Low-Overhead and Fully Automated Statistical Debugging with Abstraction Refinement

Zhiqiang Zuo *† Lu Fang† Siau Cheng Khoo§ Guoqing Xu† Shan Lu‡

†University of California, Irvine §National University of Singapore ‡University of Chicago

†{zzuo2,lfang3,guoqingx}@uci.edu §khoosc@nus.edu.sg ‡shanlu@uchicago.edu

Abstract

Cooperative statistical debugging is an effective approach for diagnosing production-run failures. To quickly identify failure predictors from the huge program predicate space, existing techniques rely on random or heuristics-guided predicate sampling at the user side. However, none of them can satisfy the requirements of low cost, low diagnosis latency, and high diagnosis quality simultaneously, which are all indispensable for statistical debugging to be practical.

This paper presents a new technique that tackles the above challenges. We formulate the technique as an instance of *abstraction refinement*, where efficient *abstract-level* profiling is first applied to the whole program and its execution brings information that can pinpoint suspicious coarse-grained entities that need to be *refined*. The refinement profiles a corresponding set of fine-grained entities, and generates feedback that determines what to prune and what to refine next. The process is fully automated, and more importantly, guided by a mathematically rigorous analysis that guarantees that our approach produces the same debugging results as an exhaustive analysis in deterministic settings.

We have implemented this technique for both C and Java on both single machine and distributed system. A thorough evaluation demonstrates that our approach yields (1) an order of magnitude reduction in the user-side runtime overhead even compared to a sampling-based approach and (2) two orders of magnitude reduction in the size of data transferred over the network, completely automatically without sacrificing any debugging capability.

* Work was partially done while the author was with National University of Singapore.

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984005

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—statistical methods; D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, diagnostics, monitors, tracing

General Terms Experimentation, reliability

Keywords Automated debugging, field failures, statistical bug isolation, abstraction refinement

1. Introduction

Despite extensive *in-house* testing, bugs commonly escape to production runs on user machines. To debug *field failures* [12], cooperative statistical debugging [23, 24] has been shown to be particularly promising due to its crowd-sourcing nature. Cooperative statistical debugging profiles program executions at each user site, compares success-run profiles with failure-run profiles, and identifies failure predictors — the top k failure-correlated program entities, such as the outcome of a branch, the value of a variable, the thread interleaving pattern at a shared-variable access, *etc.* [19, 24, 35]. These failure predictors reflect failure root causes and can be helpful in patch design. Although cooperative statistical debugging is promising, severe challenges still exist.

1.1 Problems and Motivation

An open and fundamental challenge in statistical debugging is how to quickly identify a small number of failure-correlated program entities (*e.g.*, needles) from an ocean of entities in any program of a reasonable size (*e.g.*, a haystack). For any type of entities, often referred to as *predicates* in statistical debugging, collecting a complete profile for every production run – *e.g.*, logging the run-time outcomes of all branches – would incur not only unacceptable user-side slowdown but also huge amounts of overhead in data transfer, storage, and analysis, making it impossible for production deployment.

To address this challenge, two approaches have been proposed so far. One uses the random sampling technique [23], with the hope that the predicate profiling cost can be amortized across a large number of users and runs.

While random sampling reduces the overhead the user experiences in *each run* of the instrumented program, it does *not* reduce the aggregated total cost of data collection and analysis from the developer's perspective. In other words, although sampling collects less data from each run at each end-user, to achieve statistical significance, more runs/end-users need to be involved and their data need to be transferred, leading to increased latency for failure diagnosis and delayed patch design. For example, under the common 1/100 or 1/1000 sampling rate, hundreds or thousands more failure runs need to be traced before sufficient predicates get sampled to produce statistically meaningful results [4, 19, 23]. Furthermore, a whole-program sampling infrastructure may lead to a large baseline overhead (*e.g.*, more than 50%) that cannot be amortized through sampling [6].

The other approach uses the heuristics-guided sampling technique [5, 7, 10], with the hope that failure-correlated predicates get sampled earlier than others. Some assume that failure predictors are likely around failure points [5, 7], while others first profile predicates near the program entrance and iteratively search down the control-flow graph based on the failure-correlation metric, informally referred to as *suspiciousness* in this paper, of already profiled branch predicates.

Unfortunately, these heuristics-guided techniques still have limitations. Each of these heuristics naturally only works for some types of bugs and predicates, and may behave worse than random sampling in other cases [5, 7]. More importantly, they can never prune out any predicates while guaranteeing the quality of debugging reports — without exhaustively profiling all predicates, it is impossible for them to know whether the best predictor has been found or not. As a result, some of these techniques simply terminate after a given number of predicates are profiled, sacrificing diagnosis ability [5]. Others require developers to *manually* and *periodically* check diagnosis results to determine whether a good enough failure predictor has been found [7], which is, obviously, a daunting task that most developers would be reluctant to do [32].

Instead of looking for a new sampling strategy, this paper takes on a new quest driven by a key question: *can we prune the predicate space with quality guarantees?* If so, we can fundamentally reduce diagnosis cost without sacrificing diagnosis ability.

1.2 Our Contributions

We propose a general, rigorous, and automated predicate-space pruning technique for statistical debugging. Our technique applies to all types of predicates. It can greatly reduce the number of predicates that need to be profiled and analyzed, while never missing top-ranked failure predictors with *mathematical guarantees*[*]. With the help of the effective pruning, our approach greatly saves the cost of production-run failure diagnosis — the costs of both user-side profiling and developer-side data collection and analysis are reduced, without sacrificing any failure diagnosis capability.

Our technique is inspired by — and formulated as an instance of — the *abstraction refinement* framework [11]. Our key observation is simple: predicates are *concrete* program entities constituting a huge space; profiling and analyzing them directly is doomed to result in a high cost. If we can raise the *abstraction level* by first profiling and analyzing data from coarse-grained[†] program entities (*e.g.*, functions), we may obtain a bird-eye view of how each coarse-grained entity is correlated with a failure. This view may then help us decide, iteratively, (1) which coarse-grained entity should be *refined*, with all the fine-grained entities (*e.g.*, predicates) it represents profiled and analyzed, and (2) which coarse-grained entity does not need to be refined, with all the fine-grained entities it represents pruned away.

Informally speaking, using inexpensive, coarse-grained suspiciousness information, we can identify the top k fine-grained suspicious predicates — the ultimate goal of statistical debugging — efficiently and rigorously by profiling and analyzing only a small portion of the large predicate space.

To carry out this insight, there are two critical questions to answer: (1) given an existing suspiciousness metric I for concrete entities (*e.g.*, the *Importance* metric in [23] for predicates), how to design a suspiciousness metric C for abstract entities; and (2) how to use C to guide the fine-grained predicate profiling and analysis?

1.2.1 Designing Metric C

Answering these questions is not trivial. Let us first consider the first question. Our goal here is to make C as indicative of I as possible so that precise refinement guidance can be obtained from applying C to coarse-grained profiles. An ideal design of C is such that the function with the highest C value must contain the predicates that have the highest I values. Hence, to identify the top k predicates, one only needs to refine and analyze a small number of functions with the highest C values. However, this ideal situation is impossible to achieve, as designing such a C that works for all types of predicates is infeasible.

An alternative way to design C is to use various kinds of heuristic information so that the top bug-correlated predicates (*e.g.*, ranked by I) are *likely* to be inside the most suspicious functions (*e.g.*, ranked by C). However, this is fundamentally no better than heuristics used by previous work.

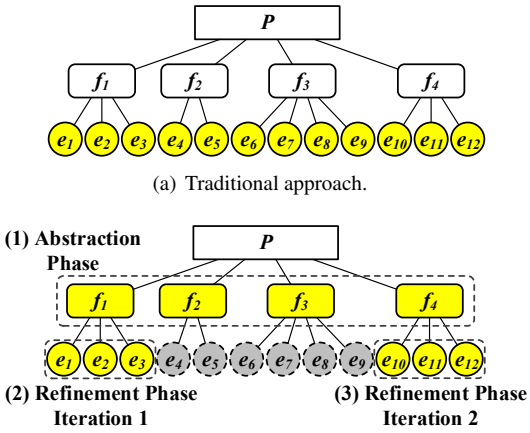
To overcome this challenge, we take a novel perspective that allows us to explore the middle ground. Given a suspiciousness metric I for predicates, we derive C from I with the guarantee that for any function f, $C(f)$ gives the *tightest upper bound* of all $I(e)$ such that e is a predicate in f. This

[*] We will refer to these guarantees several times in the paper, and discuss the assumptions behind them in §6.1.

[†] The terms "abstract" and "coarse-grained", and "concrete" and "fine-grained" are used interchangeably.

Approaches	User-side Overhead	Data Analysis Effort	IsAuto	Quality of Results
Sampling-based statistical debugging [23]	Dep. on sampling rate	Analyze all profiles	Yes	Dep. on sampling rate
Iterative statistical debugging [7, 10]	Low	Analyze selected profiles	No	Dep. on heur. and manual insp.
Statistical performance debugging [35]	Dep. on sampling rate	Analyze all profiles	Yes	Dep. on sampling rate
This work	Low	Analyze selected profiles	Yes	Fully precise

Table 1. A comparison between existing statistical debugging techniques and the proposed work.



(a) Traditional approach.

(b) New approach.

Figure 1. A high-level comparison between the traditional approach and our approach.

upper bound guarantee will be useful in guiding predicate profiling and pruning, as discussed below. The derivation is done by exploring the entire input space of I, and encoding the inputs and outputs of C in a table; details of this derivation can be found in §4.

1.2.2 Using C to Guide Refinement

Following our design of C, we answer the second question with a diagnosis process consisting of two major phases, with the high-level idea illustrated in Figure 1(b).

The first phase conducts simple and light-weight profiling of functions (*i.e.*, abstract entities indicated by the yellow rectangles in Figure 1(b)), collecting a set of *abstract profiles* for functions. These profiles are then used to calculate $C(f)$ for each function f.

The second phase conducts iterative refinement. Each iteration takes a function f from a list $\mathcal{L}$ of all functions sorted on their C values and refines f by profiling *all* predicates in it. For instance, the predicates corresponding to yellow circles in Figure 1(b) are profiled. Since the suspiciousness of f, $C(f)$, provides an upper bound for the I values of all predicates inside f, our diagnosis *automatically* terminates when it has obtained k predicates whose I values are higher than the suspiciousness of all the remaining functions on $\mathcal{L}$, effectively pruning away a large number of predicates (*i.e.*, dashed gray circles in Figure 1(b)) from profiling and analysis. The detailed discussion and formulation of the iterative process can be found in §3.

As a result, our approach is fully automated and fully precise: the mathematical guarantee (*cf.* Theorem 3.1) dictates that when our process terminates, the top bug predictors reported are the same as those that would have been reported by an exhaustive approach that instruments all predicates.

In comparison, the original approach, shown in Figure 1(a), does not profile any functions. Instead it profiles all fine-grained predicates in one phase. A quantitative comparison between our approach and several representative existing techniques can be found in Table 1.

Summary of Results We have implemented the proposed technique for both C and Java. An additional distributed program has been developed for the Java-based implementation, enabling us to evaluate our approach on a 12-node cluster. This makes our results a better reflection of how the technique would be used and how well the technique would perform in practice. To the best of our knowledge, this is the first time that the network traffic and latency is evaluated for a statistical debugging technique in a distributed setting closer to its practical usage.

An evaluation on a variety of real-world C and Java subject programs demonstrates that our approach yields (1) an order of magnitude reduction in user-side runtime overhead even compared to a sampling-based approach and (2) two orders of magnitude reduction in the size of the data transferred over the network, completely automatically without sacrificing debugging capability.

2. Background

The cooperative statistical debugging approach was first proposed in the Cooperative Bug Isolation (CBI) work [24, 25], and has been well researched since then [4, 5, 7, 10, 19, 35]. Its key idea is to collect execution information from both failing and passing runs of production-run software at many end users' sites, and apply statistical techniques to analyze the collected traces and identify likely failure root causes. Most existing statistical debugging techniques only consider one bug or one type of failures at a time. Software companies often conduct triage/bucketing to group traces with similar root causes [15].

2.1 Predicates

Cooperative statistical debugging collects the outcome of *predicates* at run time. A predicate reflects a certain aspect of program state. CBI considers the following three categories

of predicates, which have been shown to be effective for diagnosing a wide variety of software bugs:

- **Branches**: At each conditional, two predicates are tracked, recording which branch is taken at run time;
- **Returns**: At each scalar-returning call site, six predicates are tracked, capturing whether the return value r is $> 0, \geq 0, < 0, \leq 0, = 0$, or $\neq 0$, respectively;
- **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 tracked: $x <, \leq, > , \geq, =, \neq y_i$.

The outcomes of these predicates are obtained through software instrumentation or hardware support [4], and constitute the profile of each run. Finally, a profile consists of a set of predicate counts, each recording the number of times a predicate is observed true during the run. In the statistical model of CBI [24], these counts are *simplified* to indicate (1) whether the predicate has been observed at all (no matter true or false) *at least once* and (2) whether a predicate has been observed *true at least once*, regardless of the exact number of times in the run. Additionally, each profile is labeled *passing* or *failing*, depending on whether it is collected from a passing or failing run.

2.2 Statistical Model and Metric

After the profiles are obtained from many runs, statistical analysis is performed to compute a *suspiciousness* value for each predicate. The top scored predicate is regarded as the best *predictor* of the failure.

CBI [24] uses a metric *Importance* (Equation 1) to assess predicate suspiciousness. This metric has since been used by many other tools. *Importance* is defined as the harmonic mean of *Increase* (Equation 2, measuring how much e being true increases the probability of failure over e simply being observed, no matter true or false) and *Sensitivity* (Equation 3, measuring how much e being true accounts for failing runs). For a predicate e, let $p(e)$ and $n(e)$ be the total number of passing and failing runs in which e is observed, no matter true or false, respectively. Let $p_t(e)$ and $n_t(e)$ be the number of passing and failing runs in which e is observed to be true, respectively. For all these functions, their inputs are natural numbers: $p \in [0, P], n \in [0, N], p_t \in [0, p], n_t \in [0, n]$, where P and N are the total number of passing and failing runs, respectively.

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

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

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

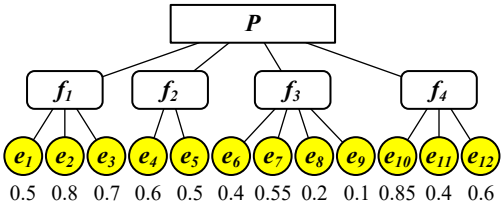


Figure 2. A detailed illustration of the original approach.

Previous work [18, 24, 26] has shown that the top k (k being a small number) suspicious predicates can effectively point out the root cause of many software failures.

2.3 Sampling

A crucial challenge for production-run tools is how to lower the run-time overhead at user sites. To address this challenge, CBI and several other cooperative statistical debugging tools use sparse random sampling [23], which evaluates and records the outcomes of only a sparse and random subset of all predicates during each run. Good failure predictors can still be found under this setting, as long as profiles from many failing runs and passing runs are collected.

The CBI work has found that, without sampling, usually 10 – 20 failing runs are sufficient for identifying failure predictors with statistical confidence. Sampling would greatly increase this number, because only a subset of predicates are profiled during each run. Given the default sampling rate used in previous work, 1/100 – 1/1000, thousands to tens of thousands of failing runs and similar numbers of passing runs are needed to get high-quality failure predictors.

Note that, although sampling reduces predicate evaluation and logging overhead, it introduces overhead due to maintaining a whole-program random sampling infrastructure, which cannot be amortized by sampling. This baseline overhead is often more than 50% for CPU-intensive workloads [6].

3. Abstraction-Guided Statistical Debugging

This section presents our new statistical debugging approach, first through an informal description illustrated by an example in §3.1 and then through an abstraction-refinement based formulation in §3.2.

3.1 Overview

Basic idea At the core of all existing approaches is predicate profiling. While the outcomes of predicates are critical for failure diagnosis, whole-program predicate tracking and handling is too expensive. As discussed in §1, existing overhead reduction techniques all have drawbacks. None of them can automatically reduce the overall cost of predicate tracking and handling, without sacrificing diagnosis quality.

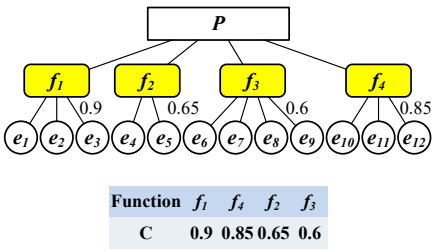


Figure 3. Abstract info collection.

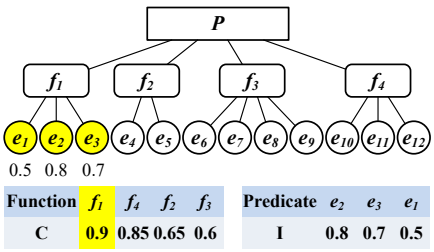


Figure 4. First refinement.

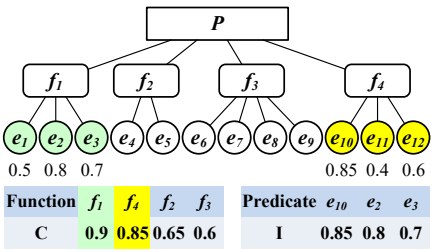


Figure 5. Second refinement.

The goal of our work is to automatically prune the predicate space without sacrificing diagnosis quality, and we will achieve this through lightweight information collected for coarse-grained entities such as functions.

Two-phase approach Our statistical debugging has a two-phase process: an *abstract information collection* phase followed by an *iterative refinement* phase.

In the first phase, instead of profiling predicates, our approach profiles functions, recording which function is executed at least once in each run. This profiling is lightweight and the resulting profiles can be analyzed to obtain the *suspiciousness* score for each function based on a new metric C. We will discuss what is C and how it is designed in §4. Here, we just need to keep in mind that C is derived from the predicate-level suspiciousness metric $Importance$. Given a function f, $C(f)$ is guaranteed to be the *lowest upper bound* of all such $Importance(e)$ that e is a predicate inside f.

The second phase conducts iterative refinement. Each iteration retrieves the top function f from the list $\mathcal{L}$ of all not-yet-refined functions ranked by their suspiciousness values in C, and *refines* f by instrumenting *all* predicates it contains. Note that our instrumentation is *not* accumulative — the instrumentation code added in previous iterations is all removed. The re-instrumented program is executed and, similar to the original approach, $Importance$ is employed to produce a suspiciousness value for each instrumented predicate. At this point, suspiciousness values have been obtained for predicates instrumented in both the current iteration and all previous iterations. These predicates are then sorted and the top k predicates are identified.

At the end of each iteration, we need to decide whether the existing top k predicates are already the top k predicates the original (exhaustive) approach would have produced globally. This decision is made by comparing the lowest suspiciousness value $Importance(e)$ among the existing top k predicates with the highest abstract suspiciousness value $C(f')$ among the remaining unrefined functions in $\mathcal{L}$:

- If $Importance(e) > C(f')$, the debugging process can be safely terminated with the guarantee that the globally top k predicates have been found. The reason is that $C(f')$, which is smaller than $Importance(e)$, is an upper bound for the suspiciousness value of all the remaining unrefined functions and the non-profiled predicates within them. Consequently, all the non-profiled predicates can be pruned at this point without affecting diagnosis quality.
- If $C(f') \geq Importance(e)$, we will enter the next refinement iteration, as there might be a predicate in method f' with a suspiciousness value equal or higher than $Importance(e)$.

Example We use an example shown in Figure 2 – 5 to illustrate the difference between the traditional approach and our approach. Here a program P is represented as a tree structure where leaf nodes e represent predicates and non-leaf nodes f represent functions.

The original statistical debugging technique (Figure 2) profiles all predicates across the whole program through many runs, and computes a suspiciousness value *Importance* for each predicate. Predicates are ranked based on their suspiciousness and the top k predicates are reported. In the example shown in Figure 2, each predicate has a hypothetical suspiciousness measurement and the top 3 suspicious predicates are shown in the bottom table.

Our debugging approach is different. Its first phase is illustrated in Figure 3. This phase profiles every function. The values of the C metric obtained after this phase is shown next to the function nodes.

The second phase starts with its first refinement iteration retrieving the top suspicious function f_1 and instrumenting e_1, e_2, and e_3 in f_1, as shown in Figure 4. The profiles collected by running the re-instrumented program give rise to the following $Importance$ values: 0.8 (e_2), 0.7 (e_3), and 0.5 (e_1). Suppose the goal is to find 3 predicates that are most indicative of the bug. To know whether e_2, e_3, and e_1 are these three predicates, we compare $Importance(e_1)$ (*i.e.*, 0.5) with $C(f_4)$, because f_4 has the next highest suspiciousness value. Since $0.85 > 0.5$, f_4 may contain predicates whose $Importance$ is higher than that of e_1 (*i.e.*, the 3-rd predicate identified) and thus another iteration of refinement is needed. Hence, all predicates in f_4 are instrumented.

The second iteration shown in Figure 5 profiles the three predicates in f_4. Together with suspiciousness values already obtained from the previous iteration, a re-ranking identifies the new top 3 predicates: e_{10}, e_2, and e_3. Because the next function on the list is f_2 and $Importance(e_3)$ (*i.e.*, 0.7) is greater than $C(f_2)$ (*i.e.*, 0.5), we are guaranteed that

e_{10}, e_2, and e_3 must be the globally top 3 predicates and the debugging process can be safely terminated. Clearly, this report is the same as what the exhaustive approach would have produced, although only predicates in two of the four functions are instrumented, profiled, and analyzed.

3.2 Problem Formulation

After the above informal description, we now present an abstraction-refinement based formulation of our approach.

Definition 3.1 (**Abstract and Concrete Entities**). *A program P consists of a set of functions F, each of which is an* abstract entity *for instrumentation. A function $f \in F$ contains a set of predicates E, each of which is a* concrete entity *for instrumentation. An abstraction relation $\alpha : E \to F$ maps each predicate $e \in E$ to a function $f \in F$ containing e. A concretization relation $\gamma : F \to 2^E$ maps each f to a set of predicates inside f.*

In our definition, both the abstraction and concretization functions are straightforward – they are determined by the "containing" relation between functions and predicates. Although this paper focuses on a two-level abstraction hierarchy, one can easily extend our approach to multiple levels, for example, by incorporating other types of entities such as loops and basic blocks. Based on the definitions of abstract and concrete entities, we define dynamic profiles.

Definition 3.2 (**Abstract and Concrete Profiles and Measures**). *A concrete entity profile $\overline{e} \in \overline{E}$ is a five-tuple $\langle e, p(e), n(e), p_t(e), n_t(e)\rangle$, where $p(e), n(e), p_t(e), n_t(e)$ have the same meanings as introduced in §2. An abstract entity profile $\overline{f} \in \overline{F}$ is a triple $\langle f, p(f), n(f)\rangle$. A concrete suspiciousness measure (CSM) Ψ of a program is a pair $(S_e, Importance)$ defined over a set of concrete entity profiles $S_e \in 2^{\overline{E}}$ reversely ordered by $Importance(p(e), n(e), p_t(e), n_t(e))$ for each $\overline{e} \in S_e$. An abstract suspiciousness measure (ASM) Φ of a program is a pair (S_f, C) defined over a set of abstract entity profiles $S_f \in 2^{\overline{F}}$ reversely ordered by $C(p(f), n(f))$ for each $\overline{f} \in S_f$.*

A profile is a statistical record of an entity collected from multiple runs of the program. A concrete profile of a tracked predicate e contains four values $p(e), n(e), p_t(e)$, and $n_t(e)$ that are fed to metric $Importance$ to obtain the suspiciousness value for e. An abstract profile of a tracked function f contains two values $p(f)$ and $n(f)$, representing the occurrences of f in passing and failing runs, respectively. We do not track functions' return values. Consequently, $p_t(f)$ and $n_t(f)$ are not defined.

As shown in the motivating example, at the heart of our refinement-based technique is the metric C. It takes as input two parameters $p(f)$ and $n(f)$ and returns the suspiciousness value of function f. A CSM is essentially a list of predicates ranked by their *Importance* values, while an ASM is a list of functions ranked by their *C* values. Before describing how C is obtained in §4, we first discuss here some important mathematical properties we want C to have.

Definition 3.3 (**Abstraction Soundness**). *An ASM $\Phi = (S_f, C)$ is a sound abstraction of a CSM $\Psi = (S_e, Importance)$ iff $\forall \overline{e} \in S_e : \exists \overline{f} \in S_f$: (1) $(e, f) \in \alpha \wedge$ (2) $C(p(f), n(f)) \geq Importance(p(e), n(e), p_t(e), n_t(e))$.*

The second property indicates that $C(p(f), n(f))$ must be an upper bound of the suspiciousness values for all predicates e inside f. Clearly, the definition of C plays a central role in the development of a sound abstraction. Given an appropriately defined C, we can easily obtain a sound ASM Φ by instrumenting all function entries and collecting function profiles (in the first phase). The subsequent iterative refinement phase is essentially a process of incrementally building the CSM Ψ. Our hope is that with the guidance of Φ, we can find the top k predicates without constructing the complete Ψ.

Lemma 3.1 (**Upper Bound Guarantee**). *Let ASM $\Phi = (S_f, C)$ be a sound abstraction of CSM $\Psi = (S_e, Importance)$. Let $\overline{e_i} = \langle e, \ldots\rangle$ and $\overline{f_i} = \langle f, \ldots\rangle$ be the i-th concrete and abstract entity profile in the (reversely-ordered) set Ψ and Φ, respectively. We have the following guarantee: $\forall$ index $i \in [0, |\Psi|), j \in [0, |\Phi|)$: $Importance(\overline{e_i}) \geq C(\overline{f_j}) \implies \forall$index $k \geq j : \forall$index $t \in [0, |\Psi|) : (\overline{e_t}.e, \overline{f_k}.f) \in \alpha : t \geq i$.*

Proof. Step (1): $Importance(\overline{e_i}) \geq C(\overline{f_j})$ implies that, for any index $k \geq j$, $Importance(\overline{e_i}) \geq C(\overline{f_k})$ due to $C(\overline{f_j}) \geq C(\overline{f_k})$.

Step (2): Since ASM $\Phi = (S_f, C)$ is a sound abstraction of CSM $\Psi = (S_e, Importance)$ and $(\overline{e_t}.e, \overline{f_k}.f) \in \alpha$, based on Definition 3.3, we have $C(\overline{f_k}) \geq Importance(\overline{e_t})$.

From Steps (1) and (2), we have $Importance(\overline{e_i}) \geq Importance(\overline{e_t})$, which indicates that e_t must have a larger index (*i.e.*, t) than e_i (*i.e.*, i) in the CSM. □

Informally, if the suspiciousness value of a predicate e indexed i in the CSM (*i.e.*, $Importance(\overline{e_i})$[‡]) is $\geq$ the suspiciousness value of a function f indexed j in the ASM (*i.e.*, $C(\overline{f_j})$), as long as Φ is a sound abstraction of Ψ, the suspiciousness value of any predicate e' in f or any function lower than f in the ASM must be $\leq$ that of e. Therefore, e' must have a larger index (*i.e.*, t) than e (*i.e.*, i) in the CSM.

Definition 3.4 (**Abstraction Refinement**). *Given a partially constructed CSM $\Psi = (S_e, Importance)$ and an abstraction $\Phi = (S_f, C)$, an abstraction refinement $\sqsubseteq$ produces another pair of CSM Ψ' and ASM Φ' by (1) removing the top function f from Φ, (2) instrumenting all predicates e in f, (3) executing the newly instrumented program, and (4) collecting a set of concrete entity profiles $\overline{e}$ and adding them into Ψ.*

[‡] For simplicity of presentation, we use $Importance(\overline{e_i})$ and $C(\overline{f_i})$ instead of their full notations (with four and two parameters, respectively).

A refinement shrinks an existing ASM by removing its top entry while grows an existing CSM by adding new predicate profiles and re-sorting all contained profiles based on their *Importance* values. It is important to note that, at any step during the refinement, the entities in Φ and Ψ are always *disjoint*: for each function profile in Φ, its corresponding predicate profiles must *not* be in Ψ. In other words, Φ *soundly* abstracts the *complement* of Ψ. The refinement process starts with a full Φ and an empty Ψ and gradually removes abstract entries from Φ and adds concrete entries into Ψ.

Theorem 3.1 (**Termination Quality Guarantee**). *Suppose a chain of i refinement steps results in a partial CSM $\Psi = (S_e, Importance)$ and a partial ASM $\Phi = (S_f, C)$. For a given integer k, if $Importance(\overline{e_k}) > C(\overline{f_0})$, $Importance(\overline{e_k})$ is guaranteed to be greater than the* Importance *values of all predicates in the functions in Φ.*

Proof. Based on Lemma 3.1, if $Importance(\overline{e_k}) > C(\overline{f_0})$, then for all such e_t that $(\overline{e_t}.e, \overline{f_q}.f) \in \alpha$ and $q > 0$, we have $t > k$. In other words, the index of any predicate e_t in a function in Φ must be greater than the index of e_k (*i.e.*, k) in Ψ. Therefore, $Importance(\overline{e_k})$ is guaranteed to be greater than the *Importance* values of all predicates in the functions in Φ. □

Suppose our goal is to find the top k predicates that are most indicative of a bug. This theorem provides a guarantee that if the $Importance$ value v of the k-th predicate in the CSM is $>$ the C value of the first function profile in the ASM Φ, v must be $>$ the C value of any function profile in Φ. Since the C value of a function is an upper bound of the $Importance$ values of all predicates in the function, the theorem further guarantees that none of the functions yet to be refined may contain a predicate that might make the top-k list in the CSM.

4. Abstract Suspiciousness Metric Design

As we have seen, the abstract suspiciousness metric C plays a critical role in our debugging approach. This section discusses the design and computation of this metric.

4.1 Properties and Design Goals

When processing profiles from a set of runs, C takes as input two parameters $p(f)$ and $n(f)$, the number of passing runs and failing runs that have executed function f. It returns the suspiciousness value of f, and has to satisfy several key properties:

- Upper bound. As discussed in §3, requiring $C_f \geq Importance_e$ for all e inside f is a necessary condition to guarantee the correctness of our debugging process.
- Tightness. C has to be a tight upper bound in order to effectively prune the predicate space. For example, making C always return ∞ would turn our technique into an exhaustive approach. The tighter bound C gives, the more efficient our debugging process is.
- Generality. C is expected to work for all types of predicates, including branch predicates, return predicates, and scalar-pair predicates discussed in §2.

Achieving these goals is challenging, as C needs to be computed without profiling any predicates. In the following, we first present our insights around two key design questions, before we present the algorithm of computing C.

What is the input domain of $Importance_e$***?*** We first discuss how to infer the input domains of $Importance_e$ based on function profiles. As discussed in §2, $Importance_e$ takes in four input parameters: $p(e)$ and $n(e)$, which denote the numbers of passing and failing runs that have observed predicate e, as well as $p_t(e)$ and $n_t(e)$, which denote the numbers of passing and failing runs that have observed e to be true.

Answers to this question hinge upon the relationship between the number of runs x that have observed a function f and the number of runs y that have observed a predicate e, among a given set of runs. We only care about whether or not f or e has been observed, not the exact number of times it has been observed in a run as discussed in §2. Consequently, y has to be less than or equal to x when e is inside f — if a predicate is observed, its enclosing function must have been observed. This holds for all types of predicates.

This simple observation allows us to infer the input domain of $Importance_e$ based on the profile of f, when e is in f. That is, when function f is observed in $p(f)$ passing runs and $n(f)$ failing runs, the numbers of passing and failing runs that observe e must be $p(e) \in [0, p(f)]$ and $n(e) \in [0, n(f)]$, respectively. The numbers of passing and failing runs in which e is observed to true, respectively, must be $p_t(e) \in [0, p(e)]$ and $n_t(e) \in [0, n(e)]$.

How to compute the upper bound of $Importance_e$***?*** Knowing the above relationship, ideally, we could directly derive the mathematical formula of C from the mathematical formula of *Importance e.g.*, by computing an equation modeling the area enclosed by the tangent lines of the *Importance* function. However, since *Importance* is a complex function, this derivation requires significant mathematical development and may yield large over-approximations at various points.

One might wonder if the upper bound can be computed by setting the parameters to their bound values. The answer is no because *Importance* is *not* a monotone function. Hence, using bound values as its parameters does *not* guarantee to generate the maximum *Importance* value.

Here we take a different and more practical approach. Given a function profile $\langle f, p(f), n(f)\rangle$, we can enumerate all valid input combinations for $Importance_e$, where e is in f, using the input-domain relationship discussed above. We can then take the maximum of these *Importance* values as C_f. This high-level idea allows us to compute C_f based on

the function profiles collected in our first debugging phase *before* any predicate is instrumented or monitored.

To illustrate, consider a simple example where a function f is executed in one passing run and two failing runs (*i.e.*, $p(f) = 1, n(f) = 2$). e is a predicate inside the function f. Without profiling e, we do not know e is observed in how many passing runs (*i.e.*, $p(e)$) and how many failing runs (*i.e.*, $n(e)$). Fortunately, we know that e cannot be observed in a run unless f is executed in that run. Consequently, we can infer that e is observed in at most one passing run and two failing runs (*i.e.*, $0 \leq p(e) \leq p(f) = 1$, $0 \leq n(e) \leq n(f) = 2$). Hence, we enumerate all possible combinations of $p(e)$ and $n(e)$, namely, $(0,0), (0,1), (0,2), (1,0), (1,1)$ and $(1,2)$, and compute an *Importance* value for each of these pairs. Finally, the maximum of all these *Importance* values is used as the C value for function f. Obviously, this value is an upper bound of the Importance values of all predicates e inside f. Moreover, this upper bound is the lowest upper bound that can be obtained.

4.2 Our Algorithm

This subsection describes our algorithm to implement this idea. A naive algorithm would separately compute C_f for every function, which is extremely time-consuming for large software. Our design is much more efficient.

Our algorithm aims to produce a table that maps (i, j) to $C(i, j)$, where $0 \leq i \leq P$ and $0 \leq j \leq N$ with P and N being the total number of passing runs and failing runs. With this table, we can get the C value for every function f through a simple table lookup using index $(p(f), n(f))$.

To efficiently produce this table, we leverage dynamic programming[§] and an insight that the computation of $C(i, j)$ can be largely simplified leveraging that of $C(i-1, j)$ and $C(i, j-1)$. Algorithm 1 shows the details. Following the high-level idea discussed above, Algorithm 1 computes $C(i, j)$ by enumerating and getting the maximum of all such *Importance*(a, b, l, k) that $0 \leq a \leq i, 0 \leq b \leq j, 0 \leq l \leq a$, and $0 \leq k \leq b$. Line 10 shows the key optimization in our algorithm: to compute $C(i, j)$, we only need to take the maximum of already-computed $C(i-1, j)$, $C(i, j-1)$, and the to-be-computed $maxInLastTwoD(i, j)$. The auxiliary function $maxInLastTwoD$ takes two parameters i and j, and computes the maximum value of all *Importance*(i, j, l, k) where $0 \leq l \leq i$ and $0 \leq k \leq j$. This whole algorithm has an $O(P^2N^2)$ time complexity.

When to run this algorithm? Algorithm 1 can be performed without any knowledge of the targeting software or function; the resulting C-table is valid for all types of software and functions. Consequently, we can re-use the C-table from one debugging process to another, and keep expanding the table when facing larger numbers of passing (P) or failing runs (N).

[§] A natural and efficient way to compute a mathematical property over a large (*e.g.*, multi-dimensional) domain.

Algorithm 1: Computation of the abstract suspiciousness metric C.

Input: Metric *Importance*, the total numbers of passing and failing runs P and N obtained from the first phase
Output: C encoded as a table

```
// Base case
1  C(0,0) ← Importance(0,0,0,0)
2  for i ← 1 to P do
3  |  C(i,0) ← max{maxInLastTwoD(i,0), C(i−1,0)}
4  end
5  for i ← 1 to N do
6  |  C(0,i) ← max{maxInLastTwoD(0,i), C(0,i−1)}
7  end
   // Iterative computation
8  for i ← 1 to P do
9  |  for j ← 1 to N do
10 |  |  C(i,j) ← max{maxInLastTwoD(i,j), C(i−
   |  |  1,j), C(i,j−1)}
11 |  end
12 end

13 Function maxInLastTwoD(i, j)
14 largest ← 0
15 for l ← 0 to i do
16 |  for k ← 0 to j do
17 |  |  largest ← max{largest, Importance(i,j,l,k)}
18 |  end
19 end
20 return largest
```

In our debugging framework, after the abstract information collection phase, we will use the profile of each function f to look up the C-table and obtain the suspiciousness value of f, and then use that to conduct further refinement.

Theorem 4.1 (Lowest Upper Bound). *For each abstract entity profile $\overline{f} = \langle f, a, b \rangle$, a lookup $C(a, b)$ in the table computed by Algorithm 1 returns the* lowest upper bound *of $Importance(i, j, l, k)$ for all such concrete entity profile $\overline{e} = \langle e, i, j, l, k \rangle$ that $(e, f) \in \alpha$.*

Proof. This theorem can be proved by contradiction. Based on the way C is computed in Algorithm 1, $C(a, b)$ must be equal to a particular $Importance(i, j, l, k)$ where $0 \leq i \leq a, 0 \leq j \leq b, 0 \leq l \leq i$, and $0 \leq k \leq j$. If $C(a, b)$ is not the lowest upper bound, there must exist another upper bound less than $C(a, b)$ and the $Importance(i, j, l, k)$ from which $C(a, b)$ is obtained, contradicting its upper bound property. □

The suspiciousness of a function being the lowest upper bound of the suspiciousness of its contained predicates provides a basis for the early termination of the refinement pro-

cess, yielding debugging algorithms that are both precise and efficient.

It is important to note C is computed when the *entire input space* of $Importance$ is considered. In practice, many integer inputs we consider when computing C may not actually appear in a particular set of predicate profiles. For these profiles, there may be a gap between the suspiciousness of a function given by C and the maximum suspiciousness of predicates in the function computed from $Importance$.

The amount of time and space used by Algorithm 1 is reasonable even for large programs. Although the algorithm has an $O(P^2N^2)$ complexity, in our experiments (§7), the computation of the whole metric C takes less than 10 minutes even for the largest settings of P and N (*i.e.*, more than 1000 for P and more than 100 for N, with the sum of P and N shown in the Column "Tests" of Table 2). Furthermore, this algorithm only incurs a one-time cost: invoking it once would generate the whole metric-matrix $C(0..P, 0..N)$. After that, the C metric for every function can be obtained by a constant-time matrix look-up.

4.3 Applicability

Our technique works for all types of predicates and all types of statistical models, as long as the statistical debugging approaches only distinguish whether a predicate has been observed at least once or never in a run. In other words, our technique would fail if debugging relies on the *exact* number of times a predicate is observed in each run — no mathematical relation can be established between the parameters of $Importance$ and C, and thus C is no longer valid.

5. Implementation

We have implemented the proposed approach for both C and Java. Our detailed workflow is as follows:

1. Instrument function entries, deploy the instrumented program, and collect function profiles;
2. Compute C with the numbers of passing and failing runs P and N using Algorithm 1;
3. Compute the abstract suspiciousness measure Φ using C on the collected function profiles;
4. Iteratively refine functions from Φ to build the concrete suspiciousness measure Ψ until termination.

Re-deployment No user involvement is needed in any of these steps: program (re-)instrumentation and (re-)deployment is completely automated. To achieve efficiency in deployment, we do *not* re-instrument or re-deploy the *whole* application in each iteration. We only re-compile the changed component, which is essentially one function selected in each iteration. The component recompiled and redeployed at each iteration is often small in size (demonstrated empirically in Table 5).

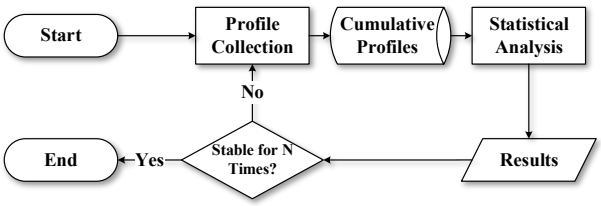


Figure 6. Profile collection strategy.

For a C program, the changed component (*i.e.*, one source file and/or header file) is re-compiled as a patch to a shared library; while for Java, a new class file is generated. Version update is done by utilizing a function wrapping mechanism — we wrap a function in a special way so that calls to this function can be intercepted and rerouted to a specific instrumented version of the function in the library.

Note that re-deployment could be a problem for certain systems that consider dynamic instrumentation as a security risk. For now, we restart all re-instrumented programs after re-deployment. Dynamic software update (DSU) techniques [29, 36] can be employed in the future to avoid re-executing the program.

Profiling sufficiency Like all statistical approaches, statistical debugging relies on a large amount of data to ensure the stability of results. In other words, given a sufficiently large number of execution profiles, the same results (*e.g.*, the same ranking of functions and predictors) can be obtained even if the individual profiles under analysis are different. In our implementation, we need sufficient profiles from both the coarse-grained phase (*i.e.*, Step 1 in the workflow) and each refinement step in the iterative phase (*i.e.*, Step 4).

To achieve the goal, we apply an iterative, fixed point based profile collection strategy, shown in Figure 6. Specifically, we start off by collecting a small number of profiles, on which our analysis is run to generate debugging results. Collection of additional profiles depends on whether the results are stable enough. The arrival of new profiles triggers the re-analysis of all (old and new) profiles and the process is repeated until a fixed point is reached - no difference can be found in the results (*i.e.*, the ranking of functions and predicates) generated in the last N iterations (*e.g.*, $N = 3$ is used in our experiments).

Distributed environment We have created an additional distributed program for our Java-based implementation that runs on a master and a set of slave nodes. The master program simulates the developer who performs the debugging while slave programs simulate users that run the instrumented program. The master instruments the program in each iteration and sends the recompiled class files to the slaves. The instrumentation code records traces at each slave and sends them back to the master over the network. The master then analyzes them and performs refinement. The master-slave communication is done over the socket. Hav-

ing a distributed implementation enables us to measure performance factors such as profile sizes and network traffic, which have never been evaluated in existing statistical debugging techniques.

6. Discussion

This section discusses several important issues that concern the soundness and practicality of the proposed technique.

6.1 Statistical Soundness

It is important to understand whether and under what condition the mathematical guarantees of our debugging process hold. In theory, in order for the abstract suspiciousness to be a sound abstraction of the concrete suspiciousness, the coarse-grained phase and each refinement must execute on exactly the same set of test cases. Therefore, in an in-house debugging scenario for a deterministic system, our approach is mathematically sound.

However, in the cooperative debugging scenario where test cases executed at different phases may be different, our approach cannot achieve mathematical soundness.

To mitigate this problem, our approach currently relies on a large number of profiles to ensure the stability of results, like many other statistical approaches, as discussed in §5. That is, we assume that, with sufficient profiles, the $Importance$ values and hence the ranking among functions and predicates will become stable. This assumption matches with our empirical results — the diagnosis results from our approach are valid in a random distributed setting (§7.3).

A promising direction worth exploring in the future is to turn the mathematical properties stated in §3.2 into *statistical properties*. For example, we can incorporate confidence intervals in our metric design and turn suspiciousness scores into random variables. In this way, suspiciousness comparisons (*e.g.*, whether the $Importance$ value of a predicate is greater than a C value of a function) become hypothesis tests and their results carry statistical meanings.

6.2 Multi-Version Deployment

As an iterative approach, our technique needs to run the multi-iteration refinement process. Like all sampling-based debugging or iterative debugging techniques [4, 7, 19, 23], our technique reduces the user-side overhead of running a heavily instrumented program, but increases the latency of collecting debugging information. This trade-off is widely considered to be worthwhile by previous work, considering the stringent user-side overhead requirement. Since there are a large number of end users, we can simultaneously deploy programs with different instrumentations to different users for execution. As a result, the iterative process can be "parallelized", thereby reducing the waiting time for profiles. Specifically, we can remove top n functions from Φ at a time, refine them to obtain n versions of instrumented programs — each version with one function refined, and deploy them to different user sites. Consequently, developers can quickly collect execution profiles.

6.3 Multi-Function Instrumentation

Our framework can also be extended to refine multiple functions (*i.e.*, instrumenting predicates in multiple functions) at each iteration to reduce the number of iterations. The number of functions to be refined in each iteration essentially defines a trade-off framework — more functions-per-iteration means fewer iterations, but more overhead, less predicate pruning, and more total debugging cost. Expanding all functions in one iteration is essentially whole-program instrumentation. In this work, we expand one function at a time, because reducing user-side overhead is our primary goal.

6.4 Multi-Level Abstractions

In this paper, we only consider two levels of instrumentation in a program. However, our technique can be easily extended to consider other types of program entities, such as basic blocks and classes. For example, if a program has a very large code base and a great number of functions, the cost for running code with function-level instrumentations may not be acceptable. In this case, we may consider class as a new level of abstraction.

As another example, if a program has large functions (*e.g.*, big C programs without a modular design) , the user-side overhead for running even one function with instrumented predicates may be too high. In this case, we can reduce the overhead by adding basic blocks as a new abstraction layer. Before predicate instrumentation, we first examine and prune basic blocks. The refinement process has a nested structure: the iterative refinement of basic blocks would be nested within the refinement of functions. The algorithm for a general n-level abstraction refinement can be easily derived by recursively invoking the current algorithm that refines on a 2-level abstraction hierarchy.

6.5 Detection of Unknown Bugs

So far our discussion has been focused on how to prune predicates when failing runs are seen. To support the detection of unknown bugs that have not yet manifested, we can always enable function-level instrumentation on all or randomly selected runs. Once failures are reported, predicate-level profiling and abstraction refinement are enabled to help locate the cause.

7. Evaluation

This section presents an empirically evaluation of our implementations for both C and Java. Please refer to [1] for the detailed artifact information.

7.1 Benchmarks

While we would like to use the same benchmark set as used by CBI [24], their test suites are not publicly available. As a

Subject	Faulty Ver.	LoC	Functions	Predicates	Tests	Lang.
space	34	6,199	136	25,449	1,248	C
sed	4	14,427	112	101,233	363	C
gzip	13	5,680	91	179,962	213	C
grep	5	10,068	129	228,498	809	C
bash	6	59,846	1,458	928,566	300	C
nanoxml	22	7,646	552	5,128	236	Java
siena	3	6,035	249	19,130	567	Java
ant	4	80,500	8,863	203,576	149	Java
derby	3	503,833	28,887	1,389,976	258	Java

Table 2. Characteristics of the chosen subject programs.

result, we had to resort to the SIR repository [14], as it contains both buggy programs and test suites that were generated both manually and automatically. To select benchmarks, we first found all programs that have at least 5,000 lines of code and then removed those whose test suites contain less than 100 test cases. This left us twelve programs, among which one could not compile and two did not have failing runs. Removing those three resulted in the selection of nine programs, as shown in Table 2.

Each program has multiple *faulty versions* whose number is reported in Column "Faulty Ver.". Each faulty version contains a distinct bug, which is either a real bug or manually injected. A faulty version was excluded if the number of test cases triggering the bug is too small — it would not be possible to obtain any statistically meaningful results. The three types of predicates introduced in §2 (*i.e.*, branches, returns, and scalar pairs) are all considered in our evaluation. To instrument C programs, we employed the *sampler-cc* tool developed by Liblit *et al.* [23]. For Java, we implemented our own instrumenter named JSampler[¶] based on the Soot framework[‖].

Similarly to past work [23], we chose roughly the same numbers of passing- and failing- runs to conduct statistical analysis. Since the goal of this work is *not* to increase the precision of any debugging method, but rather to improve performance with quality guarantees, our evaluation focuses on understanding various performance factors, including, for example, instrumentation effort, user-side runtime overhead, and network traffic and latency.

To have a thorough assessment of these factors, we have designed two sets of experiments, conducted in two different environments. First, we experimented with both our C and Java implementations on a single commodity desktop. This set of experiments focuses on evaluating the instrumentation effort and the user-side runtime overhead incurred by instrumentation code. Second, we ran our distributed implementation on a cluster that simulates real-world usage of statistical debugging of Java programs. The goal is to understand important performance factors such as network traffic and latency; these factors are impossible to measure on a single machine.

[¶] https://bitbucket.org/z_zhiqiang/jsampler

[‖] http://sable.github.io/soot/

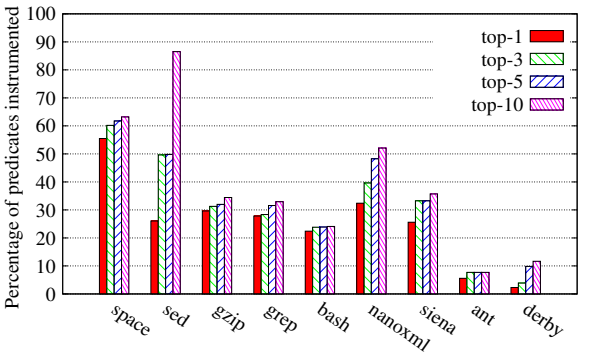


Figure 7. Percentage of predicates instrumented.

7.2 Results on Single PC

Methodology The machine on which the experiments were run has an Intel Core i5 3.30GHz CPU and 8GB main memory, running 64-bit Ubuntu 14.04. Through these experiments, we would like to answer the following four research questions.

- Q1: Compared to the traditional CBI approach where all predicates are instrumented and tracked, how many predicates does our abstraction refinement based bug isolation (ARBI) approach need to instrument and track?
- Q2: How well does our approach perform for different types of predicates?
- Q3: What is the user-side overhead of our approach? How does it compare to the overhead of CBI, with and without sampling [24]?
- Q4: How does ARBI compare to adaptive bug isolation (ABI) [7]?

Q1: Predicates Instrumented We used the traditional CBI as the baseline (that instruments all predicates) and ran it with the entire test suite for each program. For our technique, we first obtained abstract suspiciousness information and then performed iterative refinements by running the same test suite for each iteration until termination. During the process, we measured the percentage of predicates instrumented to obtain the same top k predictors. We assume the test suite is large enough with sufficient failing test cases to make both CBI and ARBI reach the fixed points (*i.e.*, statistically meaningful results).

Figure 7 shows the percentages of predicates instrumented by our approach to find the top k predictors for each program (averaged across all faulty versions), with k being 1, 3, 5, and 10. Note that the baseline always instruments 100% of predicates. Our approach yields an overall 68% reduction in the number of predicates instrumented, without requiring any manual effort. Observe that our approach performs better for larger programs. For very large programs

	Branch		Return		Scalar-pair	
Subject	Iter	Pred(%)	Iter	Pred(%)	Iter	Pred(%)
space	75.8	65.8	73.3	77.7	70.2	58.4
sed	59.5	80.9	57.3	82.0	53.3	86.2
gzip	28.0	39.8	27.2	53.4	25.2	32.0
grep	24.8	35.4	20.8	38.9	22.0	31.5
bash	175.7	20.4	164.3	17.7	173.7	24.2
nanoxml	37.3	42.6	39.1	53.6	17.3	76.4
siena	30.7	31.4	20.3	32.0	26.3	36.8
ant	193.0	10.1	145.5	9.0	71.3	7.1
derby	1328.7	13.0	1281.7	13.6	712.7	10.1
GeoMean	-	33.1	-	33.2	-	30.8

Table 3. Average numbers of iterations needed for each type of predicate to find the top 5 predicates, as well as average percentages of predicates instrumented during the process.

such as `ant` and `derby`, more than 90% of predicates were pruned away.

Although the quality of ranking metric is not a concern of this paper, we manually inspected the top predicates reported for each program. We found that when $k = 5$, the predicates reported do include those truly correlated with the bugs.

Q2: Effectiveness on Different Predicate Types We ran ARBI over the entire test suite by instrumenting one of the three types of predicates (branches, returns, scalar-pairs) at each time. We compared the numbers of refinement steps needed by each type of predicate to reach termination.

Table 3 shows, for each program, the average number of iterations required (Column "Iter") to find the top 5 predicates and the average percentage of predicates instrumented ("Pred(%)") across multiple faulty versions of the program.

Clearly, our approach can effectively prune the predicate space for all the three types. Nevertheless, the effectiveness of our approach for different types of predicates does not differ much. In some large programs, such as `ant` and `derby`, the number of iterations needed for scalar-pair predicates is much smaller than that for the other types. This is potentially because scalar-pair predicates are much more dense than the other types of predicates. If such a predicate is highly correlated with a bug, it can quickly distinguish itself by gaining high suspiciousness.

Q3: User-side Overhead To answer this question, we compared the running time of each program instrumented under ARBI and under CBI with three different sampling rates: $1/1$, $1/100$, and $1/10000$; $1/1$ means no sampling.

Recall that our ARBI is iteration-based and each iteration instruments predicates in only one function. In practice, the program instrumented in different iterations would likely be executed at different user machines and thus each user only needs to pay the overhead of heavy predicate instrumentation in one function. To assess this overhead, we measured the *average* execution time of the instrumented program across iterations (phase 2), as well as the time spent on phase 1 for executing the program with only function entries instrumented.

To precisely measure running time, we ran each faulty version of each program four times and took the average of the execution times of the last three runs. Our results are shown in Table 4. The overhead is measured as the ratio between the execution time of the instrumented run and original run without any instrumentation. Phase 1 and phase 2 under "ARBI" report, respectively, the overhead of the function-entry instrumentation in phase 1 and the average overhead of the predicate instrumentation in phase 2 across iterations. The numbers reported in this table may look different from those reported in [24] and [35] for several common programs used, because we considered three types of predicates in our evaluation while [24] and [35] evaluated performance only with branch predicates instrumented.

	CBI			ARBI	
Subject	$1/1$	$1/100$	$1/10000$	Phase 1	Phase 2
space	2.617	2.517	2.460	1.294	1.384
sed	5.759	4.411	4.179	1.173	1.458
gzip	5.815	3.413	2.783	1.018	1.063
grep	22.228	15.531	12.098	1.215	1.317
bash	2.633	2.486	2.424	1.374	1.362
nanoxml	1.340	1.254	1.252	1.116	1.098
siena	1.247	1.054	1.038	1.021	1.018
ant	11.782	1.591	1.078	1.001	1.026
derby	5.176	1.358	1.204	1.159	1.001

Table 4. User-side slowdown in times ($\times$).

Overall, our iterative refinement (Column "Phase 2") suffers a much lower overhead than sampling even with the lowest 1/10000 rate. This is especially the case for C programs, because they typically have many more scalar-type variables and thus their numbers of scalar-pair predicates are much larger than that of a Java program. For each program, the average Phase 1 overhead is also very low, and often lower than that of Phase 2.

Q4: Quality Comparison with ABI ABI [7] prunes the predicate space using heuristics. In [7], the authors proposed two ABI analyses: a forward analysis that traverses forward the control dependence graph (CDG) from the main entry of the program and a backward analysis that traverses backward the CDG from the buggy point. At every control branch, the suspiciousness of the related branch predicates is used to determine the path to explore. Both approaches may likely run very long if wrong guidance is given by the heuristics. Hence, ABI requires the developer to determine when to terminate, making it difficult for us to come up with a fair comparison with ARBI. To answer question Q4, we design an experiment that compares the *Importance* value of the top predicate found between ABI and ARBI as their predicate instrumentation progresses.

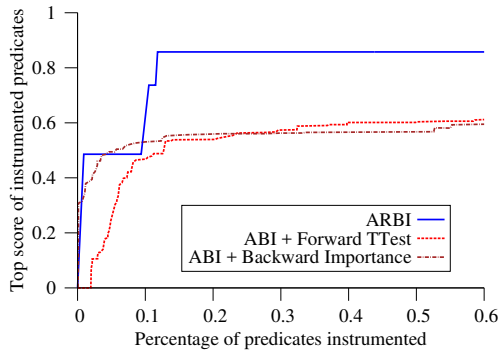


Figure 8. Top *Importance* value comparisons for program `space` between ARBI and two state-of-art ABI approaches. The forward analysis uses *T-Test* as the heuristic and the backward analysis uses *Importance* as the heuristic.

Figure 8 plots three curves for the program `space`, one for ARBI and two for ABI. Each curve represents how the *Importance* value of the top predicate changes as more iterations are executed and more predicates get instrumented. For ABI, we consider two heuristics, `T-Test` and `Importance`, as they are the top performed heuristics according to [7]. We ran the ABI forward analysis with T-Test and its backward analysis with `Importance`. These heuristics determine which control flow path to explore at each branching point.

This figure clearly shows that ARBI can quickly reach a much higher *Importance* value (beyond 0.8) while the two ABI approaches constantly stay below 0.6. Since heuristics do not provide guarantees, they may likely give wrong guidance, making an ABI analysis choose to explore paths that have nothing to do with the bug.

7.3 Results on Distributed Environment

Methodology Our distributed environment is a local 12-node cluster, each node with two Intel Xeon E5 2.60GHz CPUs and 32GB RAM, running CentOS 6.6; nodes are connected by an InfiniBand network. We explore the following question.

- Q5: How much data transferred over the network can be saved by ARBI compared to CBI, and how much is the network latency?

Different from the experiments on a single PC where both CBI and ARBI ran the entire test suite for each benchmark, here we employed the fixed-point-based strategy as discussed in §5 — we kept feeding randomly selected test cases to the instrumented program in each iteration until their results became stable. This is because the goal of this set of experiments is to simulate a real-world usage scenario in which tests are run at many different users and no profiles collected are exactly the same. A technique that can quickly reach the fixed point needs to collect less profiles and thus has lower communication overhead and debugging latency. Since we would like to compare the size of data transferred over the network before reaching the fixed points between different approaches, running the whole test suite in each iteration would defeat this purpose.

In order to answer Q5, we measured the total size of data transferred over the network. We would also like to understand the latency of ABI. However, since ABI cannot terminate automatically, manual inspection is needed in every iteration, making it incomparable with CBI and ARBI, which are completely automated. Work from [7] evaluates ABI using known bugs as "oracles". While we can also use oracles, we decided not to in the experiments because oracles do not exist in real settings and yet the goal of this experiment is to understand ARBI's real-world impact.

	Trace (#/MB)				Binary (MB)		
	CBI		ARBI		CBI	ARBI	Reduction
Subject	Num.	Size	Num.	Size	Size	Size	Total Size
nanoxml	5640	32.5	2520	0.8	0.3	0.4	96.3%
siena	13602	321.3	5773	1.9	0.3	0.4	99.3%
ant	12516	3144.7	10782	5.6	6.2	6.1	99.6%
derby	9288	15350.4	6078	12.1	53.2	43.9	99.6%

Table 5. Data transferred over the network for CBI under the 1/100 sampling rate and ARBI; **Reduction** reports ARBI's reduction in the total data size (*i.e.*, Trace + Binary).

Q5: Network Traffic and Latency Table 5 compares the numbers of profiles (runs) needed and the total sizes of data transferred over the network for each Java program for CBI under the 1/100 sampling rate and ARBI. 1/100 was used to sample predicates because much existing work [24, 35] has reported that this rate achieves a balance between the number of profiles needed (*i.e.*, latency) and the user-side overhead. Column **Trace** reports the numbers and total sizes of profiles collected (*i.e.*, "upstream" data that flows from slaves to the master), while Column **Binary** reports the size of "downstream" data flowing from the master to slaves. The upstream and downstream data consist primarily of traces and recompiled class files, respectively.

Compared to CBI, ARBI requires 40.3% less profiles (runs). When taking into account the size of profiles, our savings become huge – the amount of data transferred over the network by ARBI is on average 1.3% of that by CBI. This is because CBI instruments all predicates while ARBI instruments only predicates in one function at a time. Hence, a profile collected by ARBI is orders-of-magnitude smaller than that collected by CBI. Note that, we can configure ARBI to produce more amount of data per profile in exchange of fewer profiles (runs), by refining more than one function in one run as discussed in Section 6.

We did not measure the network latency, because it would vary a lot depending on network types and configurations. We believe ARBI will have much smaller network latency

than CBI in practical settings, where users and the developers are in different networks.

7.4 Threats to Validity

Threats to external validity arise when the results of the experiment are unable to be generalized to other situations. In this experiment, we evaluated the performance benefit of using our two-phase statistical debugging approach on simulated environments, and thus we are unable to definitively state that our findings will hold for programs in general. For example, the single machine environment is not where the technique is intended to be used. While the distributed environment is based on multiple machines, it uses a very fast network, which a real-world deployment usually does not have.

However, we are confident that these results are indicative of the real-world impacts of our approach. First, we only used the single machine to evaluate how many predicates get instrumented and the user-side instrumentation overhead. Our results should not depend on the execution environment and thus would not change much when our approach is used in a real setting. Second, although the distributed environment uses a fast network, we reported both the total number and the total size of traces needed to complete the debugging process. These numbers would not change in a real setting. In a slower network, much more significant latency reduction should be expected as communication and data transfer is often a bottleneck.

Another potential concern is the limited number of tests used to simulate real user inputs. The statistical soundness of a debugging technique relies on the assumption that there are sufficient failure and success runs collected from users, and they achieve sufficient code coverage. While our test suites are reasonably large and they were generated to cover different behaviors of the program, we cannot guarantee that they are sufficient. Hence, the fixed points reached in our iterative process may potentially be "local" fixed points and thus the correlation between our function/predicate ranking and their true suspiciousness may be spurious.

However, we are reasonably confident that our fixed points reflect truly stable results, because our validation that runs each debugging task for each program 100 times shows that more than 80 times ARBI and CBI produced the exactly same top predictors. For the remaining times, the predictors reported were slightly different but still had large overlaps.

Threats to construct validity arise when the metrics used for evaluation do not accurately capture the concepts that they are meant to evaluate. Our experiments measured the costs involved in running instrumented programs and performing the debugging process in terms of computational time and trace numbers/sizes. Although our results give an indication of the degree of such costs, our implementation can be greatly optimized in both regards. For example, our tracing information is verbose and our implementation is not optimized. However, this limitation does not affect the overall result, as this same implementation was used for both treatment techniques; *i.e.*, the direction and magnitude of the difference between the results should not significantly change when these factors are optimized.

8. Related Work

While there exists a large body of existing work, this section focuses on the discussion of work that is most closely related to our technique.

8.1 Statistical Debugging

Statistical debugging contains a family of approaches attempting to locate the failure root causes by analyzing the discriminative behavior between passing and failing executions. The rationale is that program entities that are frequently executed by failing executions and rarely executed by passing executions are likely to be faulty. We already discussed CBI [23, 24] and its related heuristic-guided sampling approaches [7, 10] intensively. Here we discuss other statistical debugging work that has not been covered.

In addition to the three types of predicates presented in §2, other types of program entities have also been considered in statistical debugging, such as statement coverage [2, 3, 20, 21], and interleaving patterns [19]. In addition to the *Importance* metric, other metrics have also been used to measure suspiciousness [5, 26]. Our framework should still work for those predicates, and can work for different suspiciousness metrics, as long as the metric is related to how many passing/failure runs a predicate has appeared in.

Recently, researchers also aim to improve statistical debugging by providing richer failure-predicting information, referred to as a *bug signature*, than a singleton predicate or statement. Identifying these signatures often requires a more sophisticated statistical approach, such as sequence mining and itemset mining [9, 17, 18, 37]. Each bug signature produced by these techniques often contains a set of statements/predicates.

Among the bug signature research, the most related one is [43] that uses a hierarchical instrumentation based approach to improve the efficiency of bug signature mining for *in-house debugging*. Similarly to our approach, it divides the debugging process into two phases — one that instruments function entries and profiles them to get function suspiciousness measurements and a second that uses these measurements to instrument predicates.

Our approach differs significantly from their approach in two aspects. First, their second phase is not iterative: the predicate pruning is done *only once* using their function suspiciousness; as a result, for almost every program they studied, the percentage of predicates instrumented is very high (60–80%). This immediately disqualifies their approach from being used for cooperative debugging where high overhead cannot be tolerated by users. Second, their function suspiciousness metric is *manually* derived and specific to the

type of predicates and the ranking metric they used for bug signature mining, whereas one core contribution of this work is the *automated* derivation of metric C that is applicable to various types of statistical models and predicates.

8.2 Other Automated Debugging Approaches

Program slicing [28, 38, 41] is a commonly-used technique for debugging. Zeller et al. propose *delta debugging* to isolate the failure-inducing difference in source code [39], inputs [40], and program states [13] between one failing and one passing run. Techniques have been proposed to improve delta debugging by combining it with dynamic slicing [16] and by better selecting the pair of passing run and failing run [33, 34]. Similarly to delta debugging for program states, Zhang et al. [42] forcibly switch the branch predicate's outcome in a failing run and localize the bug by examining the predicates whose switching produces correct result.

Much of the recent debugging techniques focus on finding concurrency bugs in multi-threaded programs. Aviso [27] is a system for avoiding schedule-dependent failures in multi-threaded programs. It uses a statistical model to determine which schedule constraints most effectively avoid failures. Falcon [30] and its follow-up work Unicorn [31] are pattern-based dynamic analysis techniques for fault localization in concurrent programs. They combine pattern identification with statistical suspiciousness ranking of memory access patterns.

RaceMob [22] is a crowdsourced data race detection tool that reduces the overhead and improves the precision by combining real-user crowdsourcing with on-demand race validation. Pacer [8] is a low-overhead sampling-based data race detector that provides a statistical guarantee: it detects races at a rate equal to the sampling rate. The resulting system provides a "get what you pay for" approach that can be tuned by the developer to find races in production systems.

9. Conclusion

This paper proposes a systematic abstraction refinement based technique to debug field failures efficiently. At the abstract level, functions are instrumented to capture abstract suspiciousness information, which will be then exploited to guide the iterative refinement of predicate instrumentation. Different from the existing approaches, our technique provides a mathematically rigorous guarantee of debugging effectiveness without the need of any developers' intervention. The paper also presents a distributed system based implementation of statistical debugging. An evaluation on a 12-node cluster demonstrates that the proposed technique effectively eliminates user involvement, and reduces analysis costs, user-side execution overhead, as well as communication overhead, thereby fully unleashing the power of statistical debugging and making the technique ready for industrial adoption.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable and thorough comments. This material is based upon work supported by the National Science Foundation under grant CNS-1321179, CCF-1409829, CCF-1439091, CCF-1514189, CNS-1514256, CNS-1613023, by the Office of Naval Research under grant N00014-14-1-0549 and N00014-16-1-2913, and by an Alfred P. Sloan Research Fellowship.

References

[1] `https://drive.google.com/file/d/0B58Jj9Us3ouQU2lvTm1XRzhYbm8/view?usp=sharing`.

[2] R. Abreu, P. Zoeteweij, and A. 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*, TAICPART-MUTATION '07, pages 89–98, 2007.

[3] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. Spectrum-based multiple fault localization. In *ASE*, pages 88–99, 2009.

[4] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-run software failure diagnosis via hardware performance counters. In *ASPLOS*, pages 101–112, 2013.

[5] J. Arulraj, G. Jin, and S. Lu. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *ASPLOS*, pages 207–222, 2014.

[6] P. Arumuga Nainar. *Applications of Static Analysis and Program Structure in Statistical Debugging*. PhD thesis, University of Wisconsin – Madison, Aug. 2012.

[7] P. Arumuga Nainar and B. Liblit. Adaptive bug isolation. In *ICSE*, pages 255–264, 2010.

[8] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *PLDI*, pages 255–268, 2010.

[9] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *ISSTA*, pages 141–152, 2009.

[10] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *ICSE*, pages 34–44, 2009.

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[12] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *ICSE*, pages 261–270, 2007.

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

[14] 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.

[15] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. C. Hunt. De-

bugging in the (very) large: ten years of implementation and experience. In *SOSP*, pages 103–116, 2009.

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

[17] H.-Y. Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *ASE*, pages 439–442, 2008.

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

[19] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, pages 241–255, 2010.

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

[21] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[22] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced data race detection. In *SOSP*, pages 406–422, 2013.

[23] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

[24] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, 2005.

[25] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

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

[27] B. Lucia and L. Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *ASPLOS*, pages 39–50, 2013.

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

[29] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. In *PLDI*, pages 13–24, 2009.

[30] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: Fault localization in concurrent programs. In *ICSE*, pages 245–254, 2010.

[31] S. Park, R. W. Vuduc, and M. J. Harrold. Unicorn: a unified approach for localizing non-deadlock concurrency bugs. *Software Testing, Verification and Reliability*, 25(3):167–190, 2014.

[32] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209, 2011.

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

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

[35] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, pages 561–578, 2014.

[36] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: A vm-centric approach. In *PLDI*, pages 1–12, 2009.

[37] C. Sun and S.-C. Khoo. Mining succinct predicated bug signatures. In *FSE*, pages 576–586, 2013.

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

[39] A. Zeller. Yesterday, my program worked. today, it does not. why? In *FSE*, pages 253–267, 1999.

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

[41] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG*, pages 33–42, 2005.

[42] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281, 2006.

[43] Z. Zuo, S.-C. Khoo, and C. Sun. Efficient predicated bug signature mining via hierarchical instrumentation. In *ISSTA*, pages 215–224, 2014.