) core, which consists of a subset of logical constraints of the unsatisfiable formula. In this example, the UNSAT core is `(t2*t3/2==ORACLE)&&(t2==5)&&(t3==4)`.

Although the constraints in the above UNSAT core somewhat explain the assertion failure—since `5*4/2==9.92` is not true—they are not very helpful to the programmer before we map these UNSAT core constraints back to the erroneous statements in the program code. Toward this end, we consider the statements that are responsible for the UNSAT core constraints as *potential* causes. For the UNSAT core constraint `(t2*t3/2==ORACLE)`, one of such statements is the assignment at Line 14, since `t2*t3/2` is obtained by executing `area=b*c/2`. This mapping algorithm will be presented in Section 3.2. Although existing methods such as BugAssist [Jose and Majumdar 2011a, 2011b] and error invariants [Ermis et al. 2012; Christ et al. 2013; Murali et al. 2014] also compute UNSAT cores and use variants of the weakest precondition computation, they focus on identifying only one cause, for instance, by heuristically deciding the most likely root cause, whereas our method focuses on systematically generating all possible causes.

After computing the first cause, the question now is whether the assignment `area=b*c/2` is a cause. We believe that in this example, the answer is yes because the failure can be avoided by changing this assignment to `area=`*sqrt*`(s*(s-a)*(s-b)*(s-c))`, where *s* is defined by `s=(a+b+c)/2`. Unfortunately, this is not the *root* cause—recall that the real defect is in the conditional expression at Line 9. This shows the main challenge in trying to design fully automated methods for fault localization—in general, there is no way to completely automate this process without knowing the programmer's intent. For a mechanical approach that does not know sufficient domain knowledge or the programmer's intention, we argue that the best it can accomplish is to compute all the potential causes and then let the programmer decide which is the root cause.

Our running example highlights the importance of computing more than one causes since the first cause is often not the root cause. If we stop after computing the first cause at Line 14, the programmer would not be able to obtain any information about the root cause at Line 9.

In order to continue the analysis beyond the first cause we introduce the concept of *critical conditions*, which initiate weakest precondition computations to search for multiple potential causes. The need for critical conditions can be clarified by a simpler example shown in Figure 3. The first critical condition is `(x==0)` that corresponds to the assertion failure itself. The first weakest precondition computation produces the cause that consists of Lines 1, 4, 5. However, if the user believe both Lines 1 and 4 are correct, the condition at Line 3 becomes critical because its negation prevents the execution of Line 4, which dissolves the previously computed UNSAT core. Therefore we can start from the second critical condition `(y!=1)` and obtain the second cause (Lines 2 and 3). Since there are no more critical conditions and the user vetoes the first cause, the second cause must be the root cause. Indeed, the assertion failure can be avoided by changing either Line 2 to `(y=0)` to or Line 3 to `(y!=1)`.

Back to our running example, after identifying the first cause at Line 14, our method continues by choosing the condition at Line 13 as a new starting point, since the condition controls whether Line 14 will be executed. A negation of the condition at Line 13, `(class!=RIGHT)`, would be able to avoid executing Line 14, which is necessary to trigger the observed assertion failure. By performing the weakest precondition computation from the negated condition at Line 13, we compute the second UNSAT core `(RIGHT!=RIGHT)`. The program statements responsible for this UNSAT core include Lines 13 and 10. Similarly, the conditions at Lines 9, 5, and 11 are critical conditions. The one at Line 9 is a critical condition because it indirectly leads to the assertion failure by influencing the outcome of the predicate at Line 13. The condition at Line 5 is a critical condition because it affects the value of variable *area* at Line 22. The condition at Line 11 is

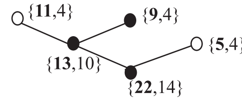

Fig. 4. The cause tree for the program in Figure 1, where each node (cause) corresponds to a set of line numbers in the code.

a critical condition because of its potential influence on the variable *class* at line 13. Although Line 12 is not executed in the given trace, its impact can be estimated by a simple static program analysis. Note that Line 7 is not a critical condition, because a simple static analysis will be able to show that its influence on *class* will be blocked at Line 10 and therefore cannot arrive at Line 13 in the failed trace. Starting from Line 9, using the same approach, we can find the third UNSAT core ($6^2 = 5^2 + 4^2$). After mapping the constraints back to the assignments in the code, we are able to report another cause {9,4}.

The causes computed by our method are organized in a tree structure, an example of which is shown in Figure 4. Each node in this tree represents a cause, labeled with a set of line numbers of the source code that constitute the cause. These line numbers may correspond to either *critical condition* or the *supporting statements*. The line numbers of critical conditions are in bold—the failure would go away were their values negated. The supporting statements, in contrast, help explain the unsatisfiability of the weakest precondition predicates originated from these critical conditions. The edges in Figure 4 denote the causal relationships between these causes.

Since each cause in the tree forms a context that explains itself, it helps the programmer understand the cause and decide whether it is the real bug. For example, the cause $\{\mathbf{9}, 4\}$ in Figure 4 essentially says that ($6^2 = 5^2 + 4^2$) because of the initial values of variables set at Line 4, which means that the condition at Line 9 should be modified.

Although our tree representation in Figure 4 can help the programmer navigate through the causes, in practice, debugging remains a long and difficult process. When the number of causes is large, applying our method is the most beneficial since in addition to providing a relevant context for each cause, our method also highlights the sometimes complex causal relationships between these causes. Our tree representation also makes *on-the-fly* pruning of redundant causes possible. For example, as soon as the programmer decides that a potential cause is not responsible for the manifested failure, we can remove all the causes leading to this *benign* cause, thereby removing a large number of irrelevant nodes and edges from the cause tree.

3. CASCADE ANALYSIS

Figure 5 depicts the architectural design of our tool CaFL, which uses the KLEE [Cadar et al. 2008] symbolic virtual machine in LLVM [Lattner 2002] as a front-end to obtain and replay the faulty execution trace, which may come from a failing test run. The key step in applying CaFL is to perform the cascade analysis, which produces the augmented cause tree. Starting from the root node of the tree, which directly explains the manifested failure, more causes that transitively explain the failure are added to the tree level by level. The algorithm terminates as soon as either the user discovers the root cause or the complete cause tree is constructed.

In the remainder of this section, we first present our algorithm for computing a cause. Then, we present our algorithm for computing additional causes based on the previously computed cause. Finally, we present the top-level procedure of our cascade analysis method.

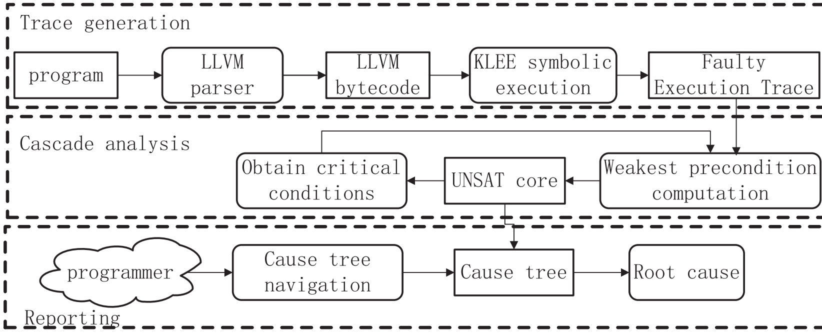

Fig. 5. The overview of our new CaFL (***Ca**scade **F**ault **L**ocalization*) framework.

3.1. Weakest Precondition Computation

Our method for computing a cause that explains the manifested failure is based on an enhanced version of the weakest precondition computation, defined originally by Dijkstra [1976]. Let $\pi^{1,n} \leftarrow \langle s_e^1, \ldots, s_j^i, \ldots, s_m^n \rangle$ be an execution trace consisting of n instruction instances. Each s_j^i is the i-th executed instance of the corresponding instruction s_j in the program. In our implementation, the program is represented in the LLVM bytecode format, constructed from the C/C++ source code using the Clang/LLVM compiler. Each instance may have one of the following types:[1]

—assignment $v := e$, where v is a variable and e is an expression;
—branch *assume(c)*, where c is a predicate that comes, for example, from one of the two branches of an if-else statement.

We assume that the last instance s_m^n in the trace $\pi^{1,n}$ is a failed assertion. In software testing, assertions are often included in the program code by the programmer, or inserted automatically as part of the test oracle. Standard programming errors such as *null-pointer-dereference*, *array-bound-violation*, and *division-by-zero* can also be modeled using assertions. For ease of presentation, we omit the subscripts when the context is clear and unify the trace with its input as $\pi^{0,n} \leftarrow \langle s^0, s^1, \ldots, s^i, \ldots, s^n \rangle$. Here, s^0 represents a conjunctive set of assumptions under which the input variables equal to their initial values. We use $\pi^{i,j}$ to represent the trace segment $\langle s^i, \ldots, s^j \rangle$, where $0 \le i, j \le n$.

Our analysis is based on computing the weakest precondition of the failed assertion condition. We have extended the original definition of weakest precondition in order to handle pointer-based memory accesses. Informally, the weakest precondition of predicate ϕ over an instruction instance s, denoted $WP(s, \phi)$, is the weakest condition that needs to hold before s such that ϕ is guaranteed to hold after executing s. For handling pointer dereferences, we add an auxiliary variable $\&m_i$ for each memory address that has been referenced by a pointer. For example, executing `int* p=malloc(2*sizeof(int))` would lead to $(p = \&m_0) \wedge (p + 1 = \&m_1)$, where m_0, m_1 are two integers and $\&m_0, \&m_1$ are their memory addresses. Since the weakest precondition is computed along a concrete execution trace that is obtained from a forward execution of the program. Therefore, all pointers already have fixed values. In other words, the alias information is already available to us by the time we perform the backward weakest precondition computation. For example, when the pointer pp points to m_1, the assignment `*pp:=x`

[1]We ignore memory allocation and deallocation to simplify the presentation.

can be represented by `assume(pp==&`m_1`)`,m_1`:=x`. Similarly, the statement `if(*pp==3)` can be represented by `assume(pp=&`m_1`)`,`assume(`m_1`==3)`.

Definition 3.1 gives the formal definition of weakest precondition. Since pointer related operations have been represented by combinations of *assume* and *assignment* statements, the definition is straightforward.

Definition 3.1 (*Weakest Precondition*). The *weakest precondition* of predicate ϕ over trace $\pi^{i,j} \leftarrow \langle s_i, \ldots, s_{j-1}, s_j \rangle$, denoted $WP(\pi^{i,j}, \phi)$, is defined by applying the following rules:

—*Rule1*. For a sequence of instructions $\pi^{i,j}$, we have $WP(\pi^{i,j}, \phi) = WP(\pi^{i,j-1}, WP(s_j, \phi))$.
—*Rule2*. When s is an assignment `v:=e`, we have have $WP(s, \phi) = \phi(e/v)$.
—*Rule3*. When s is an `assume(c)`, we have $WP(s, \phi) = \phi \wedge c$.

Table I illustrates how these rules are applied to a small example. Column 1 shows the sequence of original statements in the C program $\langle 0, 1, 2, 3, 4, 5, 6, 8 \rangle$ executed under the test input $\langle x = 2, y = 3 \rangle$. Column 2 shows the execution trace represented by a sequence of assume/assignment statements. The trace ends at Line 8, where the assertion fails since the value of `*pp` (aliased to m_1) is 4 instead of 3. That is, we have `assume(`$m_1 \neq 3$`)`, which contradicts to `assert(*pp==3)`. To compute the weakest precondition (WP) of the condition ($m_1 = 3$), we traverse the execution trace backwardly. Column 3 shows the the impact of each instruction on the WP computation, where U denotes the expression substitution and A denotes the addition of a new conjunct. Column 4 shows the result obtained after processing each instruction, starting from the assert statement at Line 8.

3.2. Identifying a Cause in the Faulty Execution

Not all the instruction instances in a faulty trace may be responsible for the manifested failure. Our first goal is to identify the minimal set of instructions that directly causes the failure. Let s^n be the last instruction instance in trace $\pi^{0,n}$ and the failed assertion predicate is $s^n.p$. The assertion fails because the desired value of $s^n.p$ is false. In other words, if the assertion predicate were $(\neg s^n.p)$ as opposed to $(s^n.p)$, the failure would have been avoided. Therefore, we want to know why the execution trace prefix $\pi^{1,n-1}$ is not compatible with $\phi_n = (\neg s^n.p)$.

The weakest precondition of predicate ϕ_n with respect to the trace $\pi^{i,n-1}$ is of the form $WP(\pi^{i,n-1}, \phi_n) = \phi'_n \wedge (p'_{l_1} \wedge \ldots \wedge p'_{l_k})$. Here, ϕ'_n is transformed from the predicate ϕ_n through variable substitutions, and each additional predicate p'_l is derived from a condition in an assume statement, denoted $s^l : assume(c_l)$, where $i \leq l \leq n-1$.

By the definition of ϕ_n and $WP()$, there always exists an index $0 \leq i \leq n-1$ such that $WP(\pi^{i,n-1}, \phi_n)$ becomes unsatisfiable. The reason is that the assumption of ϕ_n contradicts the fact that under the given test input, the assertion has failed during this execution. Therefore, during the backward WP computation, we check for the satisfiability of the WP at each step.

Let $WP_{unsat}(\phi_n)$ be the first unsatisfiable formula obtained during the backward weakest precondition computation. According to the satisfiability (SAT) theory, for each unsatisfiable formula, there exists an UNSAT core [Zhang and Malik 2003; Lynce and ao Marques-Silva 2004], which is defined as a subset of constraints of the formula that by themselves are also unsatisfiable. A minimal UNSAT core, denoted $WP^{min}_{unsat}(\phi_n)$, is one such that removing any constraint from it would make the remaining formula satisfiable. Modern SAT and SMT solvers, such as Yices [YIC], can be leveraged to compute the minimal UNSAT core [Liffiton and Sakallah 2008].

Table I. A Weakest Precondition Computation Example where the Given Execution Trace is $\langle 0, 1, 2, 3, 4, 5, 6, 8 \rangle$

Program	Trace	Rule	Weakest Precondition
0: *int x=2, y=3;*	x:= 2, y:= 3	U: $2/x$, $3/y$	step12: $(\mathbf{2} + \mathbf{2} = \mathbf{3}) \wedge (true) \wedge (2 \leq 4) \wedge (true) \wedge (true)$
1: *int *p = malloc(2*sizeof(int));*	p:=&m_0, p+1:=&m_1	U: $\&m_0/p$, U: $\&m_1/p$+1	step11: $(x + 2 = 3) \wedge (true) \wedge (x \leq 4) \wedge (\&m_0 = \&m_0) \wedge (true)$ step10: $(x + 2 = 3) \wedge (\&m_1 = \&m_1) \wedge (x \leq 4) \wedge (\&p = \&m_0) \wedge (\&m_1 = \&m_1)$
2: *int *pp = p+1;*	pp:= p+1	U: p+1$/pp$	step9: $(x + 2 = 3) \wedge (p + 1 = \&m_1) \wedge (x \leq 4) \wedge (p = \&m_0) \wedge (p + 1 = \&m_1)$
3: *x = x+2;*	x:= x+2	U: x+2$/x$	step8: $(x + 2 = 3) \wedge (pp = \&m_1) \wedge (x \leq 4) \wedge (p = \&m_0) \wedge (p + 1 = \&m_1)$
4: *p[1] = x;*	assume(p+1=&m_1) m_1:= x	A: (p+1=&m_1), U: x/m_1	step7: $(x = 3) \wedge (pp = \&m_1) \wedge (x \leq 4) \wedge (p = \&m_0) \wedge (p + 1 = \&m_1)$ step6: $(x = 3) \wedge (pp = \&m_1) \wedge (x \leq 4) \wedge (p = \&m_0)$
5: *p[0] = y;*	assume(p=&m_0) m_0:= y	A: (p=&m_0), U: y/m_0	step5: $(m_1 = 3) \wedge (pp = \&m_1) \wedge (x \leq 4) \wedge (p = \&m_0)$ step4: $(m_1 = 3) \wedge (pp = \&m_1) \wedge (x \leq 4)$
6: *if (x>4)*	assume(x≤4)	A: $x \leq 4$	step3: $(m_1 = 3) \wedge (pp = \&m_1) \wedge (x \leq 4)$
7: *p[1] = 3;*			
8: *assert(*pp == 3);*	assume(pp=&m_1) assume(m_1!=3)	A: (pp=&m_1), ϕ: (m_1=3)	step2: $(m_1 = 3) \wedge (pp = \&m_1)$ step1: $(m_1 = 3)$

Although $WP_{unsat}^{min}(\phi_n)$ is a useful concept for fault localization, by itself, the UNSAT core does not provide enough information to help the programmer locate the faulty program statement in the source code. Let p'_l be a conjunctive constraint in $WP_{unsat}^{min}(\phi)$. In order to present a meaningful cause to the programmer, we map p'_l back to the program statements that contribute to the formula of p'_l. Assume that p_l is the original predicate that is transformed into p'_l eventually. We consider all the assignment statements that participate in the transformation of p_l into p'_l are part of the cause. Intuitively, it is the execution of these assignments that eventually gives rise to p'_l, and hence the unsatisfiable formula.

Definition 3.2 (*Transforming Instance*). A *transforming instance* (TI) of the predicate p_l is an assignment $v := e$, a memory read $v := [p]$, or a memory write $[p] := e$ from the given execution trace, such that v and $[p]$ appear in the transitive support of p_l or appear in the branching condition where p_l comes from.

For example, the assignment $s : x = y + 1$ is a transforming instance of the predicate $p_l : (x > 0)$, since $WP(s, p_l)$ produces $p'_l : (y + 1 > 0)$. The definition is transitive in that both $\phi' = \phi(e/v)$ and $\phi'' = \phi'(e_2/v_2)$ are transformed from ϕ. Note that only an assignment can transform a predicate. An assume statement, in contrast, can only add a new predicate to the existing WP formula, but cannot transform the existing predicates. Let TI_p denote the set of transforming instances for the predicate p along the given trace. A cause for the failed execution $\pi^{0,n}$, whose corresponding assertion failure is $\phi = \neg s^n.p$, is defined as follows:

$$cause_\phi = \bigcup_{p \in WP_{unsat}^{min}(\phi)} TI_p.$$

Algorithm 1 presents the pseudocode that computes a cause for a failed execution $\pi^{0,k}$. The input consists of the trace prefix $\pi^{0,k-1}$ and the failed assertion in s^k. As discussed earlier, $\phi = (\neg s^k.p)$ represents an alternative outcome of the execution that ensures unsatisfiability of the WP at some step. (Later, in Section 3.3, we show that s^k may also be some instance other than the final assertion.) The resulting formula obtained during the weakest precondition computation is represented as a set of conjuncts. At each iteration of the backward computation, we check the satisfiability (Line 15) after the formula $WP_\phi = WP(\pi^{i,n-1}, \phi)$ is updated, either by adding new conjuncts via a branching statement (Line 13) or by substitution made by an assignment (represented by procedure *update* at Lines 5, or 9). If WP_ϕ becomes unsatisfiable at any moment, we compute the UNSAT core $WP_{unsat}^{min}(\phi)$ using an SMT solver. The last `for` loop computes the cause of the failed assertion, by adding to $cause_\phi$ all the transforming instances of the predicates in the UNSAT core $WP_{unsat}^{min}(\phi)$.

For our running example in Table I, the *WP* formula becomes unsatisfiable at Step 8, and its minimal UNSAT core is $2 + 2 = 3$. Let s_i be the instruction instance at Line i. According to Algorithm 1, the cause of this failing trace is $\{s_8, s_4, s_3, s_2, s_0\}$. Note that s_2 can be identified as one transforming statement of the condition at s_8, thus it is included in the computed cause.

3.3. Producing More Causes

Algorithm 1 computes a cause of the failed execution $\pi^{0,n}$. In principle, there are two ways to avoid the manifested failure. One is to revise one or more transforming instances in some TI_ϕ of the cause, which would invalidate the UNSAT core. Such direct fix makes sense only if TI_ϕ is the root cause. The other way to avoid the failure is to make one or more transforming instances unreachable. This, however, means that

ALGORITHM 1: $computeCause(\pi^{0,k-1}, s^k)$ – Computing the Cause for Failure in s^k.

Data: the failed assertion condition $\phi = (\neg s^k.p)$;
Result: the $cause_\phi$;

1 $WP_\phi.add(\phi)$;
2 $i = k - 1$;
3 **for** $i \geq 0$ **do**
4   **if** s^i *is* $v := e$ **then**
5         $update(WP_\phi, s^i, v, e)$;
6   **end**
7   **if** s^i *is* $assume(c)$ **then**
8         $WP_\phi.add(c \wedge \bigwedge_{p \in c \wedge q \in (\Phi(p) - \{p\})} c(q/p))$;
9   **end**
10   **if** WP_ϕ *is unsatisfiable* **then**
11         $WP^{min}_{unsat}(\phi) \leftarrow getUnsatCore(WP_\phi)$;
12   *break*;
13   **end**
14     $i - -$;
15 **end**
16 **for** *every* $p \in WP^{min}_{unsat}(\phi)$ **do**
17     $cause_\phi.add(TI_p)$;
18 **end**
19 **return** $cause_\phi$;
20 Procedure $update(WP, s, oldVar, newExpr)$;
21 **for** *each conjunct c in WP* **do**
22   **if** *c uses oldVar* **then**
23         $c = c[newExpr/oldVar]$;
24         $TI_c.add(s)$;
25   **end**
26 **end**

TI_ϕ is not the root cause, and therefore we need to focus on instances outside TI_ϕ. In the latter case, we need to extend our algorithm to compute the instances outside TI_ϕ.

Now, we present a cascade analysis to compute more causes based on a previously computed cause. Given the current transforming instances in *TI*, we first locate the branching instances that control the execution of the instances in *TI*. The conditions in these branching instances are called *critical conditions*.

Definition 3.3. The condition in a branching instance s^j is called a *critical condition* of some *TI* of a cause if it has *potential* impact on an instance $s^l \in TI$ in one of the following ways:

—direct influence $s^j \rightsquigarrow s^l$: the condition in s^j determines whether s^l will be executed; and
—indirect influence $s^j \looparrowright s^l$: the condition in s^j determines whether an unexecuted statement s^p will be executed, where s^p redefines a variable read by s^l.

In other words, the set CC_{TI} of critical conditions of *TI* is defined as $CC_{TI} = \bigcup_{s^l \in TI}\{s^j \mid (s^j \rightsquigarrow s^l) \vee (s^j \looparrowright s^l)\}$. Our use of *indirect influence* in the above definition is similar to the *potential dependency* used in relevance slicing [Gyimóthy et al. 1999]. By considering the indirect influence in addition to the direct influence, we can detect *execution omission errors* [Zhang et al. 2007], which are caused by not executing certain necessary program statements in the code.

Consider the example in Figure 6, which has an assertion failure along the execution $\langle 1, 2, 3, 6, 7\rangle$ because the value of d becomes 4, not 5, at Line 7. Using the algorithm presented in Section 3.2, we compute the first cause with $TI_\phi = \{1, 6, 7\}$. Let us assume that TI_ϕ is not the root cause. In other words, we should not make the failure go away

```
1 int a=2, b=1, c=1, d=0;
2 if (a>0) {
3   if (b<0)
4     if (c!=2)
5        c=2; //end if\@L4
6   d=c+3;    //end if\@L3
  }           //end if\@L2
7 assert(d==5);
```

Fig. 6. An example for computing the critical conditions.

ALGORITHM 2: *ComputeCC*(*TI*) – Computing the Critical Conditions from *TI*.

Data: the current set *TI* of transforming instances;
Result: critical conditions CC_{TI} for *TI*;
1 **for** *each transforming instance* $s^l \in TI$ **do**
2 let s^j be the closest enclosing branch of s^l that is not post-dominated by s^l;
3 $CC_{TI}.add(s^j)$;
4 **for** *each var read by* s^l **do**
5 let s be the last instruction instance in the trace that assigns *var* before s^l;
6 **for** *every branching instruction* s^x *between* s *and* s^l **do**
7 **if** $indInf(s^x, var) == true$ **then**
8 $CC_{TI}.add(s^x)$;
9 **end**
10 **end**
11 **end**
12 **end**
13 **return** CC_{TI};

by changing `c=1` to `c=2` at Line 1, `d=c+3` to `d=c+4` at Line 6, or `d==5` to `d==4` at Line 7. In such case, we check if the behavior of Line 6 can be changed indirectly.

Based on the definition of critical conditions, we know that Line 2 is a critical condition since it controls whether Line 6 will be executed. Similarly, Line 3 is a critical condition since if the condition at Line 3 is *true*, the omitted statement at Line 5 would have been executed and *c* would have been redefined. Since we only consider such direct and indirect influence on the *executed* instruction instances, Line 4 would not be considered as a critical condition.

By negating the two critical conditions and passing them each as an input to Algorithm 1 as the second parameter, we can identify the reasons why `a<=0` (denoted $cc1$) and `b>=0` (denoted $cc2$) did not hold at Lines 2 and 3, respectively. In this way, we have computed, from the cause at Level 0, the two new causes $TI_{cc1} = \{1, 2\}$ and $TI_{cc1} = \{1, 3\}$ at Level 1.

Our new method differs from the well-known dynamic slicing techniques [Agrawal and Horgan 1990]. In particular, dynamic slicing would have reported Lines 1–2 and 6–7 in Figure 6, but not Line 3, as the potential instances causing the failure, which can lead to certain causes being left out. For example, the actual bug can be fixed if the programmer changes `(b<0)` to `(b>0)` at Line 3.

Algorithm 2 shows the pseudocode for computing the critical conditions. It follows Definition 3.3 with the following modification. For each transforming instance $s^l \in TI$, we choose only the immediate preceding instance s^j that is not post-dominated by s^l (Lines 2–3), because other critical conditions will be computed during the subsequent iterations of our cascade analysis. Therefore, no root causes will be missed due to this modification. Indeed, making this computation incremental is advantageous in that it makes efficient pruning possible: if the user decides, at any moment, that the

ALGORITHM 3: $topAlg(\pi^{0,n})$ – The Top-Level Procedure of Our Cascade Analysis.

```
    Data: π^{0,n};
    Result: causes;
1   S_cc ← {(s^n, 0)};
2   while S_cc ≠ ∅ do
3   |   remove (s^k, i) from S_cc;
4   |   TI = computeCause(π^{0,k−1}, s^k);   // Algorithm 1
5   |   causes.add(TI, i);
6   |   CC = computeCC(TI);                  // Algorithm 2
7   |   S_cc = S_cc ∪ {(c, i + 1)|c ∈ CC};
8   end
```

current critical condition is not responsible for the manifested failure, then we can avoid exploring all its corresponding causes at the higher levels.

In Algorithm 2, Lines 4–11 deal with the *indirect* influence. Let s be the last statement before s^l that assigns a variable var used by s^l, then during the backward analysis, s and s^l represent the upper and lower bounds of indirect influence, respectively. For each var used by s^l, the loop at Line 6 identifies every branch instance s^x between s and s^l—which can indirectly influence var, represented by $indInf(s^x, var)$—as a critical condition. An example for such instruction instance is the one at Line 3 in Figure 6.

3.4. The Top-Level Algorithm

Algorithm 3 presents the top-level procedure of our cascade analysis, which takes a failed execution $\pi^{0,n}$ as input and returns an augmented cause tree as output. The critical condition set S_{cc} initially contains $(s^n, 0)$, which represents the failed assertion at Level 0. Within each iteration, a critical condition at Level i, where $i \geq 0$, is removed from S_{cc} and the weakest precondition computation is performed to obtain an UNSAT core.

The UNSAT core is then mapped to a set of transforming instances in *TI*. Each *TI* is then considered as a possible cause at Level i, from which we can compute a set of critical conditions CC at Level $i + 1$. By appending CC to S_{cc}, the cascade analysis continues until all critical conditions are transitively added, or (not shown in the pseudocode) the programmer discovers the root cause and therefore decides to terminate the process early.

During the cascade analysis, any newly discovered critical condition can initiate a new cause detection computation by serving as the second parameter to Algorithm 1. Such analysis would produce multiple causes. All these causes are possible bugs in the sense that a change to any of them may help avoid the final failure. To help the programmer navigating through these possible causes, we organize them level by level in a tree, an example of which is shown in Figure 4.

4. OPTIMIZATIONS

The cause tree constructed by the algorithm in Section 3 can be large, especially for non-trivial programs with complex control and data dependency. Similar to the standard debugging practice, here, a programmer may have to browse many causes before identifying the root cause. This is a main reason why fault localization is challenging. In this section, we present several optimization techniques to speed up the construction of the cause tree, as well as reduce its size via on-the-fly pruning. Furthermore, these optimizations try to minimize the amount of code that a programmer has to scrutinize before identifying the root cause.

4.1. Slicing the Faulty Execution Trace

The length of the faulty execution trace has a direct impact on the efficiency of our algorithm. Therefore, we follow the principle of dynamic slicing [Agrawal and Horgan 1990] and use an approach similar to the data-flow equation method [Weiser 1984] to remove the irrelevant instances from the given execution trace before applying our cascade analysis.

The *slicing criterion* is defined as $C = (s^n, V)$, where s^n denotes the manifested failure and V is the set of program variables observed at s^n. Let *def(s)* and *ref(s)* be the sets of variables that are defined and used at instance s, respectively. We use an auxiliary set R_C^i to record the set of variables at instance s^i that can affect C either directly or indirectly. The set R_C^i is computed along the given trace $\pi^{0,n}$ in a reverse order as follows:

$$R_C^i = \begin{cases} V, & \text{if } i = n, \\ \{v \mid v \in R_C^{i+1} \wedge v \notin def(s^i)\} \,\cup\, \{v \mid (R_C^{i+1} \cap def(s^i) \neq \emptyset \wedge v \in ref(s^i)\}, & \text{otherwise.} \end{cases}$$

When $i \neq n$, the set R_C^i is a union of two subsets. The first subset consists of all the variables in R_C^{i+1} except for those redefined at s^i. The second subset contains all the variables that are referenced at s^i if s^i redefines some variable in R_C^{i+1} – this condition is represented as $R_C^{i+1} \cap def(s^i) \neq \emptyset$.

Next, we define the condition TF, which states that the instance s^i in the given trace should be added to the slice if it is data or control dependent on s', or there exists indirect influence between them. TF is computed as follows:

$$TF = def(s^i) \cap R_C^{s^{i+1}} \neq \emptyset \;||\; ctrdep(s^i, s') || \;\exists v \in R_C^{s^{i+1}} \wedge indInf(s^i, v).$$

In the above equation, s' represents the most recent instance added to the slice and $ctrdep(s, s')$ is true when s' is control dependent on s. Note that $indInf(s, v)$, which has been defined in Section 3.3, is used in our indirect influence analysis.

Finally, given R_C^i and TF, we compute S_C^i, which represents the resulting program statements after slicing the given trace $\pi^{i,n}$. Here, S_C^i is defined as follows:

$$S_C^i = \begin{cases} S_C^{i+1} \cup s^i, & \text{if } TF = \text{true} \\ S_C^{i+1}, & \text{otherwise.} \end{cases}$$

4.2. Designated Correct Functions

In practice, certain type of software code are normally regarded by the developers as bug-free. An example is the set of functions in the standard C library, such as *strcmp*. In CaFL, we leverage this type of *domain-specific* information to prune the cause tree. Toward this end, we define the *analysis scope* in CaFL as the set of all functions that are neither the standard library functions nor the ones designated by the programmer as being correct.

When a critical condition cc is identified to be outside the analysis scope, CaFL does not compute the causes derived from cc. Instead, CaFL maps cc to the nearest caller site (cs) of a function inside the analysis scope. Subsequently, the caller site cs is used as the new starting point for computing critical conditions, because an incorrect value generated in a faulty function may be passed to a correct function through a function invocation chain. Moreover, if the user knows that a correct function has no side-effect, CaFL can use the concrete return value of that function call without any symbolic computation inside the function. This optimization not only simplifies the resulting cause tree, but also speeds up the corresponding weakest precondition computation.

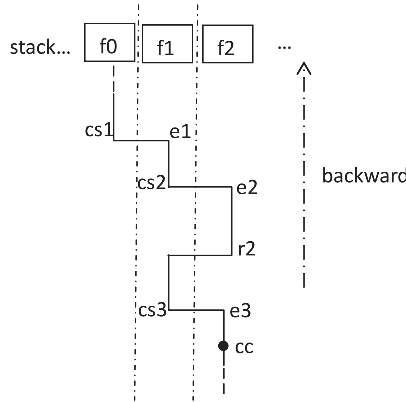

Fig. 7. Mapping a critical condition to a call site within the analysis scope (by skipping the designated correct functions).

Figure 7 depicts a faulty execution trace marked with three call sites $\{cs_1, cs_2, cs_3\}$, the function call entry points $\{e_1, e_2, e_3\}$, and the return points $\{r_1, r_2, r_3\}$. When the critical condition *cc* within a designated correct function *f2* is identified, CaFL maps *cc* to the call site *cs3* if the function *f1* is within the analysis scope. If *f1* is also designated as correct, then CaFL maps *cs3* further up in the call stack, to the call site *cs1* within the analysis scope.

4.3. Improving the SMT Solving

Constraint solving can become a bottleneck in our cascade analysis as it relies on calling an off-the-shelf SMT solver to check the satisfiability of a weakest precondition formula, and compute the UNSAT core in case of unsatisfiability. In general, formulas with fewer constraints are easier to solve. Therefore, inside CaFL, we try to prune away redundant constraints from the WP formula before passing it to the SMT solver.

We say that the constraint c_i *intersects* with the constraint c_j if they share at least one variable. We say that c_i and c_j *transitively intersect* with each other if there exists a chain of constraints $\langle c_i, c_{i+1}, \ldots, c_{j-1}, c_j \rangle$ such that any two adjacent constraints intersect with each other. For example, since $c1 : (x > y)$ intersects with $c2 : (y < z+2)$, and $c2 : (y < z + 2)$ intersects with $c3 : (z < 0)$, we say that $c1$ transitively intersects with $c3$.

Our experience with real-world applications shows that only a small fraction of the constraints obtained during our weakest precondition computation transitively intersect with each other. Let C_n be the subset of constraints in the WP formula obtained from the assertion failure. A constraint is regarded as *redundant* and therefore can be pruned away if it does not transitively intersect with any constraint in C_n. Besides speeding up the SMT solving, our optimization also minimizes the number of calls to the SMT solver. Specifically, we skip the satisfiability check of the WP formula whenever a constraint c_i added or updated in the WP formula does not transitively intersect with the currently evaluated predicate cc. For instance, at step 5 of the WP computation illustrated in Table I, CaFL will not check the satisfiability of the updated weakest precondition when variable x is updated to $x + 2$, since it does not transitively intersect the assertion instance.

This optimization differs from the dynamic slicing techniques described in Section 4.1. Slicing will remove all the statements on which the failed assertion does not depend. However, it will not remove statements that do not transitively intersect with the currently checked condition. Consider the code snippet in Figure 6 as an

```
 1:  a=7, b=5, c=0;
 2:  if (b != 0) // error: b==0
 3:    a = 8;
 4:  while(a>0) {
 5:    if (a>b)
 6:      c++;
       else
 7:      break;
 8:    a--;
 9:  }
10:  assert(c==2);
```

Fig. 8. Code snippet with the faulty execution trace $\langle 1, 2, 3, 4, 5, 6, 8, 4, 5, 6, 8, \ldots, 4, 5, 7, 10\rangle$.

Table II. The Causes Originated from the Loop Iterations of the Example Program in Figure 8

Critical Conditions	Cause	Statements
s_5^5	$\{s_5^5, s_3^3, s_1^1\}$	$\{5, 3, 1\}$
s_5^9	$\{s_5^9, s_8^7, s_3^3, s_1^1\}$	$\{5, 8, 3, 1\}$
s_5^{13}	$\{s_5^{13}, s_8^{11}, s_8^7, s_3^3, s_1^1\}$	$\{5, 8, 3, 1\}$

example. When our backward analysis arrives at Line 3, the weakest precondition becomes $(c + 3 == 5) \wedge (b \geq 0)$. Line 3 cannot be removed by slicing, but with our new solver-related optimization, the constraint $(b \geq 0)$ does not need to be fed to the SMT solver when checking for the satisfiability of critical condition $(c + 3 == 5)$, because they do not intersect with each other. In fact, CaFL will not invoke the SMT solver at Line 3, because the satisfiability of $(c + 3 == 5)$ has been checked previously, and the current update of the WP formula does not affect its satisfiability.

4.4. Handling Loops and Recursions

The execution of loops or recursive calls may result in a lengthy trace, which in turn can lead to a large number of causes. These causes may contain the transforming instances that map to the same lines in the source code. Consider the code snippet given in Figure 8, whose faulty execution trace is $\langle s_1^1, s_2^2, s_3^3, s_4^4, s_5^5, s_6^6, s_8^7, s_4^8, s_5^9, s_6^{10}, s_8^{11}, s_4^{12}, s_5^{13}, s_6^{14}, s_8^{15}, s_4^{16}, s_5^{17}, s_7^{18}, s_{10}^{19}\rangle$. After we compute the first cause, which is $\{s_1^1, s_6^6, s_6^{10}, s_6^{14}, s_{10}^{19}\}$, the following three instances are identified as critical conditions: s_5^5, s_5^9, and s_5^{13}.

Table II shows the causes related to these three critical conditions and the line numbers of the transforming instances in each cause. Although the last two causes have different transforming instances, they map to the same lines in the source code. This scenario often occurs in practice in the presence of loops or recursive calls. For example, if the initial value of a were 100 as opposed to 7, there would be 100 causes that map to the same lines $\{5,8,3,1\}$.

For such cases, CaFL provides an option for the user to skip analyzing the continuous iterations. Specifically, for every branch instance s_{br} in the given execution trace, CaFL checks whether s_{br} initiates a loop, and marks the first three iterations. Although the weakest precondition computation is performed in all loop iterations, only the marked iterations will be used to propagate the critical conditions. Our experiments shows that this heuristic is effective in practice, and furthermore, it does not lead to any root cause being missed in our benchmark programs.

Table III. Experimental Results on Applying CaFl to the Buggy TCAS Programs from the Siemens Suite

		TC#							TC#							TC#				
Name	E#	All	Det	C%	L	Time (s)	Name	E#	All	Det	C%	L	Time (s)	Name	E#	All	Det	C%	L	Time (s)
v1	1	131	131	8.2	2	0.058	v15	1	10	10	6.6	2	0.051	v29	1	18	18	12.7	2	0.059
v2	1	69	69	12.2	3	0.055	v16	1	70	70	11.2	3	0.054	v30	1	58	58	12.4	3	0.057
v3	1	23	23	7.3	2	0.057	v17	1	35	35	11.3	3	0.060	v31	3	14	14	5.2	2	0.049
v4	1	20	20	8.2	2	0.055	v18	1	29	29	11.3	3	0.060	v32	3	2	2	3.1	2	0.055
v5	1	10	10	8.2	2	0.054	v19	1	19	19	11.3	3	0.061	v33	4	89	89	6.7	1	0.011
v6	1	12	12	7.5	2	0.053	v20	1	18	18	13.2	3	0.051	v34	1	77	77	6.4	2	0.050
v7	1	36	36	13.4	3	0.058	v21	1	16	16	12.2	3	0.054	v35	1	76	76	11.5	3	0.057
v8	1	1	1	13.7	3	0.052	v22	1	11	11	10.4	3	0.057	v36	1	120	120	1.1	1	0.065
v9	1	7	7	14.6	3	0.057	v23	1	41	41	10.5	3	0.053	v37	1	93	93	10.2	3	0.063
v10	2	14	14	7.9	2	0.050	v24	1	7	7	10.8	3	0.059	v38	1	91	91	6.7	1	0.012
v11	3	14	14	9.4	3	0.042	v25	1	3	3	5.4	2	0.065	v39	1	3	3	4.9	2	0.064
v12	1	70	70	8.2	2	0.054	v26	1	11	11	7.5	2	0.053	v40	2	120	120	3.4	2	0.062
v13	1	4	4	7.5	2	0.055	v27	1	10	10	6.7	2	0.053	v41	1	20	20	5.8	2	0.053
v14	1	50	50	5.0	2	0.031	v28	1	76	76	12.4	2	0.054	-	-	-	-	-	-	-

5. EVALUATION

We have implemented our method in a software tool built upon the LLVM compiler [Lattner 2002] and the KLEE symbolic virtual machine [Cadar et al. 2008]. To compute the UNSAT cores from unsatisfiable logic formulas, we have replaced KLEE's default constraint solver STP [Ganesh and Dill 2007] with Yices [YIC]. Our tool, called CaFL, first transforms the C/C++ program to LLVM bytecode, and then leverages KLEE to discover new faulty executions or replay a known faulty execution, before conducting the cascade analysis.

We have evaluated CaFL on two sets of benchmark programs. The first set comes from the Siemens suite [Do et al. 2005], which has been widely used in previous fault localization studies [Griesmayer et al. 2007; Renieris and Reiss 2003]. The second set consists of real-world Linux applications from Busybox [Bus] and GNU Coreutils [Cor]. All experiments were performed on a computer with a 2.66GHz Intel dual core CPU and 4GB RAM.

5.1. Experiments on the Siemens Suite

We first present our experimental results on variants of a program called TCAS, and then present our results on the other six programs in the Siemens suite.

Results on TCAS programs. TCAS is a model of an aircraft collision detection system, which continuously monitors the radar information for any potential collision. TCAS is a small and yet relatively complex program with 143 lines of code. There are 41 buggy versions of TCAS with 1600 test inputs that can trigger a failure. For the purpose of experimental evaluation, we manually compared each buggy version with the correct version and marked the set of different statements as the root cause. In addition, we instrumented every buggy version with a statement that asserts its output to be same as the output of the correct version. Thus, during the experimental evaluation of our methods, an execution on a buggy version always terminates with an assertion failure.

The experimental results on all 41 versions of TCAS are given in Table III. The first two columns show the name and the number of incorrect statements in the program. The next two columns, under the label *TC#*, show the the number of all failing test runs and the number of test runs on which CaFL was able to generate the correct root cause. When a test run contains multiple buggy statements, we consider CaFL to be accurate only when it localized all the executed buggy statements. Column *C%* shows,

Table IV. Comparing CaFL with BugAssist [Jose and Majumdar 2011a] on all Buggy TCAS Programs

	Line#				Line#				Line#		
Name	CaFL	BugA	Ratio	Name	CaFL	BugA	Ratio	Name	CaFL	BugA	Ratio
v1	6	11	0.55	v15	11	13	0.85	v29	10	12	0.83
v2	11	13	0.85	v16	8	8	1.00	v30	9	9	1.00
v3	15	17	0.85	v17	7	4	1.75	v31	5	7	0.71
v4	6	12	0.50	v18	7	4	1.75	v32	4	10	0.40
v5	15	14	1.07	v19	7	4	1.75	v33	1	1	1.00
v6	12	5	2.40	v20	6	13	0.46	v34	5	14	0.36
v7	11	5	2.20	v21	14	12	1.17	v35	9	9	1.00
v8	8	13	0.62	v22	15	23	0.65	v36	2	14	0.14
v9	10	10	1.00	v23	9	8	1.13	v37	7	3	2.33
v10	12	6	2.00	v24	10	14	0.71	v38	5	2	2.50
v11	7	9	0.78	v25	7	12	0.58	v39	6	3	2.00
v12	15	14	1.07	v26	11	14	0.79	v40	4	15	0.27
v13	12	14	0.86	v27	11	14	0.79	v41	6	12	0.50
v14	3	1	3.00	v28	10	9	1.11	**Avg**	**8.51**	**9.93**	**0.86**

on average, the percentage of instances in the given trace that were included in cause tree. Column *L* shows the level of the root cause reported by CaFL. Column *Time(s)* shows the average time take by CaFL to analyze a faulty execution trace.

For all the buggy versions of TCAS, CaFL was able to catch the root causes and the execution time was small. This is the case even for *v11*, which has a *missing code error*, meaning that some necessary program statements are missing from the program. CaFL was able to find the root cause because there was only a partial statement missing, that is, the buggy statement was (*if (A)* when it should have been *if (A&& B)*).

Comparing CaFL with BugAssist. In table IV, we compare CaFL with BugAssist [Jose and Majumdar 2011a, 2011b] on all 41 buggy versions of TCAS. We downloaded the executable of BugAssist from its website. For each buggy TCAS version, we compare the number of source code lines explored by CaFL and by BugAssist when the root cause is identified. We also compare these numbers in the column labeled *Ratio*, where a ratio less than 1 means that CaFL identified fewer statements than BugAssist. On average, the number of statements identified by CaFL is 0.86 times of the number of statements identified by BugAssist.

It is worth noting that comparing the number of statements identified by the two methods is only part of the story, since BugAssist was designed to return a most likely cause for the given failure, whereas CaFL was designed to explore all possible causes. Therefore, it is possible for BugAssist to miss the real bug. Indeed, we have observed a couple of cases in our experiments on real-world applications from the GNU Coreutils. Unfortunately, most of the applicatoins other than *TCAS* could not be handled by BugAssist due to limitations of its C/C++ front-end.

Results on all Siemens programs. We also evaluated CaFL on the other six programs in the Siemens suite. The results are shown in Table V. In addition to *TCAS*, *schedule2* and *schedule* are two priority schedulers, *totinfo* is a program that computes statistics of the given data sets, *printtokens* and *printtokens2* are two lexical analyzers, and *replace* is a program that performs pattern matching and substitution. Since these are small programs, our goal is to evaluate the accuracy and effectiveness of CaFL in localizing the root cause as opposed to the execution time. We say that CaFL is accurate if the root cause identified by the user is included in the cause tree computed by CaFL. We say that CaFL is effective if the reported cause tree is small.

Table V. Comparing CaFL with Dynamic Slicing (DS) [Korel and Laski 1988] and Relevance Slicing (RS) [Gyimóthy et al. 1999] on All Programs in the Siemens Suite

Name	LoC	Trace#	V#	DS			RS			CaFL			
				DV#	C%	Time(s)	DV#	C%	Time(s)	DV#	C%	Time(s)	Level
tcas	143	276	40	40	56.2	0.01	40	88.5	0.01	40	8.7	0.04	2
schedule2	564	8327	3	2	44.3	1.54	3	87.3	1.56	3	9.6	4.52	2
schedule	374	7623	3	2	61.3	2.11	3	75.3	2.25	3	4.2	6.12	2
totinfo	565	4377	6	4	46.2	3.07	6	81.3	3.11	6	10.5	5.73	2
printtokens2	523	5094	7	4	27.0	2.15	7	73.7	2.38	7	4.8	13.71	3
printtokens	726	3469	5	4	58.5	1.12	5	81.5	1.44	5	3.0	5.11	2
replace	512	12458	6	4	46.2	2.47	6	88.2	3.69	6	7.6	7.87	2
Average	487	5946	10	**8.6**	**48.5**	1.78	**10**	**82.3**	2.06	**10**	**6.9**	6.16	2.1

```
$ busybox arp -Ainet
$ busybox tr [
$ busybox top d
$ busybox printf %Lu
$ busybox ls -co
$ busybox install -m
```

(a) Busybox

```
$ paste -d\\ abcdefghijklmnopqrstuvwxyz
$ mkdir -Z a b
$ mkfifo -Z a b
$ mknod -Z a b p
$ ptx x t4.txt
$ seq -f %0 1
```

(b) Coreutils

Fig. 9. Commands for triggering crashes of applications in Busybox and GNU Coreutils, where the content of *t4.txt* is "a".

Toward this end, we compared the result of CaFL with the results of two related methods: dynamic slicing [Korel and Laski 1988] and relevant slicing [Gyimóthy et al. 1999]. Columns 1–4 shows the name of the program, the number of lines of code, the average length of the faulty execution trace, and the total number of buggy versions of the program analyzed. The remaining columns show the result of dynamic slicing (*DS*), relevant slicing (*RS*), and CaFL, respectively. For each method, we show the number of versions for which root causes are identified (*DV#*), the percentage of instruction instances localized (*C%*), and the execution time in seconds. CaFL has an additional sub-column named *Level*, showing the the level of the root cause found in the cause tree.

Overall, dynamic slicing missed 9 real root causes and retained 48.5% of the instruction instances in the given trace. Furthermore, all the 9 undetected root causes are *execution omission errors* [Zhang et al. 2007], in which certain statements are mistakenly omitted during execution. Relevant slicing addressed this issue by adding the implicit dependency in its analysis. As a result, relevant slicing was able to capture all the root causes missed by dynamic slicing. However, on average, it retained 82.3% of instruction instances, which means that it is less effective in pruning away the redundant instances. In contrast, CaFL successfully detected all the root causes, and at the same time, retained only 6.9% of the instruction instances in the given trace—this is a significant improvement over DS and RS.

5.2. Experiments on Busybox and Coreutils

We now present our experimental results on a set of buggy applications from Busybox [Bus] and GNU Coreutils [Cor]. Busybox is the de-facto standard implementation of Embedded Linux for networking devices such as the wireless routers, which bundles many standard Linux utilities into a single executable. GNU Coreutils has a total of 72.1K lines of C code, which implements some of the most frequently used Linux commands such as *ls*, *mkdir*, and *top*.

```
1 const struct hwtype *get_hwtype (const char *name) {
2   const struct hwtype * const *hwp;
3   hwp = hwtypes;
4   while (*hwp != NULL) {
5     assert(name != NULL);
6     if (!strcmp((*hwp)->name, name)) // crash point
7       return(*hwp);
8     hwp++;
9   }
10  return NULL;
11}

446 int arp_main(int argc, char **argv) {
      ...
      // set mask in option_mask32
469   getopt32(argc, argv, "A:p:H:t:i:adnDsv", &protocol, &protocol, &hw_type,
               &hw_type, &device);
      ...
477   if (option_mask32 & ARP_OPT_A || option_mask32 & ARP_OPT_p) {  //error
478     hw = get_hwtype(hw_type);
      ...
```

Fig. 10. Code snippet of the buggy *arp* in Busybox, where the bug is in the conditional expression at Line 477.

In this experiment, we used twelve randomly chosen buggy applications from Busybox version 1.4.2 and GNU Coreutils version 6.10. Figure 9 shows the commands and command-line arguments that we used to reproduce the faulty executions. In the remainder of this subsection, we will first present the details of two case studies on *arp* from Busybox and *mkdir* from GNU Coreutils, and then present our results on the other applications.

Case study on arp. The *arp* utility manages the Linux kernel's network neighbor cache. It may add entries to or delete entries from the cache, or display the current content. Using the command *'busybox arp -Ainet'* we can reproduce a bug in the *arp* implementation that crashes at Line 6 of the code shown in Figure 10. In this experiment, we manually added an assertion *assert(name != NULL)* before the crash point. The root cause of the failure is at Line 477. Before using *hw_type* at Line 478, the mask of hardware type (*ARP_OPT_H*), instead of the mask of address family (*ARP_OPT_A*), should be checked. That is, the correct statement should be *if (option_mask32 & ARP_OPT_H || option_mask32 & ARP_OPT_t)*. Since the command line does not provide the *H* option or the *t* option, *hw_type* is set to NULL in *getopt32* at Line 469. Through parameter passing, *name* in *get_hwtype* obtains the value *NULL*, which leads to the failure at Line 6 as the string comparison function *strcmp* does not expect a NULL parameter.

We illustrate how CaFL can help the user identify the root cause of the failure in *arp*. The first generated cause contains Lines 5 and 478, which state that *(name!=NULL)* conflicts with the actual argument *hw_type*. The value of *hw_type* is actually *NULL*. After checking this cause, the programmer realizes that something is wrong with the actual argument, but the code at Line 478 itself is correct. Based on this information, CaFL identifies *(option_mask32&ARP_OPT_A)* at Line 477 as a critical condition, since it controls whether Line 478 will be executed. Indeed, the correct execution would follow if the *else*-branch had neither the option *H* nor the option *t* provided.

Starting from Line 477, our cascade analysis performs a new weakest precondition computation on the predicate *(option_mask32 & ARP_OPT_A==FALSE)*. The second level cause computed by CaFL contains Line 477, Line 469, and some omitted assignments to *option_mask32* in the function *getopt32*, which set the mask of address family instead of hardware type. By studying Lines 477 and 469 side-by-side, a user with sufficient knowledge of the code can identify the bug within the cause. During this

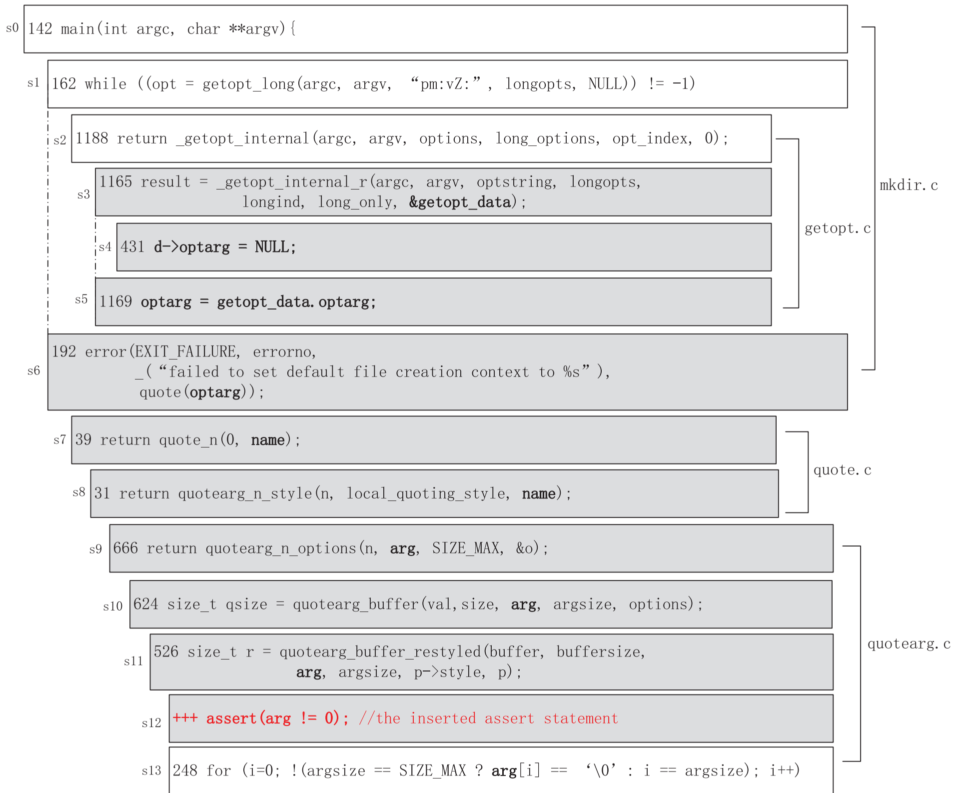

Fig. 11. The root cause of the failure in *mkdir* under the command'mkdir -Z a b'.

debugging process, CaFL can significantly reduce the amount of software code that the user has to inspect. Although the faulty trace has a total of 32,479 instruction instances, the user only needs to inspect 28 instruction instances (0.086%).

Case study on mkdir. The *mkdir* application from GNU Coreutils is meant to be used for creating new directories. In our experiment, the buggy version encountered a crash under the command *'mkdir -Z a b'*. By using KLEE, we were able to reproduce this crash and analyze the faulty execution trace. We found that the first cause computed by CaFL was the root cause. However, this cause is somewhat complex and not easy to manually identify since it spans across nine functions from four disk files. We organize the corresponding statements in the ladder shape, shown in Figure 11.

The *assert(arg != 0)* statement inserted before Line 248 of file *quotearg.c* represents the condition under which the crash occurs. The statements contained in the root cause are $\langle s3, s4, s5, s6, s7, s8, s9, s10, s11, s12\rangle$, and the bold expressions are responsible for propagating the error value. We explain how the root cause leads to the failure as follows.

—Step 1, *s3* passes *getopt_data* as a parameter to the function *_getopt_internal_r*, which then sets the field *optarg* to *NULL* at *s4*. Then, the global variabe *optarg* is set to *getopt_data.optarg*, which is *NULL* at the moment.

—Step 2, *s6* invokes the function *quote* with *optarg* as a parameter, which eventually is propagated to the variabe *arg* at the statement *s12* through

Table VI. Applying CaFL to Applications in Busybox and GNU Coreutils with Different Optimization Heuristics

Name	Trace#				Line#				C%			
	Original	After_C	After_S	After_CS	None	C+L	SO+SL	All	None	C+L	SO+SL	All
arp	32,405	7,399	9,215	5,505	-	102	124	102	-	3.19	7.23	3.19
tr	15,993	10,828	4,739	3,857	36	24	36	24	5.11	3.58	5.11	3.58
top	32,127	7,719	9,111	5,968	-	140	158	140	-	1.73	9.26	1.73
printf	9,150	5,650	5,362	4,166	96	72	96	72	6.52	2.13	6.52	2.13
ls	35,769	8,462	11,724	6,186	-	91	123	91	-	3.24	5.42	3.24
install	31,999	7,591	8,856	5,692	-	82	115	82	-	2.02	8.32	2.02
paste	10,183	5,159	5,816	3,535	68	53	68	53	11.68	4.55	11.68	4.55
mkdir	13,492	7,504	7,539	5,469	77	69	77	69	14.72	4.85	14.72	4.85
mkfifo	12,854	6,896	6,971	4,931	74	64	74	64	16.66	6.33	16.66	6.33
mknod	13,176	7,082	7,026	4,986	75	67	75	67	17.08	5.86	17.08	5.86
ptx	130,951	68,036	49,467	45,630	-	-	387	354	-	-	10.27	7.95
seq	11,719	5,583	6,044	3,928	97	75	97	75	14.29	6.54	14.29	6.54

parameter passing of *quote_n*, *quotearg_n_style*, *quotearg_n_options*, *quotearg_buffer*, and *quotearg_buffer_restyled*.
—Step 3, the assertion at *s12* fails because the value of *arg* is *NULL*.

The actual faulty statement is *s6*, which should invoke function *quote* with the parameter *scontext* instead of *optarg*.[2] In addition to *s6*, the root cause identified by CaFL also contains nine other statements, which are necessary for the user to understand why the failure occurs.

Results on all Linux applications. Table VI shows our experimental results on all twelve buggy applications from Busybox and Coreutils. In this experiment, our goal is to evaluate the effectiveness of CaFL with various optimization heuristics. We have compared the following four variants.

—*None*: no optimization.
—$C + L$: with designated ***C***orrect functions and ***L***oop recognition optimizations. In other words, functions in the standard C libraries such as *strcmp* are designated as correct.
—$SO + SL$: with ***SO***lving performance enhancement and dynamic ***SL***icing.
—*All*: with all the optimizations described in Section 4.

We combine optimization C with L because they are designed for reducing the size of the cause tree. We combine SO with SL because, although they do not change the size of the cause tree, they may speed up the cause tree computation.

As shown in Table VI, the four columns under *Trace#* show the statistics of the faulty execution traces, including the actual length of the trace, the length after excluding the correct functions, the length after slicing, and the length after applying both simplifications. The four columns under *Line#* compare, under different optimization heuristics, the number of source code lines reported by CaFL as responsible for the given failure. The four columns under *C%* compare the percentage of the given trace reported by CaFL as responsible for the given failure. In all cases, the symbol $'-'$ means that our experiment timed out after the one-hour limit.

We manually examined the buggy programs and found that CaFL was able to identify the root cause in all benchmark examples as long as it is allowed to run into completion. In order to compare the efficiency and effectiveness of the different optimizations, we

[2]http://lists.gnu.org/archive/html/bug-coreutils/2008-03/msg00189.html.

Table VII. Comparing the Execution Time of Different Optimizations in CaFL on Busybox and Coreutils

Name	None			C+L			SO+SL			All		
	WP (s)	SMT (s)	Total(s)	WP (s)	SMT (s)	Total(s)	WP (s)	SMT (s)	Total(s)	WP (s)	SMT (s)	Total (s)
arp	–	–	>3600	4.1	26.0	30.9	3.6	28.0	32.0	1.7	9.7	11.9
tr	9.1	1146.8	1156.0	8.1	16.7	25.2	3.6	1.7	5.6	1.8	0.2	2.2
top	–	–	>3600	3.2	62.1	65.6	4.9	50.1	55.2	3.2	33.2	36.8
printf	5.0	440.2	445.9	2.1	4.5	6.9	1.1	4.3	5.8	0.8	0.2	1.2
ls	–	–	>3600	5.6	48.5	54.9	4.1	39.7	44.2	3.0	17.6	20.7
install	–	–	>3600	6.9	29.5	37.2	3.1	1.0	4.5	2.3	0.1	2.7
paste	7.1	812.2	820.6	4.8	11.2	16.7	3.5	1.2	5.3	0.8	0.1	1.1
mkdir	5.3	485.2	494.4	2.3	18.2	21.4	2.1	5.9	8.9	0.7	0.1	1.0
mkfifo	6.7	512.8	520.6	2.1	20.3	24.3	1.8	6.2	8.5	0.8	0.1	1.0
mknod	6.2	555.1	562.5	2.1	20.4	24.5	1.9	6.3	8.7	0.5	0.1	1.0
ptx	–	–	>3600	–	–	>3600	107.4	158.3	266.5	34.1	79.3	144.2
seq	10.2	1147.2	1158.8	2.2	3.1	5.7	1.7	5.7	7.9	1.3	0.2	1.9
Avg.	**>1504**	**>1924**	**>1929**	**>303**	**>321**	**>326**	**11.6**	**25.7**	**37.8**	**4.3**	**11.7**	**18.8**

forced CaFL to continue the computation of the cause tree even after the root cause has been discovered. Our results show that for *Line#* and *C%*, the variants *None* and $SO + SL$ always return the same results. Similarly, the variants $C + L$ and *All* always return the same results. This is consistent with our expectation, as $C + L$ is designed for reducing the size of the cause tree, whereas $SO + SL$ is designed for speeding up the computation only. Furthermore, we found that for this set of benchmark examples, less than one percent of the instruction instances in the faulty execution trace were explored by CaFL before the root causes was identified, highlighting the effectiveness of CaFL in localizing the bugs.

Table VII shows the results of comparing the efficiency of the four optimizations in terms of the execution time. For each optimization method, we report the following data:

—*WP*. as the time (in seconds) spent on weakest precondition computation;
—*SMT*. as the time (in seconds) spent on the SMT solver;
—*Total*. as the overall time (in seconds) spent on the entire cascade analysis.

The last row shows the average time for each step.

The results show that, in all cases, CaFL was able to complete the cascade analysis within 20 seconds if all heuristic optimizations are used. However, without using $SO + SL$, CaFL would time out after one hour on *ptx* and the average time taken by the analysis would rise to >326 seconds. Without using any of the heuristic optimizations, CaFL would time out on five of the twelve benchmark programs. A closer look at the data revealed that, for the timed-out cases, most of the time were spent on constraint solving inside the SMT solver.

Table VIII compares the number of logical constraints fed to the SMT solver under different optimizations. The results show that the optimization $C + L$ was effective in pruning away the redundant constraints. Furthermore, since many of the remaining constraints did not intersect with each other, the optimization *SO* was able to avoid a large number of solver calls. This proved to be particularly important for *ptx*, which otherwise could not be analyzed by CaFL within the one-hour time limit.

6. LIMITATIONS

Although CaFL is able to identify root cause within a reasonable amount of time for all the evaluated cases in the Siemens suite, Busybox, and Coreutils, there are still

Table VIII. The Number of Logical Constraints Fed to the SMT Solver under Different Optimizations

Name	None	C+L	SO+SL	All
arp	-	65,026	9,601	1736
tr	5,813,789	71,480	842	252
top	-	182,913	4,481	2,188
printf	884,241	13,793	6,628	286
ls	-	126,395	17,171	1194
install	-	86,290	3,721	794
paste	2,888,436	80,404	1,141	576
mkdir	1,997,815	78,701	6,229	944
mkfifo	1,988,116	79,936	6,448	950
mknod	2,072,425	90,228	6,692	1018
ptx	-	-	52,063	28,705
seq	6,251,612	132,392	6,263	686

several limitations. First, for programs with complex control dependency and long traces, the cause trees computed by our method may be very large. In this case, more aggressive heuristics may need to used to prune the cause tree, even if such pruning in principle is unsound, for instance, it may lead to missed root causes. Although we do not expect fault localization to be fully automated for general applications, such unsound reduction—which aggressively prunes away causes without the user's involvement—may still be useful in practical settings. Second, CaFL concentrates only on one failing execution, which means that sometimes the *critical conditions* and related causes produced by CaFL cannot fully explain the observed failure. An example of this is handling *missed code errors*. Although we have shown through experiments that CaFL does better in identifying such errors than existing methods, there is still room for improvement. Finally, frequent use of SMT solving on large formulas can become a performance bottleneck. Although we have developed several effective heuristics to mitigate the problem, it remains an issue when the input trace become even larger. For future work, we plan to design a compositional analysis framework to further reduce the runtime overhead of SMT solvers, similar to the techniques used in [Lee et al. 2011; 2009].

7. RELATED WORK

There is a large body of work on automatically localizing the faulty statements in a piece of software code. A method closely related to ours is BugAssist [Jose and Majumdar 2011a, 2011b], which first encodes the faulty execution trace into an unsatisfiable *extended trace formula*, and then uses a MAX-SAT solver to find a maximal subset of statements that remain unchanged for the program to become correct under the given input. It reports the complement set to the user as a likely cause of the manifested failure. However, BugAssist does not attempt to compute all causes as in our method, nor does it report additional information to highlight the causal relationships between these causes.

Ermis et al. [2012] introduce the concept of *error invariants* to explain why certain portions of a faulty trace are irrelevant. The method is later improved by Christ et al. [2013], who introduce the concept of *flow-sensitive trace formulas* to explain how relevant statements can be reached in the faulty trace. In addition, Murali et al. [2014] proposes another interpolants-based method, which relies on the minimal unsatisfiable core for computing a sound and minimal slice for the identified error trace. However, neither of these two methods systematically generates all possible causes as in our

method. Furthermore, these methods rely on Craig's interpolants [Craig 1957], which requires a specialized SMT solver to compute. In contrast, our method uses a combination of weakest precondition and UNSAT core computations, which are more broadly supported by existing SAT and SMT solvers. Weakest precondition computation has also been used in Wang et al. [2006], but the method can only report a single cause, which may not always be the root cause.

Dynamic slicing [Korel and Laski 1988] is an inexpensive and widely used error triaging technique. However, the resulted slice still consists of a large number of instances. Furthermore, Dynamic slicing cannot locate *execution omission errors*. In order to address this problem, relevant slicing [Gyimóthy et al. 1999] has been proposed to extend dynamic slicing, by considering the *potential* influence between program statements. Unfortunately, as a result of this extension, the resulted slice becomes even larger. Zhang et al. [2007] introduce the concept of *implicit dependences* to eliminate unnecessary dependence through guided reexecutions of the program. However, the result is still not precise enough for practical use. Another problem of these slicing techniques is that, the resulted slice is often presented to the programmer as a single piece of information without any structure. In contrast, our new method produces a tree of causes to highlight their relationships. The difference between navigating through our cause tree and navigating through the raw data is analogous to reviewing 10 well-structured functions, each with 100 lines of code, versus reviewing a monolithic function with 1000 lines of code.

Some bug triaging methods are based on comparing the passing and failing executions [Groce et al. 2006; Groce and Visser 2003; Ball et al. 2003; Renieris and Reiss 2003; Qi et al. 2009; Befrouei et al. 2014]. For a given failing execution, they first find passing executions that are the most similar to the failing one. Then, they present the deviation of the failing executions from the passing executions as an explanation of the failure. For instance, Groce et al. [2006], Groce and Visser [2003], and Groce et al. [2004] use distance metrics of program executions to find minimal abstractions of erroneous traces. Methods based on the use of likely program invariants [Pytlik et al. 2003; Sahoo et al. 2013] leverage the invariants to identify the differences between failing and passing executions. They typically infer the invariants from a set of successful executions, and use the inferred invariants to analyze the failing executions. The invariants violated by a faulty execution are candidates for the failure's root cause, which will be trimmed further using dynamic slicing and other filtering heuristics. However, the effectiveness of these techniques is heavily dependent on the test suite quality, such as the number and coverage of the tests.

Zeller et al. propose delta debugging [Zeller 2002; Cleve and Zeller 2005], which is a fully automated method for isolating the relevant input variables and input values of a failing execution. This is a trial-and-error based technique that systematically narrows the state difference between a passing run and a failing run. It often needs to run the program a number of times before finding the difference. Rößler et al. [2012] propose a fully automated fault localization method based on test input generation, which keeps generating additional execution traces to guide the systematic isolation of failure causes. Both methods are based on the trial and error approach, which often requires generating and execution a large number of test runs. Furthermore, they can only discover the *correlation* between the anomalous events encountered during the test runs and the failure, which are not necessarily the *causal relationship*. The techniques of delta debugging has also been integrated with symbolic analysis to explain failed regression tests faults [Yi et al. 2015]. The synergistic approach uses previously correct version as the golden model to identify the root cause to newly introduced bugs during software updates.

There is also a large body of work on automated program repair [Griesmayer et al. 2006, 2007; Balakrishnan and Ganai 2008; Zeller 2002; Zhang et al. 2006; Liu and Li 2010], which are even more ambitious in that they aim at automatically fixing bugs. Typically, auxiliary variables, referred to as *selectors*, are introduced to control the behavior of certain program statements and branch conditions. Through constraint solver based analysis of the modified program, the values of these selector variables are identified, which represent a set of statements that need to be modified to prevent the observed failure. In addition, automated predicate switching [Zhang et al. 2006] can be used at runtime to modifying the control flow, with the hope of leading the program to a successful run. These methods differ from ours in that they focus on solving a closely related, but different, problem. We believe that the current techniques for automated program repair is still in the early development stage without knowing the programmer's design intent, they tend to rely on certain known error patterns and often *dodge* the erroneous code rather than fix them.

8. CONCLUSIONS

We have presented a cascade analysis method for fault localization, which can systematically generate potential causes of an observable failure to help the user identify the root cause. Each cause is augmented with a proper context to illustrate its origin in the source code, and all causes are organized into a tree structure to highlight their causal relationships. We have implemented our method in a software tool and conducted experiments on a large set of public benchmarks, including programs from the Siemens suite, Busybox, and GNU Coreutils. Our experimental results show that CaFL is both accurate and effective in localizing the root causes for failures in real applications.

REFERENCES

Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 246–256.

Gogul Balakrishnan and Malay Ganai. 2008. PED: Proof-guided error diagnosis by triangulation of program error causes. In *Proceedings of the International Conference on Software Engineering and Formal Methods*. 268–278.

Thomas Ball, Mayur Naik, and Sriram K. Rajamani. 2003. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 97–105.

Ansuman Banerjee, Abhik Roychoudhury, Johannes A. Harlie, and Zhenkai Liang. 2010. Golden implementation driven software debugging. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*. 177–186.

Mitra Tabaei Befrouei, Chao Wang, and Georg Weissenbacher. 2014. Abstraction and mining of traces to explain concurrency bugs. In *Proceedings of the International Conference on Runtime Verification*. 162–177.

Boris Beizer. 1990. *Software Testing Techniques*. Van Nostrand Reinhold Co., New York.

Busybox. http://busybox.net/.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 209–224.

Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. 2013. Flow-sensitive fault localization. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, vol. 7737, 189–208.

Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *Proceedings of the International Conference on Software Engineering*. 342–351.

Coreutils. http://www.gnu.org/software/coreutils/.

William Craig. 1957. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Logic* 22, 3, 269–285.

Edsger Wybe Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.

Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Eng.* 10, 4, 405–435.

Evren Ermis, Martin Schäf, and Thomas Wies. 2012. Error invariants. In *Proceedings of the International Symposium on Formal Methods*. Lecture Notes in Computer Science, vol. 7436, 187–201.

Vijay Ganesh and David L. Dill. 2007. A decision procedure for bit-vectors and arrays. In *Proceedings of the International Conference on Computer Aided Verification*. 519–531.

Andreas Griesmayer, Roderick Bloem, and Byron Cook. 2006. Repair of Boolean programs with an application to C. In *Proceedings of the International Conference on Computer Aided Verification*. 358–371.

Andreas Griesmayer, Stefan Staber, and Roderick Bloem. 2007. Automated Fault Localization for C Programs. *Electron. Notes Theor. Comput. Sci.* 174, 4, 95–111.

Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. 2006. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.* 8, 3, 229–247.

Alex Groce, Daniel Kroening, and Flavio Lerda. 2004. Understanding counterexamples with explain. In *Proceedings of the International Conference on Computer Aided Verification*. 453–456.

Alex Groce and Willem Visser. 2003. What went wrong: explaining counterexamples. In *Proceedings of the International SPIN Workshop on Model Checking Software*. 121–136.

Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. 1999. An efficient relevant slicing method for debugging. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*. 303–321.

Manu Jose and Rupak Majumdar. 2011a. Bug-Assist: Assisting fault localization in ANSI-C programs. In *Proceedings of the International Conference on Computer Aided Verification*. 504–509.

Manu Jose and Rupak Majumdar. 2011b. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 437–446.

B. Korel and J. Laski. 1988. Dynamic program slicing. *Inf. Process. Lett.* 29, 3, 155–163.

Chris Lattner. 2002. LLVM: An infrastructure for multi-stage optimization. Master's Thesis.

Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, and Zijiang Yang. 2011. Offline symbolic analysis to infer total store order. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*.

Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, Zijiang Yang, and Cristiano Pereira. 2009. Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 564–575.

Mark H. Liffiton and Karem A. Sakallah. 2008. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning* 40, 1, 1–33.

Yongmei Y. Liu and Bing Li. 2010. Automated program debugging via multiple predicate switching. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 327–332.

Inês Lynce and Joao Marques-Silva. 2004. On computing minimum unsatisfiable cores. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*.

Vijayaraghavan Murali, Nishant Sinha, Emina Torlak, and Satish Chandra. 2014. What gives? A hybrid algorithm for error trace explanation. In *Proceedings of the 6th International Conference on Verified Software: Theories, Tools and Experiments (VSTTE'14)*. 270–286.

Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. 2003. Automated fault localization using potential invariants. CoRR cs.SE/0310040 (2003).

Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. 2009. Darwin: An approach for debugging evolving programs. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*. 33–42.

Manos Renieris and Steven P. Reiss. 2003. Fault localization with nearest neighbor queries. In *Proceedings of the IEEE/ACM International Conference On Automated Software Engineering*. 30–39.

Jeremias RöBler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. 2012. Isolating failure causes through test case generation. In *Proceedings of the International Symposium on Software Testing and Analysis*. 309–319.

Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. 2013. Using likely invariants for automated software fault localization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 139–152.

Chao Wang, Zijiang Yang, Franjo Ivančić, and Aarti Gupta. 2006. Whodunit? Causal analysis for counterexamples. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis*. 82–95.

Mark Weiser. 1984. Program slicing. *IEEE Trans. Software Eng. SE-10*, 4, 352–357.

Qiuping Yi, Zijiang Yang, Jian Liu, Chen Zhao, and ChaoWang. 2015. A synergistic analysis method for explaining failed regression tests. In *Proceedings of the International Conference on Software Engineering (ICSE'15)*.

Yices: An SMT Solver. In http://yices.csl.com/.

Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*. 1–10.

Lintao Zhang and Sharad Malik. 2003. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'03)*. 880–885.

Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the International Conference on Software Engineering*. 272–281.

Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. 2007. Towards locating execution omission errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 415–424.

Received February 2014; revised December 2014, February 2015; accepted February 2015