THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRC7/14

Iterative Statistical Bug Isolation via Hierarchical Instrumentation

Zhiqiang Zuo and Siau-Cheng Khoo

July 2014

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

David ROSENBLUM
Dean of School

Iterative Statistical Bug Isolation via Hierarchical Instrumentation

[Technical Report]

Zhiqiang Zuo
School of Computing
National University of Singapore
zhiqiangzuo@nus.edu.sg

Siau-Cheng Khoo
School of Computing
National University of Singapore
khoosc@nus.edu.sg

ABSTRACT

Cooperative statistical bug isolation approaches monitor program runtime behavior from end-user executions and apply statistical techniques to pinpoint likely causes of field failures. Having end-users to run fully instrumented programs is extremely undesirable, as it adds tremendous performance overhead to end-users; it is also unnecessary, since in practice major part of these programs are error-free and need no instrumentation. In order to reduce the monitoring and computational costs as well as the performance overhead of end-users, we propose a novel iterative (and cooperative) statistical bug isolation approach for field failures via *hierarchical instrumentation* (HI). Different from the existing iterative approaches, our approach via HI does not only maintain small end-user overhead, but also performs iterative bug isolation *automatically without developers' intervention*, thereby relieving developers of repetitive efforts in isolating bugs. In this paper, we formalize the concept and principles behind the HI technique, and demonstrate its practicality through experimentations. Our experiments show that this new approach via HI saves significant instrumentation effort and sharply reduces runtime overhead, while upholding the accuracy of bug isolation.

Keywords

automated debugging, hierarchical instrumentation, iterative statistical bug isolation

1. INTRODUCTION

Most software deployed around the world remains buggy in spite of extensive in-house testing. Hitherto, debugging continues to be a tedious and painstaking effort for developers. As an essential and yet expensive process in debugging, bug isolation (or fault localization) aims to isolate or locate program faults [1]. It has spun off many research activities aiming to automate this process. One automated bug isolation approach for field failures which has received much attention recently is *cooperative (statistical) bug isolation* [2, 3]. This approach applies the idea of crowd-sourcing in sampling classes of program runtime behavior from a large pool of end-users running instrumented programs for bug tracking. The gathered program traces enable developers to apply statistical techniques to pinpoint the likely causes of failures. The success of this approach hinges on the availability of a sufficiently large user base to run the instrumented programs. In order to encourage a great number of users to participate, the user's overhead for running the instrumented programs should be kept sufficiently low. To this end, Liblit et al. adopted a sparse random sampling technique [2]. This technique amortizes the cost of monitoring user-end executions to a large number of users so that each user suffers a relatively low time overhead. Nevertheless, from the perspective of developers, this approach does not really reduce the monitoring cost or the total size of execution data, which consumes many resources such as network bandwidth, storage space, CPU time, etc, due to the need for data transfer, storage and analysis. This constrains the practicability of post-deployment bug hunting, especially for large applications.

A major problem with these bug isolation systems is that they consider *every program statement* to be potentially relevant to the failure. The instrumentation sites thus spread across the entire program, even though, in actual fact, *only small portions of a program are relevant to a given bug* [4]. As mentioned in [3], the majority of the predicates tracked (often 98-99%) are irrelevant to program failures, and thus unnecessary for collection, storage and analysis. To ensure minimal effort spent by end-users and developers, *iterative bug isolation* approaches have been proposed: Chilimbi et al. employed an iterative, locality-based instrumentation to isolate the causes of program failures [4]. Specifically, at each iteration, they monitored a set of functions, branches and paths to analyze whether these are strong predictors of the failure. If so, they terminated by returning these strong predictors. Otherwise, they expanded the search via a static analysis to monitor other parts of code closely interacting with the weak predictors. Similarly, Arumuga et al. [5] proposed an adaptive monitoring strategy based on the following principle of locality: *If a predicate is highly predictive of failure, then predicates in its vicinity are potentially good bug predictors as well.* To this end, the strategy monitors a few predicates at each iteration and adaptively adjusts the instrumentation plan to include sites close to the highly suspicious predicate currently explored.

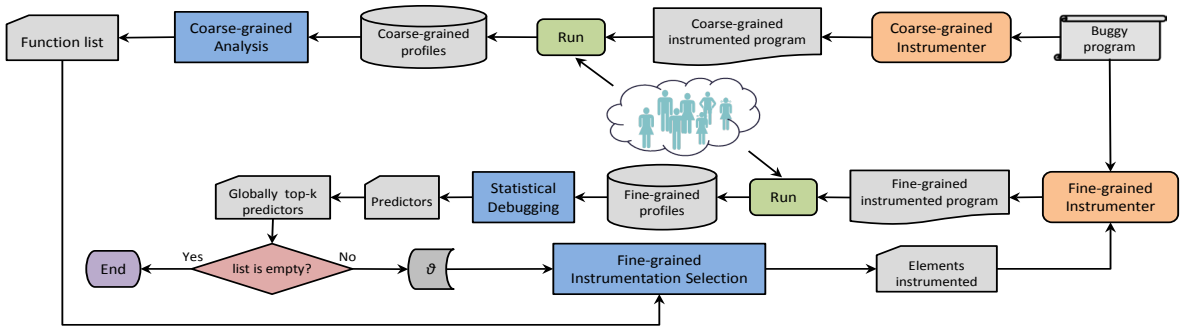

Figure 1: Workflow of iterative statistical bug isolation for field failures via hierarchical instrumentation

However, there are two main drawbacks. Firstly, both iterative approaches [4, 5] make use of the principle of locality to guide their search for bugs; this principle however is not always effective in localizing bugs, as experiments have found [5]. Secondly and also more importantly, both iterative approaches require developers to check the predictors reported at each iteration, until the bug cause is found. As claimed in [6], developers are reluctant to go through a list of predictors, not to mention the need to repetitively perform this check at every iteration.

In order to tackle the above drawbacks, we devise a novel *Hierarchical Instrumentation* (HI) technique. We employ HI to perform selective instrumentation and propose an iterative bug isolation approach for field failures via HI. We perform instrumentation at two different levels in HI, namely coarse-grained (function-level) and fine-grained, the latter of which is done in typical bug isolation approaches (e.g., predicate-level). Our approach via HI consists of two phases: a one-pass coarse-grained phase followed by an iterative fine-grained phase guided by results obtained from the first phase. The process runs as follows: first we instrument the entire program at a coarse granularity (e.g., function entry-level) and deploy the instrumented program to users for execution; such instrumentation is guaranteed to be lightweight. Once a sufficient number of user executions are collected, we calculate the coarse-grained execution information for each function. This information is then exploited to guide the iterative fine-grained (predicate-level) instrumentation. In the iterative phase, we use the coarse-grained information and feedback from previous iterations to help select a function at each iteration and instrument all the predicates in that function. This fine-grained instrumented program is then deployed. As we only instrument one function, the runtime overhead incurred at user side is kept minimal. We again collect the execution data and measure the suspiciousness value for each predicate in the function. We iterate the process until all the selected functions are exhaustively examined. Eventually, the globally top-k predictors are reported to the developers. Figure 1 illustrates the workflow of our system. We adopt the instrumentation scheme used in CBI [3] for our fine-grained instrumentation and apply *Importance* measure to assess the suspiciousness of each predicate.

Comparison. Compared with HOLMES [4] and Adaptive Bug Isolation [5], our technique is much simpler: we do not need to perform any static analysis or construct dependence graphs. Secondly, our technique is more straightforward and more effective. It does not rely on the validity and/or effectiveness of the principle of locality. More importantly, we perform iterative bug isolation in a *fully automated* way without developers' intervention while safely pruning away unnecessary instrumentation. Developers are only required to check for the globally best predictors computed at the end of all iterations. Thus they are relieved of repeated checking throughout the entire bug isolation process.

Our current work is an enhancement to our earlier work that applied the HI technique to in-house debugging (bug signature mining) [7]. In this paper, we formalize the principles of HI, and demonstrate its effectiveness in debugging field failures. We will further discuss the differences in Section 7.

Contributions. We list the following contributions.

- We formalize the underlying principles of the hierarchical instrumentation technique and provide a general and systematic approach to applying it. Various statistical debugging approaches can thus benefit from this technique.
- We propose an iterative statistical bug isolation approach for field failures via hierarchical instrumentation. We iteratively perform fine-grained instrumentation as guided by the coarse-grained information, and dynamically prune unnecessary instrumentation.
- We conduct the experiments on real-world programs. The experimental results validate that our approach not only saves the total monitoring and computational costs, but also sharply diminishes the performance overhead of end-users.

The remainder of this paper is organized as follows. Section 2 gives some background about statistical bug isolation. We formalize the HI technique in Section 3, followed by our iterative statistical bug isolation approach via HI in Section 4. Section 5 presents the evaluation of our work. We provide a discussion and an overview of related literature in Section 6 and 7, respectively. Finally, Section 8 concludes.

2. BACKGROUND

Before presenting our work, we give a brief overview of cooperative statistical bug isolation approach and its extension to adaptive bug isolation. We also briefly associate our work with these approaches.

2.1 Cooperative Statistical Bug Isolation

Cooperative bug isolation (CBI) is a dynamic analysis for locating the causes of program field failures. It collects execution information of an instrumented program from both failing and passing end-user runs, and employs statistical techniques to pinpoint the likely root causes of software failures [3].

Specifically, a program is instrumented to collect the runtime values of *predicates* at particular program points. Each program point to be instrumented is called *instrumentation site*. At each instrumentation site, several *instrumentation predicates* are tracked. There are three categories of instrumentation sites considered:

- **Branches**: For each conditional, two predicates are recorded to indicate whether the *true* or *false* branch is taken at runtime.
- **Returns**: At each scalar-returning call site, six predicates are tracked to capture whether the return value r is ever $> 0, \geq 0, < 0, \leq 0, = 0$, or $\neq 0$.
- **Scalar-pairs**: At each assignment of a scalar value, six relationships between the assigned value x and each other same-typed in-scope variable y_i are considered. Specifically, for each y_i, six predicates are created: $x < , \leq, >, \geq, =, \neq y_i$.

A profile is obtained for each run of the instrumented program. It consists of a set of predicate counts which records the number of times each predicate is evaluated to be true during this execution. In CBI [3], these counts are simplified to binary values (i.e., they only distinguish whether the predicate is true at least once or never). In addition, each profile is labeled as passing or failing. Having these profiles, the statistical technique is applied to compute a suspiciousness value for each predicate. The top scored predicate is regarded as the best predictor of the failure.

The following measure called *Importance* (Equation 1) has been used to assess the suspiciousness of predicates [3]. *Importance* is actually the harmonic mean of *Increase* (Equation 2) and *Sensitivity* (Equation 3). Notation-wise, for a predicate e, $p_t(e)$ and $n_t(e)$ are the number of passing and failing runs where e is observed to be true, respectively. $p(e)$ and $n(e)$ are the number of respective passing and failing runs in which e is observed, regardless of whether e is evaluated to be true or false. Note that *Importance* is actually a function under the natural number domain. Specifically, $p \in [0, P], n \in [0, N], p_t \in [0, p], n_t \in [0, n]$ where P and N are the total number of passing and failing runs, respectively.

$$Importance(e) = \frac{2}{\frac{1}{Increase(e)} + \frac{1}{Sensitivity(e)}} \quad (1)$$

$$Increase(e) = \frac{n_t(e)}{n_t(e) + p_t(e)} - \frac{n(e)}{n(e) + p(e)} \quad (2)$$

$$Sensitivity(e) = \frac{\log n_t(e)}{\log N} \quad (3)$$

In order to reduce the performance impact of instrumentation, CBI adopts sparse random sampling [2] derived from the work of Arnold and Ryder [8]. This technique generates instrumentation which samples a sparse and random subset of predicate counts during the execution. This helps to protect privacy and diminish performance overhead. However, the total monitoring and computational cost is not reduced as a whole. Moreover, sampling [8, 2] doubles the size of the executable, which constrains the practicality especially for large applications.

In this work, *we adopt CBI as our fine-grained statistical model*: we employ the instrumentation scheme discussed above to perform our fine-grained instrumentation, and *Importance* measure to assess the suspiciousness of predicates. However, instead of sampling, we conduct iterative monitoring. We select one function to be monitored at each iteration and instrument all the predicates within this function.

2.2 Adaptive Bug Isolation

Based on the observation that *only small portions of a program are relevant to a given bug* [4], two adaptive statistical bug isolation approaches [4, 5] have been proposed. HOLMES employs an iterative, locality-based instrumentation scheme to address both time and space overheads [4]. At each iteration, it identifies predictors of the reported failure. Based on the suspiciousness values of these predictors, HOLMES either decides that these predictors are the root causes, or expands the search by monitoring code in other functions that closely interact with these weak predictors.

Adaptive bug isolation presented by Arumuga et al. [5] is a fine-grained adaptive monitoring system which adaptively monitors the program at the granularity of predicates. It monitors a fraction of predicates at each iteration and adaptively adjusts the instrumentation plan to include sites closer to the highly suspicious predicate currently explored. It is formulated as a search on the control dependence graph and presents several heuristics to guide the search.

Similarly, we also perform statistical bug isolation in an iterative way. Different from them, we propose a novel hierarchical instrumentation to facilitate iterative monitoring.

3. HIERARCHICAL INSTRUMENTATION

In order to make statistical debugging approaches more efficient while not affecting their effectiveness, we devise a novel hierarchical instrumentation technique to safely and effectively perform selective instrumentation. The core of HI is to prune away unnecessary instrumentation and only instrument a set of prospective program elements such that bug-relevant elements can be discovered with much less instrumentation effort. HI is based on the following insight [7].

INSIGHT 1. *Information collected and measured by instrumenting composite syntactic constructs (e.g., functions) can be used to guide the selection of program elements (e.g., predicates) for subsequent instrumentation.*

We call the former instrumentation *coarse-grained* whereas the latter *fine-grained*. In brief, we first perform a lightweight coarse-grained instrumentation and obtain suspiciousness information of coarse-grained elements (e.g., functions). By means of such coarse-grained suspiciousness information, we safely and effectively prune away instrumentation of fine-grained elements (e.g., predicates). The core of the HI technique is a safe and effective pruning of instrumentation.

Specifically, given a statistical debugging approach where a fine-grained suspiciousness measure F is employed to quantify

the suspiciousness of fine-grained elements (e.g., predicates). This approach will analyze two distinct groups of executions and return the top-k suspicious elements. The goal of the HI technique is to make the statistical debugging approach more efficient by performing selective instrumentation, while upholding the original effectiveness of debugging (i.e., producing the same top-k elements with identical F values). In order to safely and effectively prune away unnecessary instrumentation of fine-grained elements, our HI technique requires two coarse-grained measures: one C_p for pruning and another C_r for ranking.

3.1 Coarse-grained Measure for Pruning

A coarse-grained pruning measure C_p assigns a real to each coarse-grained element (such as function). This real number will be used to determine if the predicates within the corresponding function need to be instrumented during fine-grained instrumentation. We formalize this measure as follows:

DEFINITION 3.1 (**Coarse-grained Pruning Measure**). *Given a fine-grained measure $F : \mathbb{N}^2 \to \mathbb{R}$ defined over the intervals $([0, X], [0, Y])$, a function $C_p : \mathbb{N}^2 \to \mathbb{R}$ under the same domain is defined as a coarse-grained pruning measure if it satisfies the following conditions:*

- *C_p is an upper bound of F, i.e., $C_p(x, y) \geq F(x, y)$*
- *C_p is not decreasing, i.e., $C_p(x, y) \geq C_p(x - 1, y) \wedge C_p(x, y) \geq C_p(x, y - 1)$*
- *C_p is as close to F as possible, i.e., $\sum_{x,y=0}^{X,Y}\{C_p(x, y) - F(x, y)\}$ is kept minimal*

According to the above definition, given a fine-grained suspiciousness measure F in a close integer sub-interval domain ($x \in [0, X], y \in [0, Y]$), the best coarse-grained measure $C_p(x, y)$ can be computed by dynamic programming. Algorithm 1 gives the details. As is widely known, its complexity is linear to the size of its domain space, i.e., $O(XY)$.

Algorithm 1: Coarse-grained Pruning Measure

Input: fine-grained suspiciousness measure $F[X{+}1,Y{+}1]$
Output: coarse-grained pruning measure $C_p[X{+}1,Y{+}1]$

```
// base case
Cp[0,0] ← F[0,0]
for x ← 1 to X do
  | Cp[x,0] ← max{F[x,0], Cp[x−1,0]}
end
for y ← 1 to Y do
  | Cp[0,y] ← max{F[0,y], Cp[0,y−1]}
end
// body
for x ← 1 to X do
  | for y ← 1 to Y do
  |   | Cp[x,y] ← max{F[x,y], Cp[x−1,y], Cp[x,y−1]}
  | end
end
```

3.2 Necessary Condition

Having the coarse-grained pruning measure, we can derive a necessary condition for a coarse-grained element to contain fine-grained elements of high suspiciousness.

THEOREM 3.1 (**Necessary Condition**). *Let e denote a fine-grained element, m be the corresponding syntactic construct (i.e., coarse-grained element) encompassing e. $p(e)$ (or $p(m)$) and $n(e)$ (or $n(m)$) denote the number of passing and failing runs where e (or m) is executed, respectively. Given a threshold θ, fine-grained suspiciousness measure F and coarse-grained pruning measure C_p, we have the following implication.*

$$F(p(e), n(e)) \geq \theta \implies C_p(p(m), n(m)) \geq \theta$$

PROOF.

$$F(p(e), n(e)) \geq \theta \tag{4}$$
$$\implies C_p(p(e), n(e)) \geq \theta \tag{5}$$
$$\implies C_p(p(m), n(m)) \geq \theta \tag{6}$$

According to Definition 3.1, we know that C_p is an upper bound of F. Therefore, (5) holds. Since m corresponds to a syntactic construct of e, whenever e is executed, m must be executed, i.e., $p(e) \leq p(m)$ and $n(e) \leq n(m)$. In addition, C_p is not decreasing, we can thus derive (6). □

Theorem 3.1 identifies the necessary condition for instrumenting fine-grained element in coarse-grained element m; that $C_p(p(m), n(m)) \geq \theta$. It asserts that, when the coarse-grained pruning value for m is less than certain threshold θ, we do not need to consider the fine-grained elements in m for instrumentation, as none of them has suspiciousness value greater than or equal to θ. We formalize this condition check as follows:

COROLLARY 3.1. *Given a threshold θ, a coarse-grained element m, let $p(m)$ and $n(m)$ denote the number of passing and failing runs where m is executed, respectively. If $C_p(p(m), n(m)) \geq \theta$ is false, then the fine-grained elements in m need not be instrumented.*

3.3 Coarse-grained Measure for Ranking

Note that a threshold θ is required when checking the above necessary condition. We have to consider the following issues when determining θ:

- **Safeness.** The value of θ ensures that the top-k elements that have been identified from the fully instrumented program are still derivable in our approach.
- **Effectiveness.** The value of θ enables many bug-irrelevant elements to be excluded during fine-grained instrumentation.

Specifically, on one hand, to guarantee the safeness of pruning, the threshold θ must be kept low – it must be less than or equal to the F value of top-kth element returned by the original statistical debugging approach. On the other hand, this threshold should be as high as possible so that more fine-grained elements could be pruned away.

To guarantee safeness, we perform the same statistical debugging process on partially instrumented subject programs. Specifically, only a small subset of elements in the entire subject program are instrumented. The debugging algorithm then produces the current top-k suspicious elements each with their respective suspiciousness values. We set the threshold θ to the suspiciousness value of top-kth element. Since this small subset of elements chosen for instrumentation is a proper subset of the set of elements instrumented

in the original approach, the top-k elements returned here will also be returned by the original approach, except that their suspiciousness values might not all be within the top-k. θ thus set is safe because it is a lower bound of the actual top-kth suspiciousness value.

To ensure effectiveness, we introduce another coarse-grained measure C_r in HI such that the C_r value of a coarse-grained element is highly correlated with the F values of enclosing fine-grained elements. In other words, if the C_r value of a coarse-grained element is high, then it is quite likely that the F values of the enclosing fine-grained elements are high as well. Thus, based on this correlation, and the fact that the value of θ is obtained from the F value of the top-kth element obtained at previous iterations, we are likely to obtain a high threshold by giving the priority of performing fine-grained instrumentation to those functions with high C_r values.

There are several possible ways to obtain C_r. For instance, we can adopt the fine-grained suspiciousness measure F as C_r. This has been employed in our earlier work [7]. There, we also validated our belief that the C_r value of coarse-grained element is highly correlated with the F values of their respective enclosing fine-grained elements. However, this solution does not work properly when the fine-grained measure depends on other variables, besides negative and positive support, which are not meaningful at coarse granularity[1]. In that case, a possible solution is to adopt a modified version of fine-grained measure with reduced dimensions. We will discuss it in detail in Section 4.3.

4. ITERATIVE BUG ISOLATION VIA HI

Based on the systematic HI technique discussed in Section 3, we propose an iterative bug isolation approach via HI. In brief, our approach aims to discover the top-k suspicious predictors at less instrumentation effort, while ensuring lightweight instrumentation throughout.

Algorithm 2 depicts the pseudo code of our approach which consists of a coarse-grained phase followed by an iterative fine-grained phase. At the first phase, a coarse-grained instrumentation is performed to instrument all the function entries of the entire program (Line 1). The instrumented program is then deployed in the field to collect sufficient execution data from a large number of end-users (Lines 2-3). We analyze the collected execution data (i.e., profiles) and eventually obtain a list of functions with their respective execution information (i.e., the number of failing and passing profiles containing the function) (Line 4). This list provides the coarse-grained execution information which will be utilized to guide the iterative fine-grained instrumentation in the second phase. The second phase is an iterative phase. At each iteration, we instrument at most one function at fine granularity. We select the function with the highest ranking value (i.e., C_r value) in the current list (Line 8). Based on the feedback θ which is the highest suspiciousness value (*Importance* value) explored so far, we verify the necessary condition for function m (by comparing its C_p value against θ) to contain predicates whose *Importance* value can be greater than or equal to θ. If the necessary condition is not satisfied, we skip this function m and proceed to the next iteration (Lines 10-12). Otherwise, we instrument all predicates within m according to the instrumentation scheme of CBI discussed in Section 2.1 (Line 13). Similarly, we redeploy the instrumented program and wait for sufficient execution data to be collected (Lines 14-15). We next compute the *Importance* value of predicates in m and update the top-k predicates P_k (Line 16) as well as the top-kth *Importance* value θ (Line 17). If the current list of functions is not empty, we proceed to the next iteration. It is worth noticing that each function in the list can be examined at most once as it is removed from the list after it is considered (Line 9). We will provide the detailed discussion for each step in the remaining section.

[1]For example, *Importance* depends on $p_t(e)$ and $n_t(e)$, which are not meaningful at function level.

Algorithm 2: Iterative Statistical Bug Isolation via HI

```
   // coarse-grained instrumentation and analysis
 1 Instrument all function entries in the entire program;
 2 Deploy the instrumented program;
 3 CollectSufficientData();
 4 list ← AnalyzeCoarseGrainedProfiles();
   // fine-grained instrumentation and analysis
 5 θ = 0;
 6 P_k ← ∅;
 7 while list ≠ ∅ do
 8 |  m ← function with the highest C_r value from list;
 9 |  list ← list − m;
10 |  if NecessaryCondition(m, θ) is false then
11 |  |  continue;
12 |  end
13 |  Instrument all predicates in function m;
14 |  Deploy the instrumented program;
15 |  CollectSufficientData();
16 |  Update(P_k, predictors in m);
17 |  θ = getTopKthImportance(P_k);
18 end
19 return P_k;
```

4.1 Instrumentation

The HI technique involves two different levels of instrumentation. At the coarse-grained instrumentation level (Line 1), we only instrument the function entries in the whole program. There is only one instrumentation site (function entry) per function, thus fairly lightweight. For each function entry, we count the number of times this function is called during a run. After a sufficient number of end-users running the instrumented program, we successfully collect the required number of coarse-grained profiles. Each profile consists of a set of functions which are called at least once, as well as a label marking this run as failing or passing.

For fine-grained instrumentation (Line 13), we adopt the instrumentation scheme of CBI discussed in Section 2.1 where three types of instrumentation sites are considered. Obviously, the number of sites (or the number of predicates in these sites) is much larger than that of function entries in coarse-grained instrumentation. However, here we only instrument the sites within one function at one iteration. The performance overhead is much lower than that of sampling, as validated in our experimental evaluation (Section 5.4).

4.2 Pruning Measure Calculation & Necessary Condition Derivation

By employing the systematic approach presented in Section 3.1, we derive a specific coarse-grained pruning measure and

use it to derive a necessary condition for the purpose of pruning (Line 10).

Concretely, given the fine-grained suspiciousness measure *Importance* which can be regarded as a function of four variables in natural number domain, i.e., $Importance = F(p, n, p_t, n_t)$ where $p \in [0, P], n \in [0, N], p_t \in [0, p], n_t \in [0, n]$. (Please refer to Section 2.1 for notations.) The coarse-grained pruning measure can be defined recursively as follows:

$$C_p(p,n) = \begin{cases} M(0,0) & \text{if } p = 0, n = 0 \\ \max\{M(p,0), C_p(p-1,0)\} & \text{if } p \in (0,P], n = 0 \\ \max\{M(0,n), C_p(0,n-1)\} & \text{if } p = 0, n \in (0,N] \\ \max\{M(p,n), C_p(p-1,n), \\ \quad C_p(p,n-1)\} & \text{if } p \in (0,P], n \in (0,N] \end{cases}$$

and

$$M(p,n) = \max\{F(p,n,p_t,n_t) \mid p_t \in [0,p], n_t \in [0,n]\} \quad (7)$$

Notice that in this particular case, the fine-grained suspiciousness measure *Importance* is of four dimensions whereas the coarse-grained measure only has two[2]. We reduce the dimensions just by computing the maximum over the entire domain of additional variables p_t and n_t, as shown in Equation (7). Consequently, we are able to employ dynamic programming (Algorithm 1) to conduct our coarse-grained suspiciousness measure.

Having the coarse-grained measure C_p, let e be a predicate, m be the function where e is located, and a threshold value θ, we can readily deduce the following implication.

$$Importance(e) \geq \theta \implies C_p(p(m), n(m)) \geq \theta$$

In other words, $C_p(p(m), n(m)) \geq \theta$ is a necessary condition for function m to contain predicates with *Importance* value at least θ. Consequently, when this necessary condition is *invalid*, we can safely skip function m for fine-grained instrumentation.

4.3 Ranking Measure Calculation

In order to effectively prune away unnecessary instrumentation, a coarse-grained ranking measure is required (Line 8). As we mentioned earlier, *Importance* measure has additional variables (p_t and n_t). Here, we propose to adopt the pruning measure derived above as the ranking measure. We will validate its effectiveness in the following.

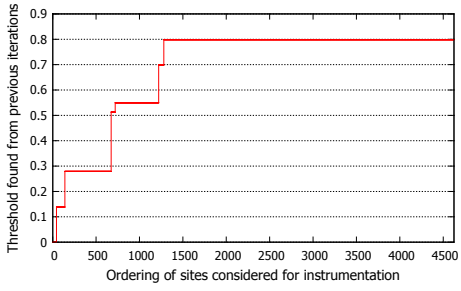

Figure 2: Threshold used for pruning versus ordering of sites considered for instrumentation

[2] *Importance* distinguishes whether a predicate is evaluated to true or false (i.e., $p_t(e)$ and $n_t(e)$). However, such values are not meaningful at function level.

Let's consider the case where we are looking for top-1 suspicious element. We order each instrumentation site by the iteration in which it was considered for instrumentation; i.e., sites with lower order are considered for instrumentation at earlier iteration. Sites considered in the same iteration are randomly ordered. Figure 2 plots the threshold used when a site is considered for instrumentation versus the site order for a faulty version of subject *space*. Ideally, we want threshold to be set sufficiently high for as many sites as possible, because this will increase the chance for necessary condition to be falsified, and the instrumentation of sites be pruned. As such, it is desirable to have a plot in which the threshold raises very quickly and stabilizes itself at a high level.

To quantify the effectiveness of ranking measure, we represent the curve in Figure 2 by a function $\theta = f(s)$ under the integer domain ($s \in [1, S]$) and real number codomain ($\theta \in [0, \theta_{max}]$) where S and θ_{max} denote the total number of sites and the maximum threshold found in the entire program, respectively. Given this function, we introduce a metric called *threshold mass* which is defined as follows:

$$threshold\ mass = \sum_{s=1}^{S} f(s)$$

Note that the value of *threshold mass* is exactly the area under the curve. We then define the effectiveness of ranking measure as the following ratio:

$$ratio = \frac{threshold\ mass}{\theta_{max} \times S}$$

The higher the ratio, the more effective the ranking measure is. To test the practicality of our adopted ranking measure (which is the same as the pruning measure, as mentioned earlier in this section), we compute this ratio for multiple subject programs. Table 1 shows the averaged ratio over all the versions in each subject. On average, the ratio is about 80%. This indicates that, in practice our ranking measure can guide the iterative process to raise the threshold to a relatively high level at very early stage.

Table 1: Ratio of threshold mass

Subject	Siemens	space	grep	sed	gzip	Overall
Ratio	0.53	0.70	0.84	0.97	0.94	0.80

4.4 Sufficient Data Collection

Statistical bug isolation relies on huge amount of data to ensure the stability of results. In other words, given a sufficiently large number of execution profiles, we can obtain the same result (e.g., the same ranked predictors) even if the profiles under analysis are different. In practice, we can wait for sufficient user executions until a stable result is achieved. Specifically, once we have collected a good number of profiles, we can start generating a result. Next, we can continue to wait for the arrival of additional number of profiles and analyze all the profiles collected so far to get a new result. We keep this iterative process until we always achieve the same result in the recently consecutive iterations. Although this is not the main focus of our paper, we will adopt this strategy in our experiments discussed later in Section 5.2.

5. EMPIRICAL EVALUATION

In this section, we evaluate our approach by conducting experiments on real world programs.

5.1 Experimental Setup

The effectiveness of statistical bug isolation has been validated by lots of prior work [2, 3, 9, 10, 11]. We thus mainly focus on validating the performance of our technique in reducing instrumentation effort in Section 5.2 and runtime overhead in Section 5.4 compared with non-iterative statistical approach [3]. In addition, we also experimentally compare our approach with adaptive bug isolation [5] in Section 5.5.

Table 2: Characteristics of subject programs

Subject	Versions	LoC	Functions	Sites	Tests
printtokens	7	726	18	211	4,130
printtokens2	10	570	19	243	4,115
replace	31	564	21	543	5,542
schedule	9	412	18	210	2,650
schedule2	9	374	16	252	2,710
tcas	40	173	9	490	1,608
totinfo	23	565	7	268	1,052
space	34	6,199	136	4,620	13,585
grep	12	10,068	130	39,427	809
sed	16	14,427	169	17,200	363
gzip	9	5,680	91	30,453	213

We have conducted an empirical evaluation using the subject programs from SIR repository [12]. Table 2 lists our subjects used, number of faulty versions, lines of code, number of functions, number of instrumentation sites, and the size of test suite used. The *Importance* measure we adopted as the fine-grained suspiciousness measure is not applicable if the faulty version fails in less than two test cases. Such versions were omitted and not included in Table 2.

To mimic a real deployment, we randomly choose a subset of test cases at each iteration according to the data collection strategy proposed in Section 4.4. Test cases used at different iterations are mostly different. This is quite similar to the practical situation where no two user executions are exactly same.

5.2 Instrumentation Effort

We take the traditional, non-iterative statistical bug isolation (Liblit et al. [3]) as the baseline approach and identify top-k suspicious predictors using the entire test suite. In order to verify the performance of our approach, we measured the total instrumentation effort required, i.e., the percentage of sites instrumented to obtain the same top-k predictors. The smaller the percentage, the better the performance of our approach is. Here we set k to 1.

Table 3: Average number of iterations and average number of sites instrumented per iteration

Subject	Funs_total	Iterations	Sites_total	Sites_per
Siemens	16	12.7	317	36.0
space	136	75.3	4,620	37.3
grep	130	48.6	39,427	444.8
sed	169	63.6	17,200	91.7
gzip	91	24.2	30,453	439.1

Figure 3 plots the percentage of sites instrumented to find the top predictor for each subject (averaged across all versions). Overall, we can guarantee that nearly 50% of instrumentation sites can be pruned away using our approach even without developers' effort. Our iterative approach performs better for larger programs. For sed and gzip, only 30% of sites need to be examined, 70% are skipped. For Siemens, all the subjects are of small size and most of the functions are executed in both failing and passing runs. It is hard to distinguish them based on the coarse-grained information. Therefore, more than 90% of sites have to be instrumented. In more detail, Table 3 shows the average number of iterations required (Column "Iterations") to find the top predictor and the average number of sites instrumented during an iteration (Column "Sites_per").

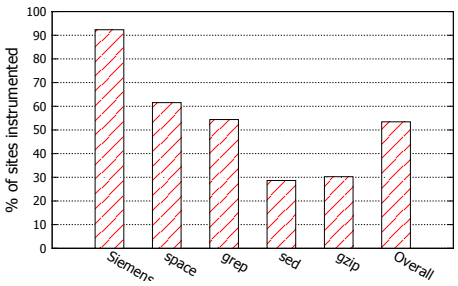

Figure 3: Percentage of sites instrumented

5.3 Stability of Results

According to the data collection strategy stated in Section 4.4, we randomly select a subset of test cases at each iteration. Such a data collection strategy is effective in practice due to a large pool of data available from users. In our experiment though, the number of test cases can be limited, and there might be a challenge to obtain stable experiment result; i.e., our iterative approach might not be able to attain the same top predictor as the baseline approach (i.e., the traditional, non-iterative approach). In this section, we discuss the success rate of our approach in attaining the desired result using the data collection strategy in our experiment.

Table 4: Average number of successful runs and average number of test cases used per iteration

Subject	Runs	Tests_total	Tests_per(#)	Tests_per(%)
Siemens	80.78	3,115	1,823	0.59
space	81.79	13,585	8,559	0.63
grep	81.83	809	542	0.67
sed	83.56	363	232	0.64
gzip	66.89	213	185	0.87
Overall	78.97	3,617	2268	0.68

Specifically, we ran our approach for 100 times and measured how many times our approach can successfully obtain the same top predictor as identified by the baseline approach. Table 4 shows the average number of successful runs (Column "Runs") among all 100 runs, and the average number of test cases used at each iteration (Column "Tests_per(#)"). Column "Tests_per(%)" indicates the ratio of the number of selected tests at each iteration (Column "Tests_per(#)") to the total number of tests (Column "Tests_total") we have.

As can be seen, we can get the same top predictor as the baseline approach in most of runs, overall near 80%. gzip has a quite low success rate. That is because the test suite

available for gzip is very small and thus it is difficult to achieve the really stable result for each iteration.

5.4 Performance Overhead

We compared the user's time overhead of our (non-sampling) fine-grained iterative instrumentation against that of the sampling scheme [3] with different sampling rates (1/1, 1/100, 1/10000). In addition, we measured the time overhead of our coarse-grained instrumentation scheme (shown as Column "coarse"), where only function entries are instrumented without sampling. We run each subject four times, ignore the first run and compute an average execution time of the other three runs.

Table 3 shows that only a few sites instrumented during each iteration (shown as Column "Sites_per") with respect to the total number of sites (Column "Sites_total") in the program. This provides an empirical evidence that our approach has rather low user's time overhead. We further validated this expectation by assessing the average execution time for executing the fine-grained instrumented program at each iteration, as well as the execution time for coarse-grained instrumentation in the first phase of our approach. Table 5 presents the time overhead for running the instrumented programs of our approach (Column "HI") and that of the sampling (Column "Sampling"). All the instrumentation in our experiments is performed via *sampler-cc* developed by Liblit et al. [2] using two different schemes: sampling and non-sampling.

Table 5: Time overhead

	Sampling			HI	
Subject	1/1	1/100	1/10000	coarse	fine
Siemens	0.695	0.670	0.659	0.424	0.595
space	2.172	2.284	2.296	0.396	0.459
grep	21.960	13.042	9.248	0.211	0.389
sed	4.946	3.301	2.824	0.229	0.303
gzip	4.870	2.427	1.793	0.050	0.075
Overall	6.929	4.345	3.364	0.262	0.364

Overall, our fine-grained iterative instrumentation (Column "fine") suffers a much lower overhead than sampling. Especially for grep, sed and gzip, the overhead of our approach is at least an order of magnitude smaller than that of sampling. For Siemens, there is no significant difference between our instrumentation and the sampling schemes. This is because the programs are of small size and we instrument most of the predicates at each iteration. As for our coarse-grained instrumentation (Column "coarse"), its overhead is even smaller than that of the fine-grained iterative instrumentation.

5.5 Performance Comparison with Adaptive Bug Isolation

We also compared our approach with adaptive bug isolation [5]. Recall that the adaptive approach requires developers to check and verify the identified predictors for bug at each iteration. This is in stark contrast to our approach, where developers only need to check the top-k predictors *at the end of all iterations*. For instrumentation effort saved, without developers' intervention, the adaptive approach can not save any instrumentation effort. In this sense, our approach (50% saving) definitely outperforms theirs. For performance overhead, the adaptive approach in general performs better than ours. At each iteration, it only instruments a small fraction of predicate sites on the control flow graph, while our approach instruments all the sites within one function. As a result, our approach could suffer from higher time overhead than theirs. However, on the other hand, their approach requires running more iterations than ours (more than 3 times as many as ours in general). This further increases the developers' burden for checking predictors. Consequently, the debugging process will also take longer time.

6. DISCUSSION

As an iterative approach, Algorithm 2 needs to run the bug isolation process multiple iterations. Consequently, it increases the waiting time for results, but relieves users of running instrumented programs with high overhead. The productivity trade-off between users and developers is common in approaches that adopt crowd-sourcing. In what follows, we suggest several ways to better balance this trade-off.

Multiple Deployment. Since we have a great number of users, we can simultaneously deploy multiple different instrumented programs to different users for execution. As a result, the waiting time for execution data can be reduced. Specifically, we can identify n functions which are of high ranking values and can not be pruned away. We separately instrument the predicates in each of n functions and eventually obtain totally n instrumented programs which are then deployed at the same time. As a consequence, developers can collect execution data with less waiting time.

Multi-function Instrumentation. While only the predicates in one function are instrumented in one instrumented program, it does not stop us from doing fine-grained instrumentation on multiple functions in one instrumented program. This particularly works well for functions of small size, where such additional performance overhead remains tolerable to end users. Thus we can reduce the number of iterations. Developers can obtain the final top-k predictors earlier.

Multiple Hierarchies. So far, we only consider two levels in the hierarchy: function-level and predicate-level. Our HI technique need not be restricted to these two levels. According to practical requirements, HI can be extended to handle multiple hierarchies (e.g., package-level, class-level, function-level, block-level, predicate-level). For example, if a function which is subject to fine-grained instrumentation happens to be a big piece of code, the overhead of running its fine-grained instrumented counterpart may be unacceptable. In this case, we can perform instrumentation with more hierarchies (e.g., first function-level, then block-level and finally predicate-level). By instrumenting the program at multiple hierarchies, we can reduce the overhead incurred at each iteration, while saving the monitoring and computational cost.

7. RELATED WORK

We provide a general review on automated debugging in this section.

Statistical Bug Isolation. Statistical bug isolation is a family of approaches which locate the root cause of failure by analyzing the discriminative behavior between passing and failing executions. The rationale is that program enti-

ties which are frequently executed by failing executions and rarely executed by passing executions are very likely to be faulty. Different types of entities are considered to represent the execution behavior in different approaches. In addition, researchers have proposed different measures to assess suspiciousness of entities. Tarantula [13, 14] and Ochiai [15, 16] both use statement coverage information to represent the execution behavior and assess the suspiciousness for each statement based on their proposed measure. Liblit et al. [3] collected runtime values of predicates and introduced *Importance* to measure each predicate. Similarly, SOBER [10] discusses a different statistical model to evaluate predicates. Since runtime predicate value provides finer-grained execution information than statement coverage, more precise results can be achieved. However, it suffers from heavy instrumentation cost and performance overhead. In [3], sparse random sampling [2] is adopted to address the overhead issue. Gore et al. [17] later extended [3] by introducing elastic predicates such that statistical bug isolation is tailored to a specific class of software involving floating-point computations and continuous stochastic distributions. Furthermore, causal inference has been recently applied to reduce the control and data flow dependence confounding bias in statement-level [18, 19] and predicate-level [20] statistical bug isolation. Our approach aims to reduce the instrumentation effort and performance overhead using hierarchical instrumentation. It is independent of the entity kinds and suspiciousness measures employed by statistical models. Therefore, most of the statistical approaches discussed above can benefit from our technique.

Adaptive Bug Isolation. Our approach is most related to two adaptive bug isolation work, HOLMES [4] and Adaptive Bug Isolation [5]. They share the same objectives of saving instrumentation effort and eliminating performance overhead by iterative instrumentation and analysis, but differ on how to conduct iterative instrumentation. As mentioned earlier, HOLMES [4] and Adaptive Bug Isolation [5] both utilize static analysis over program dependence graph to capture information which is further used to guide the adaptation. Their performance strongly relies on the principle of locality. In contrast, we devise the hierarchical instrumentation and exploit the coarse-grained information to guide the iterations of fine-grained instrumentation; it enables full automation, and thus frees developers from repetitive intervention of the bug isolation process.

Other Automated Debugging Approaches. Program slicing [21, 22] is one commonly used technique for debugging. Recently, to improve the effectiveness, dynamic slicing [23] is adopted for debugging. Zeller et al. proposed *delta debugging* to isolate the failure-inducing difference in source code [24], inputs [25], and program states [26, 27] between one failing and one passing run. Gupta et al. [28] integrated dynamic slicing with delta debugging to narrow down the search for faulty code, while introducing the concept of *failure-inducing chops*. Similar to delta debugging for program states, Zhang et al. [29] forcibly switched the branch predicate's outcome in a failing run and localized the bug by examining the predicate whose switching produces correct result. In [30, 31], Renieris and Reiss selected from a large number of passing runs a passing run which most resembles the failing run using program spectra, and differentiated the program spectra of these two runs to help isolate the cause of the bug. In addition, some researchers have recently attempted to provide more information than sole buggy statement to better support debugging. Hsu et al. [32] presented RAPID to identify bug signatures. They adopted sequence mining algorithm to discover longest sequences in a set of failing executions as the context. Jiang and Su [11] combined feature selection, clustering, and control flow analysis to identify faulty control flow paths that may cover bug locations. Cheng et al. [33] mined discriminative graph patterns from both passing and failing executions as bug signatures using discriminative graph mining. Since only control flow transitions are considered in [33], bugs not causing any deviation in control flow transitions cannot be identified. To enhance the predictive power of bug signatures, Sun and Khoo [34] proposed *predicated bug signature mining*, where both data predicates and control flow information are utilized. They devised a discriminative itemset generator mining technique to discover succinct predicated bug signatures.

Own Earlier Work. In our earlier work [7], we introduced the concept of *hierarchical instrumentation* and proposed an efficient bug signature mining system via HI to enhance the efficiency of bug signature mining. In the current paper, we enhance the earlier work by formalizing the underlying principles of the HI technique and provide a general and systematic approach for its application. As stated earlier, it constitutes a main contribution of this work. We apply the proposed systematic approach to cooperative bug isolation for field failures [3] and propose an iterative (cooperative) bug isolation system via HI. Note that the statistical debugging approaches which are taken into account in these two work are different. In [7], we mainly focus on in-house debugging, whereas cooperative debugging for field failures is considered in this current work. The different scenarios lead to different performance requirements for these two categories of debugging approaches, and thus distinctly different debugging processes. In addition, since these two works adopted different fine-grained suspiciousness measures, different coarse-grained ranking and pruning measures as well as necessary conditions are constructed accordingly.

8. CONCLUSION

Cooperative bug isolation applies the idea of crowd-sourcing to sample classes of program runtime behavior from a large pool of end-users. Each end-user contributes to bug tracking by running variants of instrumented programs. This approach requires lightweight instrumentation of buggy programs so that it will not overly tax end-users for their help in running the instrumented programs. In this paper, we propose a systematic hierarchical instrumentation technique to perform selective instrumentation. We formalize the principles behind the construction of hierarchical instrumentation, and demonstrate its usefulness by applying it to the latest work in this area – iterative statistical bug isolation for field failures. Specifically, we introduce the notions of coarse-grained pruning and ranking measures, and perform coarse-grained instrumentation on the subject programs at the first place. The coarse-grained execution information thus captured is then exploited to guide the iterative fine-grained instrumentation. In particular, in the iterative bug isolation framework, by leveraging the feedback from previous iterations, the HI technique aggressively prunes away unnecessary instrumentation. The experiments conducted

affirm our claim that the HI technique does not only save total monitoring and computational costs, but also sharply diminishes the performance overhead of end-users.

9. REFERENCES

[1] I Vessey. Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Trans. Syst. Man Cybern.*, 16(5):621–637, September 1986.

[2] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 141–154.

[3] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 15–26.

[4] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44. 2009.

[5] Piramanayagam Arumuga Nainar and Ben Liblit. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 255–264.

[6] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209.

[7] Zhiqiang Zuo, Siau-Cheng Khoo, and Chengnian Sun. Efficient predicated bug signature mining via hierarchical instrumentation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA '14, 2014.

[8] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179.

[9] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 1105–1112. 2006.

[10] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: statistical model-based bug localization. In *Proceedings of the 2005 Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2005, pages 286–295, 2005.

[11] Lingxiao Jiang and Zhendong Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 184–193.

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

[13] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477.

[14] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282. 2005.

[15] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, 2007.

[16] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, 2009.

[17] Ross Gore, Paul F. Reynolds, and David Kamensky. Statistical debugging with elastic predicates. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE'11, pages 492–495.

[18] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 73–84, 2010.

[19] George K. Baah, Andy Podgurski, and Mary Jean Harrold. Mitigating the confounding effects of program dependences for effective fault localization. In *SIGSOFT FSE*, pages 146–156, 2011.

[20] Ross Gore and Paul F. Reynolds, Jr. Reducing confounding bias in predicate-level statistical debugging metrics. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 463–473.

[21] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, July 1982.

[22] J. R. Lyle and Weiser M. Automatic program bug location by program slicing. In *Proceedings of the 2nd International Conference on Computer and Applications*, pages 877–883, 1987.

[23] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG*, pages 33–42, 2005.

[24] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT FSE*, pages 253–267, 1999.

[25] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

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

[27] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[28] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE*, pages 263–272, 2005.

[29] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281, 2006.

[30] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[31] Emmanuel Renieris. *A research framework for software-fault localization tools*. PhD thesis, Providence, RI, USA, 2005. AAI3174662.

[32] Hwa-You Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 439–442. 2008.

[33] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 141–152, 2009.

[34] Chengnian Sun and Siau-Cheng Khoo. Mining succinct predicated bug signatures. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 576–586, 2013.