Profile-Guided Program Simplification for Effective Testing and Analysis*

Lingxiao Jiang Zhendong Su
University of California, Davis
{jiangl,su}@cs.ucdavis.edu

ABSTRACT

Many testing and analysis techniques have been developed for in-house use. Although they are effective at discovering defects before a program is deployed, these techniques are often limited due to the complexity of real-world code and thus miss program faults. It will be the users of the program who eventually experience failures caused by the undetected faults. To take advantage of the large number of program runs carried by the users, recent work has proposed techniques to collect execution profiles from the users for developers to perform post-deployment failure analysis. However, in order to protect users' privacy and to reduce run-time overhead, such profiles are usually not detailed enough for the developers to identify or fix the root causes of the failures.

In this paper, we propose a novel approach to utilize user execution profiles for more effective in-house testing and analysis. Our key insight is that execution profiles for program failures can be used to simplify a program, while preserving its erroneous behavior. By simplifying a program and scaling down its complexity according to its profiles, in-house testing and analysis techniques can be performed more accurately and efficiently, and pragmatically program defects that occur more often and are (arguably) more relevant to users will be given preference during failure analysis. Specifically, we adapt *statistical debugging* on execution profiles to predict likely failure-related code and use a syntax-directed algorithm to trim failure-irrelevant code from a program, while preserving its erroneous behavior as much as possible. We conducted case studies on a testing engine, *CUTE*, and a software model checker, *BLAST*, to evaluate our technique. We used subject programs from the Aristotle Analysis System and the Software-artifact Infrastructure Repository (SIR). Our empirical results show that using simplified programs, CUTE and BLAST find more bugs with improved accuracy and performance: they were able to detect 20 and 21 (out of 139) more bugs respectively in about half of the time as they took on the original test programs.

*This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, US Air Force under grant FA9550-07-1-0532, and a generous gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-59593-995-1 ...$5.00.

Categories and Subject Descriptors: D.2.4/D.2.5 [**Software Engineering**]: Testing and Debugging, Software/Program Verification—*Reliability, Statistical methods, Debugging aids*

General Terms: Experimentation, Reliability

Keywords: Statistical Debugging, Profiling, Program Simplification, Testing and Analysis

1. INTRODUCTION

Many software defects remain undetected or unfixed even after it goes through extensive in-house testing and analysis due to code complexity and the time-to-market pressure. It will be the users of the program in the field who experience those latent bugs. One way to allay the problem is to develop more effective testing and analysis techniques so that fewer defects remain in deployed code.

However, no matter how sophisticated these techniques can be made, some defects will inevitably be undetected before experienced by users. For the program failures that happen in the field, feedback from the users can be very useful for isolating and fixing related bugs. Much work has proposed mechanisms to collect program execution profiles from users so that bug localization and failure analysis can be performed [32, 33, 35, 49]. However, for privacy and performance concerns, the profiles are often not detailed enough for developers to reconstruct complete scenarios that exhibit the failures. Thus, the usefulness of the profiles for root-cause analysis is usually limited, leaving an apparent gap between the needs of the developers and the contribution of the users.

This paper aims to reduce such a gap by further leveraging execution profiles to aid in-house testing and analysis. Our key insight is that in-house techniques and profile-based techniques can complement each other. The former are often good at constructing failure scenarios for detailed analysis, but lack *a priori* knowledge about bug locations. Thus, they usually search as many locations in a program as possible for bugs, which hinders their effectiveness. The latter are often good at localizing bugs, but not effective at providing detailed failure scenarios. By combining the two kinds of techniques, we can first *simplify a program* using information about possible bug locations derived from profiles, while preserving the erroneous behavior of the program, and then *apply in-house techniques on the simplified program* to construct detailed scenarios for *failures occurred in the field*. The profile-guided program simplification inspires new uses of profiles; more importantly, it helps to scale down code complexity and thus in-house techniques can be applied more effectively.

1.1 Motivating Examples

We use two code snippets to motivate our approach. They are extracted from the subject programs used in our study (*cf.* Section 4), and modified for a simpler presentation. In this paper, we consider two state-of-the-art tools for in-house testing and analysis: (1)

CUTE, a testing engine that combines both concrete and symbolic execution [16,43], and (2) BLAST, a software model checker based on predicate abstraction [21, 22]. We add explicit assertions in the code for CUTE and BLAST to know when a failure is reached.

Example 1. Consider the following `if` statement:

```
if ( inputr * inputc > 990 &&
     inputr * inputc <= 1000 )
   assert( FALSE );
```

The assertion may be violated if the product of the two input variables, `inputr` and `inputc`, falls within the range [991, 1000]. However, CUTE fails to construct concrete inputs that lead to the assertion violation and (*optimistically*, but *incorrectly*) reports the assertion is unreachable because the constraint solver in CUTE deduces that the path constraints involve multiplications and are unsolvable. Although BLAST's theorem prover cannot resolve the constraints either, it states its incapability and (*conservatively*, and correctly) reports that the program may have faults.[1]

Imagine if we can simplify the code to the following:

```
if ( TRUE ) assert( FALSE );
```

Now both CUTE and BLAST can easily decide that the simplified code can violate the assertion and thus has a fault. Such a simplification is possible because from the profiles of the original code we know that the program only fails when the `if` condition is true.[2]

Example 2. Now consider a slightly more complicated example:

```
if ( isalnum(input[i-1]) & /* bit operator */
     isalnum(input[i+1]) ) {
   ...
} else if ( isalnum(input[i-1]) ||
            isalnum(input[i+1]) )
   assert( FALSE );
```

The assertion is unreachable since the C library function `isalnum` always returns `0` for non-alphanumeric characters and the *same* non-zero value for alphanumeric characters. Thus, when

```
isalnum(input[i-1]) & isalnum(input[i+1])
```

is false, both `isalnum(input[i-1])` and `isalnum(input[i+1])` have to be `0`, and then the second `if` condition must be false.

Neither CUTE nor BLAST can resolve the path constraints for this program; while CUTE (again optimistically, but correctly) reports that the program is safe since it cannot construct concrete inputs to reach the assertion, BLAST (again conservatively, but incorrectly) reports a failure.

Suppose we can use profiles of the program to simplify it as the following (without introducing new erroneous behavior into or removing old ones from the original code):

```
if ( FALSE ) {
   ...
} else if ( FALSE )
   assert( FALSE );
```

The assertion here is clearly unreachable. Both CUTE and BLAST can easily decide that the code is safe.

[1] We note that *failures* (observable failed executions), *errors* (invalid internal states during an execution), and *faults* (the program phenomena causing errors and failures) have different meanings. In this paper, we use assertions to turn errors into failures and the assertions were added according to faults. In addition, a profile usually does not contain complete internal states, but the result of its execution (either success or failure), so throughout this paper we will mainly use the term *failure*.

[2] Although in this case our approach does not directly provide the concrete inputs that cause the assertion violation, it does provide the actual path that leads to the failure. Section 4.3 discusses more in detail how to interpret failure reports for simplified programs.

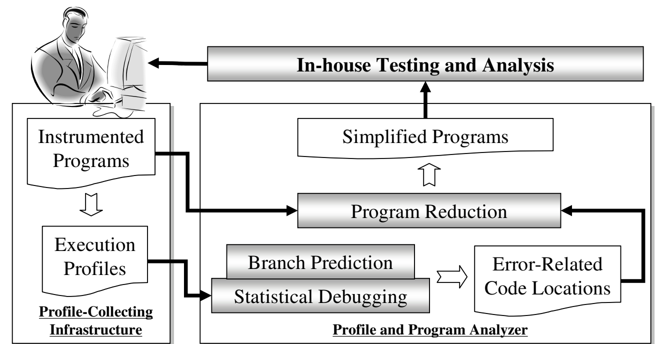

Figure 1: Work-flow of our approach.

1.2 Our Approach

This paper presents a novel, automatic technique to simplify a program w.r.t. a set of program execution profiles from users in the field. The goal is to make in-house testing and analysis more effective by scaling down code complexity and focusing on field failures occurred most often. It integrates profiles into existing in-house testing and analysis techniques and helps reduce the gap between the needs of developers and the contribution of users. Conceptually, our approach is comprised of three components: (1) a *profile-collecting infrastructure* that instruments a program before its deployment and collects execution profiles from users; (2) a *profile and program analyzer* that predicts potentially failure-related code and reduces failure-irrelevant code from the program; and (3) any *in-house testing or analysis technique* that can be applied to the simplified program for bug detection.

Figure 1 shows the work-flow of our approach. First, a program is instrumented and deployed to users; execution profiles will then be generated in the field and sent back to a central database for analysis upon users' agreement. Statistical debugging is applied on the profiles to localize the program locations that are likely failure-related; our profile analyzer also predict branching directions that likely lead to program failures. The details of these steps are described in related work [24, 32] and briefly summarized in Section 2. With the branching directions learned from the profiles, our program analyzer removes failure-irrelevant branches from the original program, but preserves program paths that may lead to the failures exposed in the profiles. The algorithm for simplifying programs and its failure-preserving property are described in detail in Section 3. Then, existing in-house testing and analysis techniques can be applied on the simplified program for further bug detection.

Two testing and analysis techniques, CUTE and BLAST, were used to validate the effectiveness of our approach. We performed the evaluation on subject programs from the Aristotle Analysis System [44] and the Software-artifact Infrastructure Repository (SIR, [15]). Our empirical studies show that both CUTE and BLAST can detect more bugs in simplified programs with improved accuracy and performance: they were able to detect respectively 20 and 21 more bugs (out of 139) for the programs, and took about half of the time as they took on the original programs.

1.3 Paper Organization

The rest of the paper is structured as follows. Section 2 gives necessary background on statistical debugging. Section 3 describes details of our approach and its failure-preserving property. In Section 4, we present our empirical evaluation based on CUTE and BLAST to show the effectiveness of our approach. Section 5 discusses related work, and Section 6 concludes.

2. BACKGROUND

This section provides necessary background on statistical debugging techniques that our approach is based on.

2.1 Infrastructure for Collecting Profiles

We first introduce the infrastructure for instrumenting programs and collecting execution profiles from users in the field. Our approach uses the Cooperative Bug Isolation project (CBI [30]). CBI lightly instruments a program with statically fixed n predicates, and an execution of the instrumented program is recorded as an n-dimensional vector, which is one profile, where the i-th value of the vector counts the number of times that the i-th predicate is observed to be true during the execution. The vector for each execution also has a label, indicating failure or success of the execution. A key observation from CBI and statistical debugging [24, 33, 35, 48] is that although it is practically impossible to recover complete program states, including user specific information, from *one* profile, a large collection of such profiles from *many* users can be useful for understanding the program's (mis)behavior.

CBI instruments the following predicates (called *branch predicates*) that are related to every branching conditions in a program and are used extensively in our profile and program analyzer[3]:

> Given a branching condition C, its then-branch and else-branch correspond to two predicates $C = true$ (*abbr.* C_t) and $C = false$ (*abbr.* C_f), respectively; each branch is thus instrumented to count how many times during an execution C_t and C_f are satisfied.

The instrumentation scheme in CBI is intended to be of little overhead for individual user, but informative for bug detection with profiles from many users.

2.2 Statistical Branch Prediction

Profiles from CBI have been used with different statistical methods to locate likely failure-related locations [24,32,33,35,49]. Such locations have also been used to construct static control flow paths to aid debugging [24, 29] and prioritization of failure reports [34]. We summarize the technique from our previous work [24] for predicting branching directions that may lead to program failures. We will utilize the predicted branching directions in this paper to simplify programs.

Given a conditional C in a program, the basic intuition for predicting the branching direction that leads to a program failure is to see how often the then-branch is taken (*i.e.*, C_t is satisfied) versus how often the else-branch is taken (*i.e.*, C_f is satisfied) in *failed* executions: if failures occur more often when C_t is satisfied, then we may decide the then-branch should be taken to lead to failures, and vice versa; if we cannot decide which branch is taken more often, we simply assume both branches are possible.

The actual mechanical process is as follows. First, we assign truth values over the range of {`True, False, Unknown`} for all branch predicates. Given a branch predicate p (either C_t or C_f): (1) If p is *never* satisfied in any *failure* profile, let the truth value of p be `False`; (2) If p is satisfied at least once in most (>50%) of the *failure* profiles, let the truth value of p be `True`; (3) Otherwise, let it be `Unknown`.

Second, we decide the branching direction for every conditional C based on the truth values of C_t and C_f: (1) If both C_t and C_f are `False`, C is likely not executed in any failed run, and all paths leading to C may be reduced. We say the branching direction for C is `Neither`; (2) If C_t is `True` or `Unknown` and C_f is `False`, the else-branch of C is likely not taken in failed runs, and we say the direction for C is `True`; (3) If C_t is `False` and C_f is `True` or `Unknown`, the then-branch of C is likely not taken in failed runs, and we say the direction for C is `False`; (4) If both C_t and C_f are not `False`, then both branches may be taken and we say the direction for C is `Both`. Table 1 illustrates the description.

[3]CBI also instruments other kinds of predicates, but we have not yet utilized those predicates in this paper.

Direction for C		Truth Value of C_t		
		`True`	`False`	`Unknown`
Truth Value of C_f	`True`	`Both`	`False`	`Both`
	`False`	`True`	`Neither`	`True`
	`Unknown`	`Both`	`False`	`Both`

Table 1: Predicting branching directions based on truth values of branch predicates.

The two-step process for predicting branching directions may also incorporate other information learned from the profiles. In previous work on statistical debugging, many statistical methods, including machine learning and clustering algorithms, have been applied to select branch predicates that are likely failure-relevant or close to actual bug locations, and empirical studies showed that the top-selected predicates are indeed useful for revealing actual bugs. Thus, it is reasonable to assign `True` to all branch predicates that are identified as failure-relevant during the first step [24].

Also, the process can be more conservative (*i.e.*, to preserve more branches by assigning `True` or `Unknown` to more branch predicates in our context) or more aggressive (*i.e.*, to reduce more branches by assigning `False` to more branch predicates even when the probability of the correctness of the assignment is low). An aggressive process may help to simplify a program more substantially and further improve the performance of subsequently applied testing and analysis techniques, with possibly more false negatives. On the other hand, simplified programs that are based on a conservative process may preserve all erroneous behavior in the original programs and more faults may be detected. Section 3.3 will give detailed account of the failure-preserving property of the above process, and Section 4 will present empirical data to show its effectiveness.

3. APPROACH DETAILS

This section presents our approach for profile-guided program simplification in detail.

3.1 Branch Reduction

The objective of our program simplification is to reduce as many branches as possible to be sequential, while preserving paths that may lead to program failures that have been exposed by some profile. The branch reduction is based on the branching directions learned from the profiles, as summarized in Section 2.2.

Algorithm 1 describes a simple syntax-directed method to reduce branches based on the branching directions. Given an abstract syntax tree of a program, the algorithm visits every `if` statement and modifies it according to its direction: (1) if the direction is `Neither`, which means the `if` statement was *never* reached in failed runs, and thus we can reduce all code following the `if`; (2) if the direction is `True`, which means failures likely happen when the then-branch is taken, and thus we can reduce the else-branch; and (3) it is similar for the `False` case.

Intuitively, the algorithm only removes failure-irrelevant branches. As long as the branching directions are failure-preserving (*cf.* Section 3.3), there must be a path in the simplified program that still leads to failures, and the faults in the original program can still be detected by in-house testing and analysis techniques.

3.2 Branch Validation

On the other hand, although the paths leading to failures are preserved during branch reduction, the semantics of the simplified program can be different from the original. Program inputs that do not cause failures may also fall on the failure paths since there are now fewer conditionals on the paths. Thus, subsequent testing and analysis may report more (false) failures than before.

Example 3. Consider the following code snippet:

Algorithm 1 REDUCE

Input: P: C program with n conditionals
$D = \langle d_1, \ldots, d_n \rangle$: directions for the conditionals
Output: P': P with certain branches reduced

```
for all node in P's AST do
  if node == IF(p_i, s_t, s_f) then
    if d_i == Neither then
      node ← CALL(exit(0));
    else if d_i == True then
      node ← IF(TRUE, s_t, s_f)
    else if d_i == False then
      node ← IF(FALSE, s_t, s_f)
    end if
  end if
end for
```

```
void f(struct list * p) {
  if ( p != NULL ) {
    p->next = ...
    assert ( .../* failure condition */ );
  }
}
```

Branch reduction may simplify it to the following:

```
void f(struct list * p) {
  if ( TRUE ) {
    p->next = ...
    assert ( .../* failure condition */ );
  }
}
```

Then, if the applied in-house technique, such as CUTE, needs to execute the code, the code may segmentation faults if `p` is `NULL`.

Example 4. Similarly, branch reduction may cause infinite loops in the simplified program before its failure points. Consider the following code for illustration:

```
if ( input[i]==32 )              if ( input[i]==32 )
   input[i] = 0;                    input[i] = 0;
done = FALSE;                    done = FALSE;
while ( ! done ) {               while ( ! done ) {
  if ( input[i]==36 )              if ( TRUE )
     input[i] = 32;                   input[i] = 32;
  else                             else
     done = TRUE;                     done = TRUE;
}                                }
assert( input[i]!=32 );          assert( input[i]!=32 );
```

The assertion in the code on the left may be violated only when the then-branch of the second `if` condition (`input[i]==36`) is taken. Suppose a set of branching directions helps to reduce the code on the left to the code on the right.[4] An infinite loop which is not possible in the original code appears in the simplified code. Such code may affect different testing and analysis techniques in different ways: (1) techniques relying on concrete executions, such as CUTE, will fail to terminate; and (2) techniques relying on static code analysis, such as BLAST, may decide the assertion is unreachable because of the infinite loop (and thus lead to a false negative), or decide `input[i]` is always 32 if the loop finishes and simply report an assertion violation.

[4] Since both of the then-branch and the else-branch of this `if` have to be taken in *failed* executions, the predicted branching directions used here are not failure-preserving, *i.e.*, not preserving the erroneous behavior of the code (*cf.* Section 3.3 for formal definitions).

Therefore, it will be beneficial to avoid unexpected results if we can enforce the validity of the reduced branches in simplified programs. For example, ensuring `p!=NULL` in **Example 3** is always true when the then-branch is taken would avoid unnecessary segmentation faults for CUTE. To achieve this in general, we can simply restore certain branch conditions back into the program to avoid invalid branching. However, it is obvious that such retrogression would compromise the aforementioned benefits of our approach.

Instead, our approach adds *assume predicates* into simplified programs. Assume predicates are commonly used to represent conditions that must hold before the code following it can be executed correctly. Their functionality is thus similar to that of function preconditions, and we use assume predicates (or just `assume`'s) in this paper as guards for branches.

Example 5. As an example, consider the following code:

```
assume( p != NULL );
if ( TRUE ) {
  p->next = ...
}
```

The `assume(p!=NULL)` ensures that `p!=NULL` must hold before `p->next` can be accessed.

Many techniques can utilize `assume`'s. Symbolic execution [13, 16, 43, 45, 46] is one: Given a path leading to an `assume`, the code along the path can be symbolically executed; then path constraints, including the assume predicate, may be collected to help decide the conditions under which the code following the `assume` can be executed. Model checking [5,21,40,45] is another: Program states that do not satisfy the `assume` can be pruned so that state exploration may be performed more efficiently.

Our approach does not stipulate how to interpret `assume`'s. As long as the subsequent testing or analysis technique can recognize assume predicates, it is free for the technique to interpret `assume`'s in ways that best fit the technique. For example, CUTE inherently uses a combination of concrete and symbolic execution to systematically explore as many paths in a program as possible, and it is natural for CUTE to terminate a concrete execution if the execution violates an `assume` and to generate new inputs that satisfy the `assume` to continue. For BLAST, the assume predicates provide easy hints to construct and refine abstract program states to explore. For other testing and analysis techniques, additional efforts may be required to enforce the assume predicates in order to enforce the validity of the reduced branches and take full advantage of the benefits of our approach. For example, a testing technique based on pure concrete execution may need to incorporate a symbolic execution engine to do so.

On the other hand, it may have little impact even if we do not enforce the branch validity because it may only cause few false positives and subsequent in-house testing and analysis may enjoy more performance gain (*cf.* Section 4 for empirical data).

The algorithm for adding assume predicates is essentially the same as Algorithm 1 except for the extra `assume`'s before every conditional (a la **Example 5**). We omit it here due to space constraints. We implemented both algorithms with CIL [39].

3.3 Failure-Preservation

When a program is simplified, it is naturally to require its possible erroneous behavior to be maintained so that subsequent testing or analysis can still detect its defects. In the following, we define such a property as failure-preservation in terms of failure paths.

Definition 3.1 (Program Prediction). Given a program P containing a set of conditionals C, a prediction D for P is a function that defines the branching direction for each conditional in C, *i.e.*,

$$\forall c \in C, \exists d \in \{\texttt{Neither}, \texttt{True}, \texttt{False}, \texttt{Both}\}, \text{such that } D(c) = d.$$

Definition 3.2 (Failure Preservation). Given a prediction D for a program P that contains a set of `asserts` A, $\text{REDUCE}(P, D)$ is *failure-preserving* for P, if $\forall a \in A$, the existence of a feasible path in P violating a implies the existence of a feasible path in $\text{REDUCE}(P, D)$ violating a. We say that D is an *failure-preserving prediction* w.r.t. all assertions in P.

Definition 3.1 requires a program prediction to provide a direction for every conditional in the program. The actual prediction used in our paper is provided by the two-step process summarized in Section 2.2.

Definition 3.2 is so strong that it leaves open the question of how to construct a failure-preserving D. Since our approach targets program faults that cause field failures that are exhibited in profiles, we focus on a weaker definition of failure-preservation that reflects our insight that we can use profiles to construct D.

Definition 3.3 (Failure Preservation w.r.t. Profiles). Given a set of profiles F of a program P containing a set of `asserts` A and a prediction D for P, $\text{REDUCE}(P, D)$ is *failure-preserving for P w.r.t. F*, if $\forall a \in A$, the existence of a failure profile in F exhibiting a violation of a implies the existence of a feasible path in $\text{REDUCE}(P, D)$ violating a. We say D is an *failure-preserving prediction* w.r.t. F.

The number of profiles in F can be arbitrary. The smaller the number is, the fewer faults the original program may have,[5] and the more branches may be reduced.

The question now is to derive a failure-preserving D from profiles F. Trivially, if a D preserves all branches that have appeared in any failure profile at least once, it must have preserved the failure paths taken by the failed executions that generate the profiles, and the D is failure-preserving.

In general, we can still easily construct a failure-preserving D with the help of *pivot branches*.

Definition 3.4 (Assertion Pivot Branch). A branch b, either a then- or else-branch, is a *pivot branch* for an assertion a in P, if b must be taken before a is violated.

Intuitively, pivot branches are the branches we must keep when simplifying a program, while non-pivot branches may be reduced at our choice. In the following, we describe a way to construct a **correlative prediction** w.r.t. a particular assertion in P:

Given a set of pivot branches B for an assertion a in P and a set of profiles F, we (1) identify a subset of *failure* profiles $F' \subseteq F$ such that $\forall f \in F'$, every $b \in B$ is taken in f; (2) assign `True` to the branch predicates that correspond to some $b \in B$ or some other branch that is taken at least once in F'; (3) assign `False` to the branch predicates that correspond to any other branch; then (4) decide all branching directions a la the second step in Section 2.2.

Such correlative predictions obviously preserve at least one path (if any) that leads to the violation of the particular assertion a since all branches taken in executions that violate a are preserved. Also, the more pivot branches exist for the same assertion, the smaller F' may be, and the more branches may be reduced, and the simpler the reduced code may be.

The definition of pivot branches and correlative predictions can be extended to a program containing a set of assertions, even when we do not know to which assertion each pivot branch corresponds.

Definition 3.5 (Program Pivot Branch). A branch b is a *pivot branch* for P if it must be taken before *some* assertion in P is violated.

[5] It may also be the case that the code containing faults is less frequently executed.

The construction of correlative predictions based on program pivot branches follows similarly: Given a set of program pivot branches B for P and a set of profiles F, we (1) identify a subset of *failure* profiles $F' \subseteq F$ such that $\forall f \in F'$, some $b \in B$ is taken in f; (2) assign `True` to the branch predicates that correspond to some $b \in B$ or some branch that is taken at least once in F'; and (3) assign `False` to the branch predicates that correspond to any other branch. The more assertions in P, the larger F' is, and the less branches may be reduced accordingly.

We devote the above discussion to correlative predictions because we easily have the following theorem that says a correlative prediction for a program is failure-preserving:

Theorem 3.6 (Correlation $\Longrightarrow$ Failure-Preservation). Given a set of program pivot branches B for P with a set of assertions A and a set of profiles F, the correlative prediction D constructed above is failure-preserving w.r.t. F if $\forall a \in A$, the existence of a failure profile in F exhibiting a violation of a implies the existence of a pivot branch of a in B.

The advantage of correlation against Definition 3.3 is that it is easily verifiable w.r.t. F (recall that CBI-based profiles record the number of times when each branch is taken, *cf.* Section 2.1). It is also a sufficient condition for a program prediction to be failure-preserving, although it is not necessary.

What is adjustable during the construction of correlative predictions is the number of profiles. As an extreme case, if there is no failure profile, there will be no need for pivot branches, and all branch predicates may be assigned `False`, which means an in-house testing or analysis technique should finish quickly and find no fault in the simplified programs. This is reasonable since it is unnecessary to test or analyze a program if it never fails in the field. Although the number of failure profiles does not affect the failure-preserving property of a correlative prediction, it may affect whether the program fault can be detected or not by particular in-house techniques. Section 4.2 will show empirical results and an example (**Example 9**) for this kind of situations.

3.4 Looking for Pivot Branches

Now, the problem of failure-preserving program simplification becomes searching for pivot branches. However, Definition 3.4 or Theorem 3.6 does not imply the existence or uniqueness of a pivot branch for an assertion; the concept of dominators in control flow graphs [2] is not helpful since it does not take program failures into account; static analysis techniques, such as Postmortem Symbolic Execution [38], may only have limited uses since they could only discover pivot branches for program failures of simple, clearly-defined characteristics (*e.g.*, `NULL`-pointer dereferences).

In this paper, identifying pivot branches becomes easy when we utilize statistical debugging [24,32,33,35,49]. Others' and our own experience [24] showed that the top-ranked program predicates ordered by statistical methods are often good indicators of bugs; the branch predicates among them are thus very likely to the branches that must be taken to exhibit the failures. Thus, we simply select the branches that are identified by statistical debugging as failure indicators as pivot branches. Then, by following the construction of correlative predictions, a prediction for P is easily made to be failure-preserving w.r.t. a set of profiles, under the assumption that statistical debugging provides us accurate pivot branches.

As an additional note, an assertion may not have a single pivot branch; it may be violated, for example, when *either* one branch *or* another is taken. This paper does not define a full treatment for such cases since the statistical debugging techniques applied in the paper only identify *individual* branches as failure indicators. Arumuga Nainar *et al.* [3] have proposed to use compound Boolean predicates (*e.g.*, with logic disjunction), instead of individual atomic

predicates, as failure indicators; our correlative predictions can be easily extended to incorporate such information. For example, if Nainar *et al.*'s technique indicates $b_1 \wedge b_2$ is a failure indicator, we can use *both* b_1 and b_2 as pivot branches for the *same* assertion. As another case, if Nainar *et al.*'s technique indicates $b_1 \vee b_2$ is a failure indicator, we can use *either* b_1 or b_2 as a pivot branch to perform program prediction and simplification separately (and get two simplified programs, each of which is still failure-preserving); as a more conservative alternative, we can treat *both* b_1 and b_2 as pivot branches, just as the case for $b_1 \wedge b_2$, to perform program prediction and simplification. The correlative prediction based on such a treatment remains failure-preserving, although fewer branches may be reduced. As a third alternative, we can view b_1 and b_2 separately as pivot branches for two *virtually different* assertions; then it is just like the situations where a program contains two bugs (Section 4.3 discusses more about multiple bugs in one program).

4. EMPIRICAL EVALUATION

In this section, we present our empirical studies to evaluate the effectiveness of our approach for improving the accuracy and performance of two testing and analysis techniques, CUTE and BLAST.

4.1 Experimental Setup

The platform for all our experiments in the paper is a Linux machine with a 2.4GHz Intel Xeon processor and 2GB of RAM.

4.1.1 Subject Programs

We use the HR variants of the Siemens Test Suite [23] from the Aristotle Analysis System [44], which are also contained in the Software-artifact Infrastructure Repository (SIR [15]). The suite contains 132 versions of seven programs. Each program has thousands of test cases and from zero to hundreds of failed runs. Some statistics on the source code can be found in Graves *et al.*'s study [17].

SIR is intended to be "a repository of software-related artifact meant to support rigorous controlled experimentation." It contains several small to medium sized programs with seeded faults and test suites. One of them is GZIP, of about 6000 lines of code, with about 200 test cases. It contains tens of seeded faults in five versions of GZIP, but many of them do not actually exhibit erroneous behavior on any of the test cases on our platform. In our study, we only used seven seeded faults in one version that have failures.

4.1.2 Branch Prediction

We performed statistical debugging for our subject programs so that we can get pivot branches for each program. First, we instrumented each program with CBI [30], then executed the instrumented program with its test suites and collected all execution profiles. Second, we applied machine learning and clustering algorithms (a la [24]) to identify failure-related branches and rank them depending how likely they are failure-related. In this paper, we use top-5 branches (if any) as pivot branches for each program.

4.1.3 Testing and Analysis Tools

This section presents a brief introduction to CUTE and BLAST, the testing and analysis tools used in our empirical evaluation.

CUTE [43] and its related work DART [16] are a form of dynamic analysis that attempts to systematically execute all feasible program paths for bug detection. It combines symbolic execution with concrete execution to generate new inputs that may direct the next execution along a different program path: The symbolic execution is performed along a concrete execution to collect path constraints; whenever it meets a non-linear constraint (and thus difficult to resolve), it uses concrete values from the concrete execution to simplify the constraint; when an execution finishes, one of the path constraints is negated and the solution for all the constraints (if any) becomes the inputs for next execution.

BLAST [21, 22] is a software model checker for C programs. Given a program and an assertion, BLAST statically proves that either the assertion can not be violated or the program has an execution trace that leads to the violation of the assertion. It performs reachability analysis on abstract program states to check whether a given assertion can be violated or not. If a spurious path (called counterexample) to the assertion violation is found, additional predicates can be discovered to refine the abstract model of the program. The process is repeated, by tracking an increasing number of predicates, until either a genuine failure trace is found, or the abstraction is precise enough to prove the absence of failure traces.

4.1.4 Subject Program Modification

To accommodate CUTE and BLAST, we inserted explicit assertions into every subject program. The assertions are mostly added in a straightforward way according to the seeded faults (of course, CUTE and BLAST use different syntax for the assertions).

For example, if a seeded fault changes a simple assignment from "`index += incr;`" to "`index = incr;`," we add an assertion "`assert( index==0 );`" before the assignment since the faulty program will execute in the same way as the original program only when `index` is 0. As another example, if a seeded fault changes a branching condition in an `if` statement from "`if ( c1 )` {...} `else` {...}" to "`if ( c2 )` {...} `else` {...}," we add an assertion in each of the branches to check the original condition:

```
if ( c2 ) { assert( c1 ); ... }
else { assert( ! c1 ); ... }
```

Also, since CUTE requires explicit annotations for input variables, we identified all input variables manually. We also limited the numbers or sizes of all the inputs if they were too large or statically unbounded; otherwise, CUTE may run for quite a long period of time. For example, one program REPLACE in the Siemens Suite reads from the standard input (`stdin`), we used a character buffer to simulate `stdin` and limited the length of the buffer to be 4; for GZIP, its command line may list arbitrary file names, but we limited the number of file names to be 1 and the name must be selected from one of the file names which are included in the test cases. Also, since the particular version of CUTE we used does not support constraints involving floating point numbers, we used random numbers whenever a program requires a floating point input.

For BLAST, we also have to remove certain assume predicates in simplified programs even when we actually enable branch validation. This is because the particular implementation of BLAST does not handle assume predicates that involve complicated expressions (*e.g.*, pointer arithmetic).

4.2 Evaluation

As mentioned in Section 1, the goal of our approach is to enhance existing in-house testing and analysis techniques so that more faults that are originally difficult to detect can be detected more easily. Therefore, how many faults are detected and how faster the testing and analysis techniques are the measurements we used to evaluate our approach. Due to the page limit, we only show summarized results with sample interesting cases.

4.2.1 More Faults Detected

Our base line for comparison is the number of faults detected by CUTE or BLAST in original programs. Then, CUTE and BLAST are applied to simplified programs with or without branch validation based on all or one failure profiles. The columns "# of Faults Found in" in Table 2 show the number of faults detected by CUTE under the different situations. P represents original pro-

Prgm Name	Total # of Conditionals	# of Reduced Conds in P_s^{bv}	P_s	P_{s1}^{bv}	Total # of Faults	# of Faults Found in P	P_s^{bv}	P_s	P_{s1}^{bv}	# of Iterations Taken by CUTE for P	P_s^{bv}	P_s	P_{s1}^{bv}
PRINT_TOKENS	231	46	46	107	7	1	1	1	0	4418	2426	4414	7
PRINT_TOKENS2	809	41	41	497	10	9	9	9	6	156	151	149	23
REPLACE	2683	881	881	26	32	27	27	31 (18)	26	1892290	707160	152500	100309
SCHEDULE	262	104	104	139	9	8	8	9 (3)	8	137675	126196	215	80507
SCHEDULE2	371	123	123	178	10	6	6	7 (1)	6	786918	331868	235388	159275
TCAS	1348	926	926	1237	41	27	28	41	23	674	205	73	69
TOT_INFO	1011	296	296	606	23	4	4	6	4	88195	25981	283764	72
GZIP	5237	3629	3635	3839	7	7 (4)	5	7 (3)	4	14	915	367	1299
Total	11952	6046	6052	6629	139	89 (4)	88	111 (25)	77	2910340	1194902	676870	341561
Relative Improvements:						0	-1	22	-12	0.0%	58.9%	76.7%	88.3%

Table 2: **Result summary for CUTE-based experiments. P means original programs, P_s^{bv} means simplified programs with branch validation, P_s means simplified programs without branch validation, and P_{s1}^{bv} means simplified programs with branch validation based on one failure profile.**

grams; P_s^{bv} represents simplified programs with branch validation; P_s represents simplified programs without branch validation; and P_{s1}^{bv} represents simplified programs with branch validation based on at most one failure profile. Numbers in parentheses (if any) are the numbers of cases where CUTE reported failures that are not caused by the original faults; all other reported failures are due to the original faults. The false failures are mainly segmentation faults that are introduced by branch reduction without validation; others (mostly for GZIP) are because we killed CUTE whenever one execution of a program (an iteration of CUTE) reached our two-hour time limit.

On average, CUTE can detect more faults (*i.e.*, reach the violation of more assertions) in simplified programs. Here we present interesting cases, in addition to the sample code in Section 1, to show why CUTE can do so.

Example 6. The following example uses an input variable as an array index:

```
int longNumArray[] = { 13, 25, 48, ...,
    /* a very long list of numbers */...,
    255, ... };
if ( longNumArray[ input ]==255 ) {
   ......
   assert( FALSE );
}
```

Although CUTE can represent the path constraint that leads to the assertion correctly (`(*(longNumArray+input)==255)`), it is not able to search for the index of 255 in `longNumArray` to generate the concrete `input` that causes the assertion violation. Here is the simplified code:

```
int longNumArray[] = { 13, 25, 48, ...,
    /* a very long list of numbers */...,
    255, ... };
if ( TRUE ) {
   // assume( longNumArray[ input ]==255 );
   ......
   assert( FALSE );
}
```

Without performing branch validation (*i.e.*, ignoring the assume predicate), CUTE can easily identify the path leading to the failure and tell us the corresponding path in the original code and failure-related constraints. Although in this case, it still does not tell what the actual input causes the failure path, presenting the path with the failure constraints to human developers can save them some time on code searching and help them to focus on the most difficult constraints and construct actual failure-inducing inputs more efficiently. Similar issues exist when code contains calls to library functions where path constraints are not traced by CUTE.

This example also shows one particular reason for the fact that for simplified programs with branch validation (P_s^{bv}) CUTE and BLAST often detect only a few more faults than they do for the original programs: it is because the assume predicates may be no easier than the original branching conditions to track or resolve.

Prgm Name	# of Faults Found in P	$P_s^{[bv]}$	# of Iterations Taken for P	$P_s^{[bv]}$
PRINT_TOKENS	1	1	4418	4414
PRINT_TOKENS2	9	9	156	149
REPLACE	27	28	1892290	731954
SCHEDULE	8	8	137675	126200
SCHEDULE2	6	6	786918	301581
TCAS	27	41	674	73
TOT_INFO	4	6	88195	283764
GZIP	7−4	6	14	865
Total	85	105	2910340	1449000
Relative Improvements:		**20**		**50.2%**

Table 3: **Result summary for combined use of P_s and P_s^{bv} for CUTE-based experiments. $P_s^{[bv]}$ means P_s with conditional uses of P_s^{bv}.**

Also, the version of CUTE we used ignores any constraint involving floating point numbers, but our approach is able to complement such an implementation limitation. For example,

Example 7. For example:

```
if ( (float)input >= 1.0 )    if ( TRUE )
   assert( FALSE );              assert( FALSE );
```

Table 2 also shows that CUTE reported 25 failures for simplified programs without branch validation (P_s) that were not caused by the actual faults in the original programs: 22 of them were segmentation faults caused by branch reduction without validation; 3 others from GZIP exceeded our two-hour time limit during one of its executions. On the other hand, CUTE did not report false positives for P_s^{bv}. It is thus a practical strategy to use P_s^{bv} only to confirm the failure reports from P_s when they appear suspicious.

The actual strategy we used is to apply CUTE on P_s first, and apply CUTE on P_s^{bv} only when an iteration of P_s takes more than two hours or segmentation faults. Such a combined use of P_s with P_s^{bv} (denoted as $P_s^{[bv]}$) helps maintain most of the benefits of P_s and reduce false positives. The columns "# of Faults Found in" in Table 3 show the numbers of failures reported by CUTE based on $P_s^{[bv]}$. In this way, CUTE detected 20 more faults than in the original programs.

It is also interesting to notice that CUTE does not report false failures in simplified programs with branch validation based on only one failure profile (P_{s1}^{bv}) either, but it missed quite a few faults that can be detected in the original programs. This is an expected effect of the adjustable parameter in our approach—the number of available failure profiles (*cf.* Section 3.3). We explain this effect with an interesting example at the end of this subsection, after we discuss the effectiveness of our approach on BLAST.

The columns "# of Faults Found in" in Table 4 show the num-

Prgm Name	Total # of Conds	# of Reduced Conds in			Total # of Faults	# of Faults Found in				# of TP+RA Queries Taken by BLAST for							
		P_s^{bv}	P_s	P_{s1}^{bv}		P	P_s^{bv}	P_s	P_{s1}^{bv}	P		P_s^{bv}		P_s		P_{s1}^{bv}	
PRINT_TOKENS	231	46	46	107	7	6/1	7	7	6/1	0	5695	0	4186	0	4147	0	3273
PRINT_TOKENS2	809	41	41	497	10	10	10	10	10	14424226	1481178	14336580	1476504	14309606	1474237	8292060	647250
REPLACE	2683	881	881	1333	32	2/30	17/15	18/13	18/13	0	662	0	211	0	265	0	265
SCHEDULE	262	64	104	93	9	4/1	4/1	7/1	4/1	136238	11896	0	2601	0	2428	0	2410
SCHEDULE2	371	123	123	178	10	6/1	6/1	6/1	5/2	39638	13799	30326	8047	29100	7447	65486	7172
TCAS	1348	926	926	1237	41	41	41	41	38/2	0	111514	0	13723	0	13734	0	9194
TOT_INFO	1011	296	296	606	23	19/2	20/1	20/1	18/3	0	39429	0	29461	0	28693	0	22881
GZIP	5237	0	3635	0	7	7	0	7	0	0	11164	0	0	0	4452	0	0
Total	11952	2377	6052	4051	139	95/35	105/18	116/16	99/22	14600102	1675337	14366906	1534733	14338706	1535403	8357546	692445
Relative Improvements:							10/17	21/19	4/13			1.6%	8.4%	1.8%	8.4%	42.8%	58.7%

Table 4: **Result summary for BLAST-based experiments. The columns "# of TP+RA Queries for" are the number of queries made by BLAST to theorem provers (TP) and control-flow automaton reachability analysis (RA) respectively.**

ber of faults detected by BLAST (*i.e.*, the number of assertions for which BLAST can discover a path that leads to its violation) under different program settings. Numbers following "/" (if any) are the number of program versions for which BLAST can not decide its correctness. The data shows that our approach helps BLAST to discover 21 more faults in simplified programs and reduces the number of undecided cases by 19. Even with branch validation enabled (*i.e.*, more complicated predicates for BLAST to handle), BLAST detected 10 more faults with 17 fewer undecided cases.[6] In addition, our approach also helped to validate the correctness of one program which is the source of **Example 2** in Section 1.1.

How could BLAST, a sound model checker, miss faults in original programs? That is because BLAST makes unsound assumptions for real programs. For example, if a function in a program has no explicit definition, BLAST would assume the function is free of side-effects except for the assignments through return values.

Example 8. `scanf` in the following code reads an integer from the standard input and changes the value of `input`. The assertion will be violated if `input` is not zero. However, BLAST would unfortunately report that the code on the left side is always safe.

```
input = 0;                      input = 0;
scanf("%d", &input);            scanf("%d", &input);
if ( input!=0 ) {               if ( TRUE ) {
   ......                          //assume( input!=0 );
   ......                          ......
   assert( FALSE );                assert( FALSE );
}                               }
```

Our approach may transform the code from the left to the right, then BLAST can easily discover a failure trace in the simplified code, and the `assume` can help human developers to actually understand the failure scenario.

Similar to CUTE, BLAST also missed some faults when it is applied to simplified programs with branch validation based on only one failure profile (P_{s1}^{bv}). This is the effect of different numbers of failure profiles. Although all of P_s^{bv}, P_s, and P_{s1}^{bv} are constructed to be failure-preserving, they only preserve failure paths that have occurred in the failure profiles used to construct them. *I.e.*, given a different set of failure profiles, P_s^{bv}, P_s, and P_{s1}^{bv} may preserve different paths that lead to failures. Constraints along certain failure paths may be much more difficult than others to resolve, and thus CUTE or BLAST may not be able to identify the failure path if P_{s1}^{bv} preserves only the paths that are difficult to resolve.

Example 9. The following example illustrates such a situation:

```
1  error = 0; y = 0;              error = 0; y = 0;
2  if ( cond ) {                  if ( FALSE ) {
3     if ( x==255 )                  exit(0);
4        error = 1;
5     else ...
6  } else {                       } else {
7     while ( i<count ) {            while ( i<count ) {
8        library_func(&y);              library_func(&y);
9        if ( y==255 )                  if ( y==255 )
10          error = 1;                     error = 1;
11       else ...                       else ...
12       i++;                           i++;
13    }                              }
14 }                              }
15 assert( error==0 );            assert( error==0 );
```

The code on the left has two possible failure paths: one is through Line 4, and the other is through Line 10. Suppose we only have one failure profile that went through Line 10, then the simplified code may look like the right side: the condition at Line 2 is reduced to `FALSE`, the `if` at Line 3 is removed since it does not occur in the failure profile, and the `while` loop is remained since all of the branches in the loop may have appeared at least once in the failure profile. Thus, due to the same reason as **Example 8**, BLAST and CUTE may fail to report the failure even in the simplified code.

4.2.2 Make CUTE and BLAST Faster

Another benefit of our approach is that programs are simplified and CUTE and BLAST can run more efficiently. The last four columns in both Table 2 and 4 show the performance gains.

We measure the performance of CUTE in terms of the number of iterations it takes to reach the violation of assertions. Each iteration is a concrete execution of a program, and we assume each iteration takes about the same of time. Overall, our approach reduced the iteration numbers taken by CUTE by more than 50%, for both P_s and $P_s^{[bv]}$ (*cf.* the last rows in Table 2 and 3). In general, the more branches are predicted to be `False`, the more performance gains we may get, although the actual speedup may depend on the complexity of the original programs.

For BLAST, we use the numbers of queries to theorem provers and reachability analysis made by BLAST to measure its performance. Similar to CUTE, the more branches are predicted to be `False`, the more reduction in the number of queries. For example, we were only able to reduce about 5% of the conditionals in PRINT_TOKENS2 to construct P_s, and the corresponding reduction in the number of queries is less than 1%. For programs that we can reduce more than 30% of the conditionals, such as REPLACE and SCHEDULE, BLAST detected more faults in the simplified versions than the original versions, and we saw more than 50% reduction in the number of queries.

[6]We did not run BLAST on GZIP with branch validation enabled since thousands of assume predicates are syntactically too complicated for the version of BLAST to handle, and we did not implement an automated predicate simplification for BLAST.

4.3 Discussions

Interpretation of Failure Reports. When CUTE or BLAST reports a failure path, we can treat the path as a path in the original program although it is just a path in the simplified program due to the reduced conditions. We can do so because we know the reduced conditions must have been satisfied somewhere in the field during users' executions and we have high confidence that the reported path is also feasible in the original program. The reported path can become a real failure path in the original program when the reduced conditions along the path are satisfied. If the reduced conditions can be resolved by a constraint solver, the real failure path can be constructed automatically; If the conditions are difficult to reason or resolve, manual efforts may still be required to help construct concrete scenarios that correspond to the failure path.

Effects of Branch Validation. As shown before, branch validation with simplified programs may help suppress false failure reports. Also, it may help speed up subsequent testing and analysis.

Example 10. Suppose the following code on the left may violate the assertion only when `input` is within the range of $[99900, 99998]$:

```
if ( input > 99899 &&            if ( TRUE ) {
     input < 99999 ) {              //assume( input>99899
   ......                           //    && input < 99999 );
   ......                           ......
   for (i=0; i<input; i++)          for (i=0; i<input; i++)
     sum += foo(i);                   sum += foo(i);
   assert( sum!=0 );                assert( sum!=0 );
}                                }
```

Then, CUTE may only need 3 iterations to generate an input that leads to the assertion violation. However, if there is no `assume` in the simplified code on the right side, CUTE may take tens of thousands of iterations, depending on its actual path search strategies, to explore many different (out-of-range) values for `input`.

On the other hand, as the numbers in Table 2 and 4 suggest, simplified programs with branch validation (P_s^{bv}) may often be slower than simplified programs without branch validation (P_s), and more importantly, miss certain bugs that may be detected in P_s since the validation may involve complicated predicates.

According to our empirical study (*cf.* Table 3), it is practical to use $P_s^{[bv]}$ to reduce false positives. To further take advantage of the possible speedup of assume predicates, we may choose to enable assume predicates that are "simple enough" only. It will be interesting to investigate what the *optimal assume predicates* that help to detect the most faults and offer the greatest speedup are.

Effects of User Profiles. Profiles from users are essential for the effectiveness of our approach. As mentioned in Section 3.3 and 4.2.1, fewer failure profiles may lead to fewer detectable faults, and thus it is desirable to have as many failure profiles as possible.

Also, each single profile may be sampled before being sent back to developers in order to further reduce run-time overhead for users and protect their privacy. Such profiles may contain even less information about each execution. Our approach is still applicable for such profiles, although it will require larger number of profiles to predict branching directions and select pivot branches accurately. Previous work on statistical debugging [32, 33] suggested that tens to hundreds of sampled failure profiles are often enough for identifying good failure indicators. Consequently, as long as the number of profiles is large enough, branches that need to be taken in failed executions will eventually occur in the same failure profiles with one of the failure indicators, and the correlative predictions will still be able to select right branches to simplify.

How long it takes to collect so many failure profiles for a program depends on how often users use the program, the frequency with which a failure occurs in an execution, and the sampling rate. One thing for certain is that, our approach pragmatically focuses one's attention more on those failures experienced more by users, and thus helps prioritize debugging efforts accordingly.

Existence of Multiple Bugs. Our approach is still applicable when there are more than one bugs in a program, as long as statistical debugging is still able to choose accurate pivot branches for us, and such techniques do exist [26, 33, 49].

On the other hand, our approach could improve the accuracy and performance of in-house techniques further if we could reduce a program further. It would be ideal that a set of profiles used to simplify a program only contains failure profiles that are generated by the same bug so that we can reduce a program specifically for one bug. Thus, it will be interesting to investigate how failure triage techniques [7, 8, 14, 20, 34, 36, 41] that separate profiles caused by different bugs from each other can actually help improve the effectiveness of our approach.

Accuracy of Statistically Selected Pivot Branches. The more pivot branches, the simpler a program can be reduced and the better for in-house testing and analysis techniques. However, the pivot branches we used may not be actual pivot branches; they were from statistical debugging: Due to various reasons, statistical debugging may rank the most failure-related predicate lower than others; If the failure-related predicate is not included as a pivot branch, the corresponding failure profile subsets may be different; This may have negative effects on bug detection. Thus, in order to be failure-preserving, it is usually better to include more predicates as pivot branches when applying our approach. When we know the accuracy of the statistically selected pivot branches is low, we can use fewer pivot branches and repeatedly apply our approach with different sets of pivot branches, so that we can utilize the improved performance provided by further simplified programs to test and analyze different aspects of the original programs for defects.

Aggressive Predictions. We could make the process for predicting branching directions summarized in Section 2.2 more aggressive. For example, in the first step, we could let the truth value of a branch predicate p to be `False` as long as p is not satisfied in *most* (>50%) failure profiles; then more branched may be reduced to facilitate subsequently applied testing and analysis techniques. However, if a more aggressive prediction is not failure-preserving, it may cause other (unexpected) side-effects besides missing bugs. For example, **Example 4** in Section 3.2 showed that the particular branch reduction caused an infinite loop in the code; testing techniques, such as CUTE, will be bound to fail. Whether there exist balance points that make the best of both conservative and aggressive predictions is also an interesting topic for future investigation.

Also, as mentioned in Section 3.3, correlation is sufficient but not necessary for a prediction to be failure-preserving. We could certainly relax the conditions in the definition of correlation by exploring other hidden information in execution profiles. It will be interesting future work to consider a more accurate but still easily verifiable alternative for correlation so that programs may be simplified further. Such considerations may provide insights about the necessary information for deriving accurate predictions and in return guide profile-collecting infrastructures to select better program predicates for instrumentation.

Still Missing Bugs. Note that the bug finding capability of our approach will be up to the extend of program simplification and the capability of existing testing and analysis tools. Our approach is not a direct testing or debugging approach in the sense that itself

does not locate or fix program faults; it replies on other tools to look for bugs. However, the potential of our approach is to significantly narrow down possible bug locations based on large amount of execution profiles from users and help other techniques to focus on the failures occurred most often and to find more faults.

Many seeded faults in our study were not detected by CUTE or BLAST, even in the simplified programs. Some of them are due to implementation limitations in CUTE and BLAST (such as limited support for floating point numbers or function pointers); others are due to path constraints that are inherently difficult to solve (such as multi-variable non-linear operations and side effects caused by external function calls). These situations may all be circumvented by our technique if enough profiles could be collected to help simplify the program significantly so that unsupported constraints can be reduced.

There are also several program faults that only cause failures when their input sizes are significantly large. Even though our technique helped to simplify the programs, CUTE and BLAST were still unable to detect the faults because they exhausted the computing resources we could tolerate. We view all such cases as opportunities for future profile-guided program simplification. It would be interesting research directions to explore what kinds of program simplifications would be more effective for such cases, and what kinds of information profiles could include to further ease the tasks of program simplification and bug localization.

5. RELATED WORK

In this section, we discuss additional related work, including partial evaluation, program slicing, and their relationship with our approach. At a high-level, all these studies share the similar idea with this paper: a program may be simplified in certain ways so that it may consequently be better for certain applications.

Partial Evaluation. This area has an extensive literature. It concerns how to specialize a program by fixing one or more of its inputs to some particular values. The specialization is often accomplished by detecting code fragments depending exclusively on specialized variables and by symbolically pre-computing these fragments. The specialized program will run faster because of reduced computation during execution. It has many applications, such as optimization and generation of program generators [12, 27].

Technically, specialized programs generated by partial evaluation take fewer inputs than original programs. However, simplified programs generated by our approach usually take the same number of inputs. More importantly, partial evaluation is usually semantically-equivalent (w.r.t. specialized inputs) program transformations. This is fundamentally different from our approach: our program reduction is not required to be semantic-preserving, but simply error-preserving.

Program Slicing. This is a technique for simplifying programs w.r.t. certain semantics of interest for code analyzers. The process of slicing deletes many parts of a program that have no effect on the selected semantics. For example, given a variable at a program point, all program statements that do not affect or are not affected by the variable may be reduced. It also has many applications, such as testing and debugging, program differencing, and program comprehension and maintenance [6]. For example, Bates and Horwitz use control slices to select test cases according to certain test data adequacy criteria [4]. Also, Groce *et al.* slice a program to a much smaller one that can produce events consistent with a given (partial) trace of (failure-indicating) events so that further failure analysis may be carried out on the sliced program [18]. Dynamic slicing is also introduced for debugging [28]; Agrawal *et al.* rely on the differences between slices of failed and successful runs to identify code segments that may contain bugs [1]. Gupta *et al.* use the intersection of forward and backward dynamic slices to reduce the sizes of code for further debugging [19].

These techniques often require existing test cases or instrumentation infrastructures that can collect slices from users. If they need existing test cases, they are more appropriate for in-house testing and debugging, while our approach aims for failures that mainly happen in the field. If they collect slices, it may impose additional burden on users since the slices often contain more detailed information about an actual execution and compromise users' privacy. Also, slicing often preserves the semantic of interest in a program, while our approach only preserves error paths.

Bug Localization. When a failure occurs, it is useful to automatically locate failure indicators based on profiles to predict possible bug locations so that the subsequent debugging process may be more focused and easier. Many studies have used different statistical methods to identify different kinds of failure indicators, such as program states, execution counts of statements, branches, functions, and event transitions [9,25,42]. The main difference between our approach and these bug localization techniques is that our approach helps connect in-house techniques with the large number of execution profiles from users so that in-house development may benefit more easily from users in the field, while other bug localization techniques have not yet provided such a connection.

Besides these static techniques that work on source code, there are also dynamic approaches that can locate program defects through actual executions. Delta debugging may be classified in this category since it locates both failure-related program states and cause transitions [11, 47] based on a given failed execution; it thus requires concrete failure-inducing inputs and is more appropriate for in-house debugging. Similar to CUTE, EXE [10] combines symbolic execution and concrete execution and forks executions at conditionals, aiming to cover as many program paths as possible and look for bugs on different paths. With a similar goal to EXE, PathExpander [37] addresses the path coverage problem in dynamic bug detection by executing both taken and non-taken branches transparently; it also utilizes hardware extensions to achieve low overhead for the parallelizable executions. Except for their dynamic nature, These techniques are different from ours in that we aim to scale down programs while they aim to increase path coverage.

Connecting In-house and the Field. There are also other studies that aim to bring the benefits of profiles into in-house development from different aspects.

Liblit *et al.* propose a family of analyses to build time lines of possible program actions that lead to failures based on information collected from users, such as failure points, stack traces and event logs [31]. Manevich *et al.* use postmortem symbolic evaluation to produce a set of execution traces along which the program may be driven to one *given* failure point [38]. Lal *et al.* construct a shortest control flow path that contains the maximum number of failure-related predicates selected by statistical debugging [29]. These techniques often need to analyze all code in original programs and resolve complicated constraints, while our approach can complement them well since we aim to simplify programs and may help those techniques to relatively scale up.

6. CONCLUSIONS

In this paper, we have presented a novel technique to utilize lightweight execution profiles from users to make existing in-house testing and analysis techniques more effective. We use information from statistical debugging to simplify a program, preserving the original program's erroneous behavior w.r.t. user execution profiles. This way, more bugs in the original program may be discovered in

less time. We performed case studies with two testing and analysis techniques, CUTE and BLAST, on several subject programs. Our empirical results have shown that using the simplified programs, both CUTE and BLAST have been able to find 20 and 21 more bugs respectively in about half of the original time. Our approach significantly reduces the gap between the needs of developers for detailed failure scenarios and the needs of users for lightweight execution profiling. We believe profile-guided program simplification is a promising direction towards realizing automated debugging.

Acknowledgments

We are grateful to all the authors of CBI, Aristotle Analysis System, SIR, CUTE, and BLAST for providing the infrastructure, data, and tools used in our study. We also greatly appreciate all the feedback from Jim Apple, Earl Barr, Mark Gabel, Gary Wassermann, and anonymous reviewers for the drafts of this paper.

7. REFERENCES

[1] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE*, pages 143–151, 1995.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Priciples, Techniques, and Tools*. Addison-Wesley, 1986.

[3] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *ISSTA*, pages 5–15, 2007.

[4] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *POPL*, pages 384–396, 1993.

[5] D. Beyer, A. J. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *ICSE*, pages 326–335, 2004.

[6] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[7] J. F. Bowring, M. J. Harrold, and J. M. Rehg. Improving the classification of software behaviors using ensembles of control-flow and data-flow classifiers. Technical Report GIT-CERCS-05-10, Georgia Institute of Technology, April 2005.

[8] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA*, pages 195–205, 2004.

[9] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, pages 480–490, 2004.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *ACM Computer and Communications Security (CCS)*, pages 322–335, 2006.

[11] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[12] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *POPL*, pages 493–501, 1993.

[13] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *ICSE*, pages 422–431, 2005.

[14] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE*, pages 339–348, 2001.

[15] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

[17] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM TOSEM*, 10(2):184–208, 2001.

[18] A. Groce and R. Joshi. Exploiting traces in program analysis. In *TACAS*, volume 3920 of *LNCS*, pages 379–393. Springer, 2006.

[19] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE*, pages 263–272, 2005.

[20] M. Haran, A. F. Karr, A. Orso, A. A. Porter, and A. P. Sanil. Applying classification techniques to remotely-collected program execution data. In *ESEC/SIGSOFT FSE*, pages 146–155, 2005.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN*, volume 2648 of *LNCS*, pages 235–239, 2003.

[23] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, pages 191–200, 1994.

[24] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE*, pages 184–193, 2007.

[25] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282, 2005.

[26] J. A. Jones, M. J. Harrold, and J. F. Bowring. Debugging in parallel. In *ISSTA*, pages 16–26, 2007.

[27] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3), September 1996.

[28] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct 26 1988.

[29] A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path optimization in programs and its application to debugging. In *ESOP*, 2006.

[30] B. Liblit. *The Cooperative Bug Isolation Project*. `http://www.cs.wisc.edu/cbi/`.

[31] B. Liblit and A. Aiken. Building a better backtrace: Techniques for postmortem program analysis. Technical Report CSD-02-1203, UC Berkeley, 2002.

[32] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[33] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[34] C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *FSE*, 2006.

[35] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *FSE*, pages 286–295, 2005.

[36] C. Liu, X. Zhang, J. Han, Y. Zhang, and B. K. Bhargava. Failure indexing: A dynamic slicing based approach. In *ICSM*, pages 382–393, 2007.

[37] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *MICRO*, pages 38–52, 2006.

[38] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. In *FSE*, 2004.

[39] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, volume 2304 of *LNCS*, pages 213–228, 2002.

[40] C. S. Pasareanu, M. B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted java programs. In *TACAS*, volume 2031 of *LNCS*, pages 284–298, 2001.

[41] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE*, pages 465–477, 2003.

[42] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[43] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.

[44] Siemens, M. J. Harrold, and G. Rothermel. *Aristotle Analysis System – Siemens Programs, HR Variants*. http://www.cc.gatech.edu/aristotle/Tools/subjects/.

[45] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA*, pages 97–107, 2004.

[46] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, 2005.

[47] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, 2002.

[48] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems 16*. 2003.

[49] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *International Conference on Machine Learning (ICML)*, pages 26–29, 2006.