Adaptive Bug Isolation*

Piramanayagam Arumuga Nainar
Computer Sciences Department
University of Wisconsin–Madison
arumuga@cs.wisc.edu

Ben Liblit
Computer Sciences Department
University of Wisconsin–Madison
liblit@cs.wisc.edu

ABSTRACT

Statistical debugging uses lightweight instrumentation and statistical models to identify program behaviors that are strongly predictive of failure. However, most software is mostly correct; nearly all monitored behaviors are poor predictors of failure. We propose an adaptive monitoring strategy that mitigates the overhead associated with monitoring poor failure predictors. We begin by monitoring a small portion of the program, then automatically refine instrumentation over time to zero in on bugs. We formulate this approach as a search on the control-dependence graph of the program. We present and evaluate various heuristics that can be used for this search. We also discuss the construction of a binary instrumentor for incorporating the feedback loop into post-deployment monitoring. Performance measurements show that adaptive bug isolation yields an average performance overhead of 1% for a class of large applications, as opposed to 87% for realistic sampling-based instrumentation and 300% for complete binary instrumentation.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*statistical methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids; distributed debugging; monitors; tracing*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*graph and tree search strategies; heuristic methods; plan execution, formation, and generation*

General Terms

Experimentation, Measurement, Performance, Reliability

ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...$10.00.

Keywords

statistical debugging, binary instrumentation, dynamic feedback, heuristic search, control-dependence graphs, Dyninst

1. INTRODUCTION

In an imperfect world with imperfect software, debugging does not end the day software is released. Statistical debugging techniques monitor run-time behavior to identify causes of crashes in end-user executions. Lightweight instrumentation [26] allows non-intrusive post-deployment monitoring, while statistical models [1, 17, 18, 26–28, 45] identify profiled events that strongly predict crashes or other failures. Yet most programs mostly work: nearly all code in any given application is not relevant for any given bug. Broad-spectrum instrumentation of many program behaviors, while seemingly necessary to catch a wide variety of bugs, guarantees that almost all data collected is uninteresting. In one study, fewer than 1 in 25,000 instrumented behaviors were reported as failure-predictive [26]. Over 99.996% of each execution profile was discarded, but only after consuming resources (CPU time, network bandwidth, storage space, etc.) that could have been better-used for other purposes.

The problem with current monitoring systems is that they begin with the worst-case assumption that nearly anything could be a clue for a bug, and then continue monitoring events even after statistical analyses show that most are not predictive of failure. Contrast this with the focused debugging activity of an expert programmer. Using feedback from a prior execution, or even just an initial hunch, the programmer uses breakpoints and other probes near points of failure to get more feedback about program behavior. Suspect code is examined more closely, while irrelevant code is quickly identified and ignored. Each iteration enriches the programmer's understanding until the reasons for failure are revealed.

We propose to mimic and automate this process on a large scale. Instead of a single run, we can collect feedback from thousands or millions of executions of the program by its users. Our technique starts by monitoring a small set of program behaviors. Based on analysis of feedback obtained during this stage, our technique automatically chooses other behaviors that could be causing failures and monitors them during the next stage. Throughout this process, statistical-analysis results are available to the programmer, who can fix failures if enough data is available or choose to wait for more data if the picture is unclear. Effectively, we replace sampled measurement of all predicates with non-sampled measurement of adaptively-selected predicates. Non-sampled instrumentation allows faster adaptation by quickly gathering sufficient data where it is most needed. Adaptive instrumentation improves upon existing approaches by prioritizing the monitoring of potentially useful behaviors over those that are less useful, thereby conserving computational resources and bandwidth for both users and developers.

*Supported in part by AFOSR grant FA9550-07-1-0210; DoE contract DE-SC0002153; LLNL contract B580360; and NSF grants CCF-0621487, CCF-0701957, and CNS-0720565. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

Application sizes and bandwidth limits preclude releasing new software each time the adaptive bug-hunting system identifies new instrumentation targets. Therefore, we use binary instrumentation as the chief mechanism for adaptivity. Adaptation decisions are distributed as a list of predicates that need to be enabled or disabled and a binary instrumentor at the user's machine re-instruments the software using this list. Binary instrumentation in itself has several advantages over source-level instrumentation. We can instrument any program, not just those written in languages supported by a source-level instrumentor. We can instrument and monitor a program even when its source code is unavailable. We can also instrument system and third-party libraries used by the application. Truly fixing bugs without source code is difficult, but remediation may still be possible once the causes of failure are identified [22, 23, 32, 35, 36, 42].

Furthermore, binary instrumentation adds only a constant overhead to the size of distributed software: the size of one generic binary instrumentor, usable for all monitored software in a machine. This improves on static sampling schemes whose fast- and slow- path code variants roughly double the size of executables [26], thereby increasing costs for packaging or network distribution.

The remainder of this paper is organized as follows. Section 2 reviews the static (non-adaptive) instrumentation model used in prior work as well as related work in the HOLMES project. Section 3 describes our binary instrumentor and several optimizations developed to reduce the overhead of monitoring. Section 4 describes our Adaptive Bug Isolation technique and Section 5 presents the results of experimental evaluations of this technique. Section 6 discusses related work and Section 7 concludes.

2. BACKGROUND

Before presenting our adaptive approach, we review basic concepts and terminology used in previous work on statistical debugging, including the recent HOLMES [7] project that uses coarse-grained adaptivity. We use a behavioral model based on that of the Cooperative Bug Isolation Project (CBI) of Liblit [24, 25].

2.1 Terminology

The decision of what to monitor is critical, as later analysis can only find bug clues among the data it is given. We follow the sites-and-predicates approach commonly used in prior work [1, 17, 26–28, 45]. An *instrumentation site* is a single program location at which the state of the running program will be inspected. Instrumentation sites are selected automatically based on syntactic features of the code. For example, one might associate one instrumentation site with each function call, or each assignment, or each conditional branch. Instrumentation sites may be both incomplete and mutually redundant with respect to possible program behavior; they are not a perfect execution trace, but rather are a wide net intended to catch useful clues for a broad variety of bugs.

Each site is decomposed into a small collection of *instrumentation predicates* which partition the state space at that site. At a branch instrumentation site, we distinguish between executions that continue along the true versus the false branch. Thus, a branch site decomposes into two predicates. A function return instrumentation site resides in the caller just after the called function returns. Each function return site decomposes the state space into three subspaces, corresponding to three predicates, depending on whether the returned value is negative, zero, or positive. This partition is especially well-matched to C programs, as the sign of a returned value often indicates success or failure of an operation.

For reasons of privacy, and to limit the sizes of execution profiles, prior work avoids reporting predicates as a linear stream of events. Instead, predicates are counted: one counter per predicate, incremented when that predicate is observed to be true. Because each site's predicates partition the space of possibilities at that site, each observation of one instrumentation site increments exactly one predicate counter. Thus, the sum of all predicate counters at a site gives the overall coverage of the site.

Liblit et al. [26] argue that one should not even collect complete predicate counts. Instead, they offer a sampling scheme based on a static source-to-source transformation applied at compile time. This technique, derived from that of Arnold and Ryder [2], creates instrumentation that yields a sparse but fair random subset of the complete counts. Sampling rates of 1/100 to 1/1,000 are typical. This helps preserve privacy and also improves performance in some but not all cases [24]. In exchange, static code size approximately doubles and data analysis becomes more difficult to cope with the fact that 99% or more of requested data is missing. Our adaptive approach eschews sampling in favor of complete measurement of a more selective subset of all possible instrumentation.

A feedback report in the sites-and-predicates model consists of a vector of all predicate counts plus a single outcome label marking this run as good (successful) or bad (failed). In the simplest case, failure can be defined as crashing, and success as not crashing. More refined labeling strategies are easily accommodated, as subsequent analysis stages do not care how the success/failure distinction was made. In particular, failure analysis does not use stack traces, and therefore can be applied to non-crashing bugs.

2.2 Holmes

The HOLMES project by Chilimbi et al. [7] makes two orthogonal contributions to statistical debugging. The first is a new predicate scheme that counts the number of times each path is taken in an acyclic region. The second is a form of adaptive predicate selection at the granularity of functions. Based on partial feedback data, weak predictors are selected, and functions close to them in the program-dependence graph are chosen. Predicates in these functions are instrumented during the next iteration. Here, we propose and evaluate a heuristic search at a much finer granularity. HOLMES also strengthens weak predictors by selecting path predicates in functions containing weak branch predicates. This strengthening is orthogonal to the heuristic searches proposed here or in HOLMES. We present a more detailed comparison with HOLMES in Section 5.

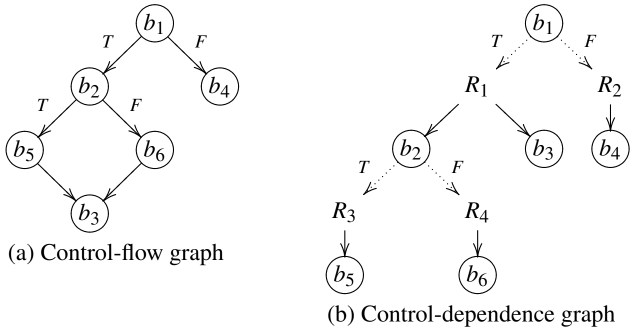
3. BINARY INSTRUMENTATION

Here we discuss the construction of a binary instrumentor that allows incorporating a feedback loop into post-deployment monitoring. We use the Dyninst [5] instrumentation framework, that allows many optimizations that are difficult, and in some cases impossible, to achieve in other tools. This section discusses those optimizations in detail. Unfortunately, even with these optimizations, overheads are too large for deployment to end users (see Section 3.3). This motivates using an adaptive approach, discussed in Section 4, to achieve truly lightweight monitoring.

3.1 Static Removal of Instrumentation

Branch predicate counts are equivalent to the edge profiles of an execution. We use static program structure to avoid redundant operations using approaches similar to those of Ball and Larus [4] or Tikir and Hollingsworth [41]. However CBI collects more than just edge profiles. Coverage of any predicate, such as a function return predicate, implies coverage of the basic block in which that predicate is defined. This in turn implies coverage of the edges leading to that block from all of its control-dependence ancestors.

Figure 1 shows an example control-flow graph (CFG) and the



(a) Control-flow graph

(b) Control-dependence graph

Figure 1: Example graphs for static removal of branch predicates. True and false edges of branches are labeled T and F respectively.

corresponding control-dependence graph (CDG). The additional R_i nodes in the CDG are *region nodes* that group nodes with identical control conditions. For example, nodes b_2 and b_3 execute if and only if the condition at b_1 is true.

Ordinarily, a branch instrumentation site at b_1 would place one predicate counter along the edge from b_1 to b_2 and another along the edge from b_1 to b_4. If there is already any instrumentation site s at b_2 or b_3, then the branch predicate count along the $b_1 \rightarrow b_2$ must equal the sum of the counts of the predicates at s. Thus, the $b_1 \rightarrow b_2$ edge need not be instrumented and its missing edge profile can be computed offline. In general, we define a branch predicate as *redundant* if its count can be inferred offline by a post-mortem analysis of the execution profile. Consider a potential branch predicate p along the edge from some block u to a successor block v. Predicate p is redundant if u dominates v and at least one instrumentation site s is defined in v or any other basic block which is *control-equivalent* to v. The first condition ensures that the entire *control-dependence region* [10] corresponding to v is executed if and only if the edge from u to v is traversed. The second condition ensures that every execution of the control-dependence region containing v will be reflected in one of the predicates from site s. Under these conditions, instrumentation for branch predicate p can be omitted. The count for p can be derived offline by summing all predicates at site s.

3.2 Binarization and Dynamic Removal

While some statistical debugging models use exact values of predicate counts [1, 17, 28, 45], others require only *binarized data*: they consider only whether a predicate was true at least once, but make no further distinctions among nonzero counts [18, 27]. If a binarized model is to be used, then predicate "counters" are merely flags. Instrumentation code can just store a 1, which takes one memory operation, instead of incrementing a counter, which takes one arithmetic and two memory operations.

Furthermore, when using binarized data, there is no benefit from additional observations of a predicate that has already been observed true once. Therefore, a predicate's instrumentation code may be removed from the target once it has triggered [6, 31, 41]. Dynamic instrumentation removal is especially well suited for branch instrumentation, as each branch predicate adds code on a distinct edge and therefore each branch predicate can be removed independently.

3.3 Performance Impact

Instrumentation must have extremely low overhead if we are to collect feedback data from members of the general public. Experiments with a small, CPU-intensive benchmark show that naïve binary instrumentation does not achieve this goal.

We use the SPEC `099.go` benchmark, compiled with gcc 4.1.2, instrumented using a beta version of the Dyninst 6.0 release and run on an otherwise idle dual-core 3.2 GHz Pentium CPU. We use one small (2stone9) and two large (5stone21 and 9stone21) benchmark workloads. All measurements reported are averages across five repeated trials. Execution time excludes instrumentor start-up costs and reflects only time spent running the instrumented code. Start-up costs can be amortized over several runs using the new binary rewriting feature in Dyninst. We instrument all branches and function returns in the main executable but not in shared libraries.

The unmodified `go` executable completes the small workload in 0.4 seconds, and the two large workloads in 21.7 and 21.6 seconds respectively. Naïve Dyninst instrumentation slows execution by a factor of 5.8 times for the small workload and 5.5 times for the large workloads. Adding static branch instrumentation removal, binarized counts, and dynamic branch instrumentation removal increases this relative slowdown to 8.9 for the small workload, but shrinks it to 1.8 for the large workloads: the benchmark's small code footprint means that dynamic branch instrumentation removal is more beneficial for longer-running tasks. While 1.8 is better than 5.5, this is still too slow. Users will not accept a 1.8× slowdown in daily use.

We also consider three non-Dyninst-based approaches: *Pin*, *Valgrind* and *sampler-cc*. Pin and Valgrind are dynamic binary instrumentors that use just-in-time (*JIT*) disassembly and recompilation, as contrasted with Dyninst's code-patching approach. Our custom Pin instrumentor built using Pin version 2.6 has slowdowns between 4.3 and 5.0. Our custom Valgrind instrumentor built using Valgrind version 3.2.1 performs similarly to lightly-optimized Dyninst, with slowdowns between 7.0 and 10.2. However, many of the more aggressive optimization strategies would not be practical to apply under a JIT execution model. Moreover, JIT code patching imposes a baseline overhead to load and execute instructions even when no instrumentation is performed. This limit cannot be improved with any static or dynamic optimization. For Pin, this overhead comes to about 1.9, which is higher than highly optimized Dyninst instrumentation. For this reason, we evaluate our adaptive techniques in the next section using Dyninst. *sampler-cc* is the CBI instrumenting compiler developed by Liblit et al. [26]. sampler-cc instrumentation is highly optimized but is completely fixed at compile time. We ran the instrumented benchmark using the sparsest possible sampling rate, which should give the best performance, and found relative slowdowns of 1.4 across all workloads.

While sampler-cc's static approach is the fastest considered here, we are not willing to give up the benefits of dynamic instrumentation while simultaneously imposing a 40% slowdown on end users. The adaptive techniques described in the next section dramatically reduce instrumentation overheads while simultaneously avoiding the drawbacks of static instrumentation.

4. ADAPTIVE INSTRUMENTATION

A *bug predictor* is any instrumented predicate which is predictive of failure. Numerous statistical debugging techniques have been proposed to find the tiny fraction of predicates that are good bug predictors [1, 17, 18, 26–28, 43, 45]. All assume that instrumentation sites are selected once remain fixed thereafter. The adaptive approach detailed in this section eliminates fixed monitoring plans. Instead, sites are speculatively added to the instrumentation plan if it appears that they may be good bug predictors and are removed from the monitoring plan once their bug-predictive ability (or inability) has been assessed. Our algorithm exploits the principle of locality: if a predicate is highly predictive of failure, then predicates in its

Procedure 1 Pseudo code for Adaptive Analysis

```
$monitored = \emptyset$
$explored = \emptyset$
$plan = GetInitialSet()$
while debugging do
    Instrument and monitor sites in $plan$
    $WaitForSufficientData()$
    $monitored = monitored \cup plan$
    $best$ = branch predicate with highest $score$ in
            $monitored \setminus explored$
    $explored = explored \cup \{best\}$
    $plan = Vicinity(best) \setminus monitored$
end while
```

vicinity are potentially good bug predictors as well. The essence of our technique is to adaptively adjust the instrumentation plan by locating a predicate that is highly predictive of failure and extending the plan to include nearby sites.

Procedure 1 defines this iterative algorithm. It is parameterized by four sub-procedures: *GetInitialSet*, *WaitForSufficientData*, *score* and *Vicinity* — that are described more fully later. *monitored* is the set of sites for which feedback information is available from previous iterations. *explored* is the set of branch predicates that have received the highest score in the previous iteration; these are predicates whose nearby vicinity has already been explored. *plan* is the set of sites that are being monitored during the current iteration. *best* is the branch predicate which receives the highest score in the current iteration. At startup, the analysis chooses the set of sites to be monitored by calling *GetInitialSet*. The set of sites is monitored until the function *WaitForSufficientData* returns, indicating that enough feedback has been collected for meaningful analysis to be applied. Using this feedback, and feedback from earlier phases (if any), the best branch predicate which was not already explored is identified. The plan for the next iteration is to monitor previously unmonitored sites in the vicinity of this best predicate as defined by the function *Vicinity*. The sets *monitored* and *explored* are updated during each phase.

Our choice of algorithms for the sub-procedures *GetInitialSet* and *Vicinity* determines whether the analysis in Procedure 1 is a forward (Section 4.1) or backward (Section 4.2) analysis. The *score* function assigns numeric values to every predicate; a higher value is assigned to a predicate that is a better bug-predictor. We describe the scoring functions that we explored in this paper in Section 4.3 and experimentally evaluate them in Section 5. Some scoring heuristics developed for non-adaptive instrumentation are designed to compute the inherent bug-predictivity of a predicate and prevent the bugginess of nearby predicates from skewing the measure for this score. Such metrics are less suitable for our adaptive algorithm, as they subvert the locality principle on which it relies. *WaitForSufficientData* assesses whether sufficient data has been collected. While not the main focus of this work, we briefly discuss this issue in Section 4.4. Section 4.5 mentions some alternative design choices.

4.1 Forward Analysis of the Program

The general pattern in forward-adaptive bug isolation is to start at the beginning of the program and iteratively work forward toward the root causes of bugs. Consider the control-flow graph in Figure 1a. Suppose the branch predicate associated with the true result of the condition at b_1 is found as the best predicate according to the *score* function. This means that whenever the edge $b_1 \rightarrow b_2$ is traversed, the program is likely to fail. This indicates that there might be some bug in the basic blocks b_2, b_3, b_5 and b_6 and predicates in these blocks may be even better at predicting the bug. However, we do not have enough evidence to believe that b_5 and b_6 have good failure predictors. It could be the case that the bug is in b_5 and hence none of the predictors in b_6 predict failure. It is also possible that the bug is in b_6 and the predicates in b_5 are not relevant. Which of the two cases is true will be known when we have information about the branch site at b_2. We can defer monitoring sites in b_5 and b_6 until we have that information.

On the other hand, we have enough reason to believe that good predictors will be found in b_2 and b_3 because the collected data shows that these blocks are executed in many failed runs. In general, the choice of b_2 and b_3 translates to choosing the children in the control-dependence graph. Thus, if the *best* predicate is associated with the branch at basic block b being true (respectively, false), then we are interested in the basic blocks that are control dependent on b with the true (respectively, false) control condition. Therefore $Vicinity(best)$ returns the sites in these basic blocks. Function calls are handled automatically by using an interprocedural CDG. To fit with the notion of searching forward in the CDG, *GetInitialSet* returns the sites in basic blocks control dependent on the entry node of the CDG.

4.2 Backward Analysis of the Program

If a program fails by crashing, then a stack trace of the program when it crashed may be available. Based on the folk wisdom that the bug is likely to be somewhere near the point of the crash, exploring the predicates near the crash point may find good bug predictors faster than a forward analysis. To illustrate backward analysis, once again consider the CFG in Figure 1a. Suppose the program crashes in block b_3. Now consider the branch site at b_1, which is b_3's control ancestor. This site is the last point where execution of b_3 could have been skipped and hence the crash averted. So, b_1 is a good candidate bug predictor and we monitor it and measure its score.

Suppose the programmer looks at the new feedback and has no idea why a predicate at b_1 could be causing the crash. There are two possible reasons:

case I: The bug may actually be in basic block b_2, b_5, or b_6. The predicate at b_1 may have a high score simply because it governs execution of these blocks.

case II: The branch predicate at b_1 may have a high score because the problem happens before the program reaches b_1. Thus, the program will fail irrespective of the outcome of this branch.

In case I, predicates in b_2, b_5 and b_6 are potential bug predictors. Using the same reasoning used during forward analysis, we only monitor b_2 and delay monitoring of b_5 and b_6 until there is information about the branch predicates in b_2. In case II, we could explore further backwards in the CDG by considering b_1's control ancestor (assuming that Figure 1a shows just a fragment of a larger program). Since there is no way to decide whether the root cause is before the execution of b_1 or after it, we take a conservative approach and include sites suggested by both cases I and II in the monitoring plan for the next iteration.

To summarize, for backward analysis, *GetInitialSet* returns the branch site in the control ancestor of each basic block in which a crash occurs. $Vicinity(best)$ returns sites in the control ancestors and control descendants having the appropriate control condition of the basic block in which the predicate *best* is defined.

4.3 Scoring Heuristics

In this section, we consider possible definitions for *score* as used in Procedure 1. All possibilities considered are heuristics in that one could contrive situations in which they perform badly. Our goal is to identify scoring heuristics that perform well on a variety of programs in realistic situations.

The data collection model of Liblit et al. [27] aggregates the data collected for a predicate p into four values: $S(p)$ and $F(p)$ are respectively the number of successful and failed runs in which p was observed to be true at least once. $S(p\text{ obs})$ and $F(p\text{ obs})$ are respectively the number of successful and failed runs in which p was observed at least once regardless of whether it was true or not. The later two values correspond to the coverage of the site containing p and can be computed offline by examining the counts of all predicates (including p) at that site. Besides these four values, there is another global value: *NumF*, the total number of runs that were labeled as failures. All heuristics described in this section are computed using these values.

Failure Counts The first heuristic scores a predicate p according to the number of failing runs in which p was observed to be true: $\mathit{FailCount}(p) \equiv F(p)$. Any region of code that is executed during many failed runs is potentially buggy. $\mathit{FailCount}(p)$ may not be a good measure of bug predictability because it does not distinguish between two predicates which are true in different numbers of successes but same number of failures. However, a predicate seen in many successful and failing runs may capture some property of the program other than its outcome, such as differing usage scenarios [45].

Importance Liblit et al. [27] argue that a good bug predictor should be true in few successful runs and should also have a significant effect on the outcome on the program. This is captured by the *Increase* metric that measures how much more likely failure is specifically when p is true versus simply reaching the site containing p at all. Formally,

$$\mathit{Increase}(p) \equiv \frac{F(p)}{S(p)+F(p)} - \frac{F(p\text{ obs})}{S(p\text{ obs})+F(p\text{ obs})}$$

The failure count at p is not completely discarded. In information retrieval terms, $\mathit{FailCount}(p)$ measures *sensitivity* or *recall*, while $\mathit{Increase}(p)$ measures *specificity* or *precision*. Liblit et al. [27] balance the two by scoring candidate predicates by the harmonic mean of normalized $F(p)$ and $\mathit{Increase}(p)$ as follows:

$$\mathit{Importance}(p) \equiv \frac{2}{\frac{1}{\mathit{Increase}(p)} + \frac{1}{\log(F(p))/\log(\mathit{NumF})}}$$

Liblit et al. found $\mathit{Importance}(p)$ as a good measure of failure predictivity. Here, we consider it as a candidate adaptive-scoring heuristic.

Maximum Importance During initial experiments, we found that *Importance* as a scoring heuristic often leads to sub-optimal adaptation decisions. For example, consider a branch condition that is always true and the associated branch predicate p. Since the branch is always taken, $F(p\text{ obs}) = F(p)$ and $S(p\text{ obs}) = S(p)$. Thus $\mathit{Increase}(p)$ and consequently $\mathit{Importance}(p)$ are both 0. Even if $F(p)$ is very large, a branch predicate p' with marginally positive values for $\mathit{Increase}(p')$ but very low $F(p')$ will be given preference over p. This is not a problem with the *Importance* heuristic because, as mentioned earlier, there can be situations where the heuristics make choices that go against the principle of locality. If the iterative ranking and elimination algorithm of Liblit et al. [27] is used, where the goal is to find predicates with high *Importance* scores, we can construct a heuristic that maximizes the *Importance* score of predicates in $\mathit{Vicinity}(p)$ rather than the *Importance* of p itself.

Predicates in $\mathit{Vicinity}(p)$ have not already been monitored, so we cannot predict the exact maximum score among predicates in that set. Instead, we compute an upper bound by considering a hypothetical predicate h that has the best possible score. Such a predicate must be true in all the failing runs in which it is observed and false in all the successful runs in which it is observed, i.e. $F(h) = F(h\text{ obs})$ and $S(h) = 0$. When a forward analysis is used, h will appear in a basic block that is control dependent on the edge associated with p and hence h will be observed only when p is true, so $F(h\text{ obs}) = F(p)$ and $S(h\text{ obs}) = S(p)$. Thus,

$$\begin{aligned}\mathit{Increase}(h) &= \frac{F(h)}{S(h)+F(h)} - \frac{F(h\text{ obs})}{S(h\text{ obs})+F(h\text{ obs})}\\ &= 1 - \frac{F(p)}{S(p)+F(p)}\end{aligned}$$

We set $\mathit{MaxImportance}(p) \equiv \mathit{Importance}(h)$ to favor predicates that have the potential to reveal new predicates with high *Importance*.

Student's t-Test The next heuristic, $\mathit{TTest}(p)$, uses a statistical test called the Student's t-test [21]. Given two samples, this test uses the mean, standard deviation and the size of the two samples to assign a numeric confidence in the range $[0,1]$ to the hypothesis that the means of the distributions underlying the two samples differ. We can apply the t-test on the two sample sets we have about a predicate p: the observations of p in successful and failing runs. Let c be the confidence assigned by the t-test for the hypothesis that the truth value of a predicate p differs significantly between successful and failing runs. During a forward analysis, if p is seen in a larger percentage of failures than successes, then the predicates in $\mathit{Vicinity}(p)$ are also observed in a larger percentage of failures than successes and the heuristic should give preference to p. On the other hand, if p is seen in a larger percentage of successes than failures, then the predicates in $\mathit{Vicinity}(p)$ should have lower preference. $\mathit{TTest}(p)$ computes the t-test confidence metric c and assigns a score of c to p in the former case and a score of $-c$ to p in the latter case. Since the information about p does not impose any useful restrictions on the coverage of the ancestor of the node associated with p in the CDG, both *MaxImportance* and *TTest* do not have a sensible interpretation for backward analysis.

Student's t-test is a *parametric* test and assumes that the samples are normally distributed. We also evaluated a *non-parametric* test, the Mann–Whitney U-test [29], which is less powerful but does not assume normal distribution. The U-test always performed worse or only as good as the t-test indicating that the normality assumption is acceptable. Therefore, we give the U-test no further consideration.

Other Heuristics To evaluate the usefulness of our techniques, we also define three other heuristics. To assess whether search heuristics are useful at all, we consider *BFS*, a naïve breadth-first search on the CDG. *Random* is a strawman heuristic that selects a "best" predicate at random on each iteration. Lastly, an *Oracle* heuristic helps to measure how well any search heuristic could possibly do; we discuss *Oracle* further in Section 5.2.

4.4 Waiting For Sufficient Data

Most statistical analyses have the ability to associate a confidence value with their output. A typical confidence measure would be a probability, between 0 and 1, that an observed trend is genuine rather than merely coincidental. Given such information, we can wait until the analysis is able to compute its output with sufficiently high confidence (for example, with probability greater than 0.95). In general, this decision involves a compromise between the speed and accuracy of debugging. Collecting more reports improves accuracy but may take longer to produce interesting results, while collecting fewer reports has opposite trade-offs. Moreover, user behavior might change across iterations and more reports will be needed to accomadate such instability. As mentioned earlier, this issue is not the main focus of this paper. Section 5 explains a simple approach

Table 1: Programs used for experimental evaluation

Program	Variants	LOC	Sites	Test Cases
bash	1	59,846	17,996	1,061
bc	1	14,288	1,799	10,000
ccrypt	1	5,276	757	10,000
exif	1	10,588	2,631	10,000
flex	47	14,705	2,538	567
gcc	1	222,196	56,850	892
grep	17	14,659	2,666	809
gzip	9	7,266	1,406	217
Siemens	7 to 41	173 to 563	94 to 184	1,052 to 5,542
space	38	9,126	1,673	13,585

that we use for our experimental evaluation.

Note that the definitions of *GetInitialSet* and *Vicinity*, as given in either Section 4.1 or Section 4.2, have an important completeness property. With either approach, starting with *GetInitialSet* and repeatedly expanding the search using *Vicinity* will eventually instrument every site that is reachable from the program's entry point. This property is important because the principle of locality is not a guarantee: a good predicate may appear in code where other predicates have low scores. The completeness property ensures that adaptive bug isolation can recover from wrong turns. In the worst case, it will still provide the same information as exhaustive (non-adaptive) instrumentation; it may just take longer to get there.

4.5 Design Alternatives

As proposed here, adaptive instrumentation is a heuristic search through the control-dependence graph, and the principle of locality is assumed to apply to predicates that are close in this graph. However, the effects of bad code can also propagate through data rather than through control flow. Thus, it may be desirable to consider data-dependence relations as well. This can be done by using the *program-dependence graph* (PDG) [14] instead of the control-dependence graph. Balakrishnan et al. [3] have demonstrated PDG construction for unannotated binaries, but the infrastructure to do this is not yet generally available.

Instead of constructing an interprocedural CDG, one could initially treat calls as opaque and only instrument callee bodies if the results they return are highly predictive of failure. However, this incorrectly assumes that called functions are pure, with no effects other than the values they return. This clearly is not true for C. Thus, a completely modular, function-by-function search is not appropriate in the general case.

Procedure 1 describes an automated search which can proceed without human intervention. But the general framework is flexible and can be manually overridden if and when needed. For example, the programmer can override *GetInitialSet* to directly debug modules that are known to be buggy from prior experience or in-house testing. Similarly, an experienced programmer can look at already-monitored sites and specify her own plan based on domain knowledge, whether to test a hypothesis or simply chase down a hunch. The set of monitored sites can be actively modified to balance aggressive exploration against user-tolerable overhead.

5. EVALUATION

Prior work [1, 7, 17, 26–28, 45] has shown qualitatively and quantitatively that statistical debugging is effective. In this section, we evaluate the main contribution of this paper, which is the use of adaptive binary instrumentation to further reduce performance overheads. Our evaluation uses the faulty programs of the Siemens suite [16]; the bash, flex, grep, gzip and space bug benchmarks [40]; and gcc 2.95.3 [11]. We also evaluated bc 1.06, ccrypt 1.2 and exif 0.6.9, each of which has known fatal bugs [27]. Some test subjects have multiple variants, each exhibiting a different bug. Table 1 lists our test programs, the number of variants, size in lines of code and number of instrumentation sites, and the size of the test suite used. All test programs except Siemens programs are realistic applications with several thousand lines of code (KLOC). bash and gcc are the largest, with greater than 50 KLOC.

Bugs in some test subjects cause incorrect output rather than crashes. A test case is labeled as a success or failure by comparing the output of the buggy program to that of a bug-free reference version. Our statistical methods are applicable irrespective of the labeling strategy used. For the purposes of backward analysis, the failure point is defined as the statement in the program that prints the first incorrect byte in the output. We find this location by tracing program output and associating each output byte with the code that printed it [15]. Since this output tracer does not support bash, we do not perform backward analysis for bash.

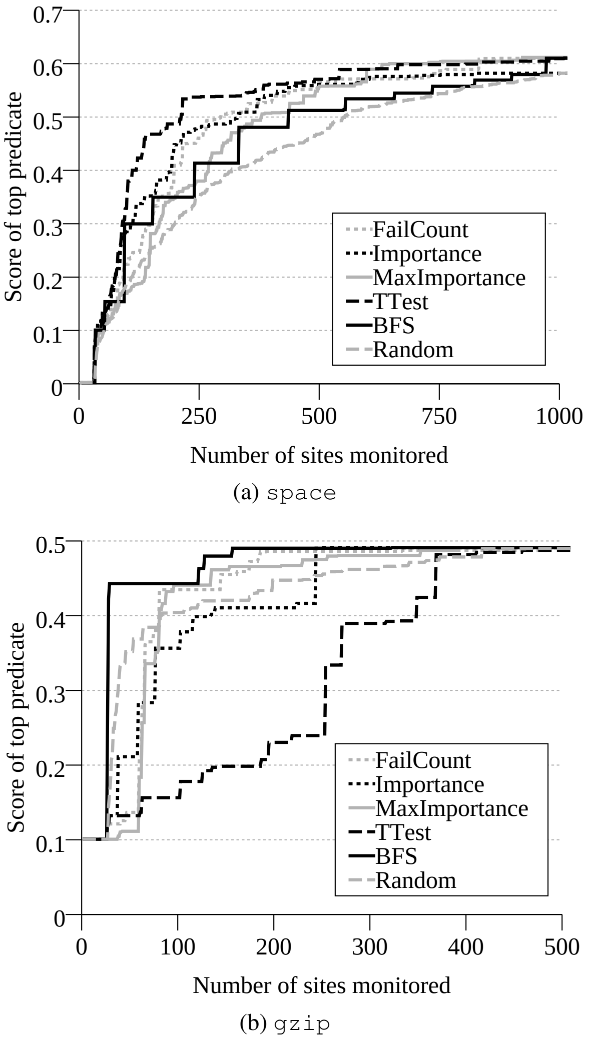
The *Importance* score is undefined if a variant fails in zero or one test case. Such variants are discarded and not included in Table 1. For the remainder, we run the adaptive analysis given in Procedure 1. To mimic a real deployment in which no two runs are exactly alike, we partition tests into random subsets of 500 cases each and cycle through these for successive iterations. When using the *Random* heuristic, all measures are averages across five trials.

5.1 Comparison of Heuristics

The intent of each heuristic is to guide adaptive analysis toward high-scoring predicates. To evaluate effectiveness, Figure 2 shows the score of the top-scoring predicate found versus the total number of sites monitored so far. Note that *Vicinity*(*best*) in Procedure 1 may return multiple sites. When this occurs, flat horizontal segments appear in the plots of Figure 2, reflecting a larger-than-unit jump in the number of sites monitored. This is particularly common with *BFS*, which can fan out quickly in each iteration.

In the plot for space in Figure 2a, all heuristics start by finding essentially the same top score before diverging after 100 sites. The divergence at 100 sites may look small in the plot, but it is significant considering the large range covered by the *x* axis. *TTest* performs best followed by *Importance*. *FailCount* lags *Importance* in the early phases because it cannot distinguish predicates seen only in failing runs from those seen always. The pathological cases for *Importance* that motivated the design of *MaxImportance* appear to be uncommon in space as indicated by the fairly poor performance of *MaxImportance*. Wide fan-out gives *BFS* a stepped profile and leads it to instrument many uninteresting sites on its way to a good predicate. All of these outperform the *Random* straw man. The heuristics behave similarly on all other applications except gzip and flex: *BFS* and *Random* are significantly worse than the other heuristics; *TTest* is the best or close to the best; the others are close to *TTest* for some cases and close to *BFS* in the rest. For the large applications (bash and gcc), *FailCount* is the best heuristic but only slightly head of *TTest*. We omit the curves for these applications in interest of space. However, the plot for gzip in Figure 2b deviates from this pattern, with *BFS* and other heuristics significantly outperforming *TTest*. Manual inspection shows that the top predictors are near the top of the CDG (usually two or three levels deep). Thus, *BFS* and *Random* are better at finding them early, while *TTest* gets sidetracked by an initial wrong choice. These are ultimately heuristics, and therefore can occasionally perform sub-optimally. flex exhibits behavior similar to gzip.

Figure 3 plots the score of the top-scoring predicate found versus the total number of sites monitored so far, for space, using the



(a) space

(b) gzip

Figure 2: Adaptation speed for various heuristics using forward analysis

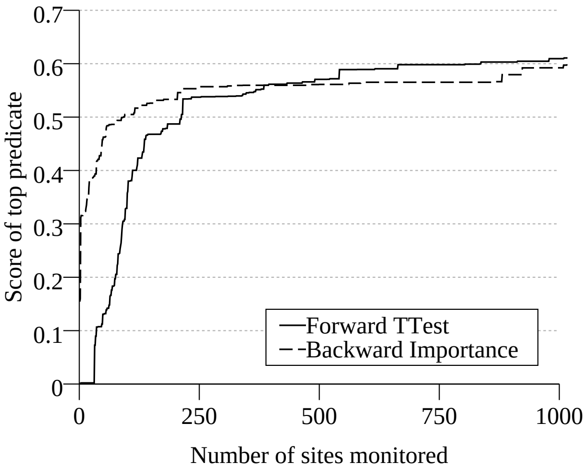


Figure 3: Adaptation speed for `space` using backward analysis with *Importance* and forward analysis with *TTest*

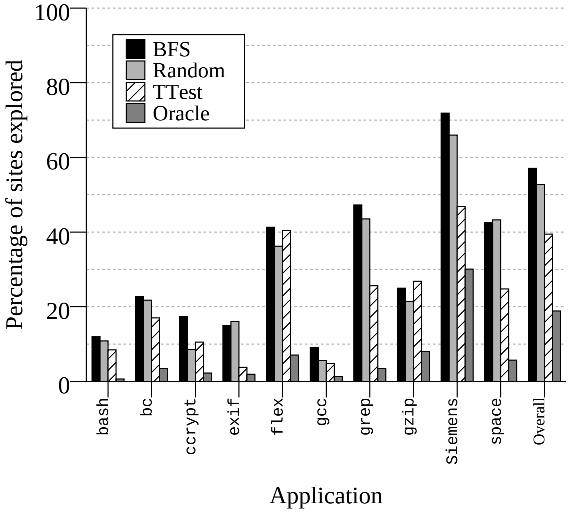


Figure 4: Mean number of sites to find top-ranked predictor. The rightmost set of bars averages across all programs.

Importance heuristic with backward analysis and using the *TTest* heuristic with forward analysis. (Backward *TTest* is meaningless.) Backward analysis begins finding good predictors much earlier than forward analysis. After about 200 sites, exploration no longer significantly improves scores. This affirms folk wisdom that many bugs are close to their points of failure. The rise in the backwards curve near 900 sites suggests that the principle of locality does not always apply; high-scoring predicates occasionally appear in the same locality as low-scoring predicates. Backward analysis has the same behavior in other applications, so we omit their plots in interest of space. For bc and exif, which have crashing bugs, the top predictor is very close to the point of failure and is found rapidly by backward analysis, after exploring fewer than 100 sites.

5.2 Instrumentation Selectivity

Section 1 noted that prior approaches discard over 99.996% of instrumented predicates. To gauge our improvement against this baseline, we measure the total instrumentation effort required for the adaptive process to discover the same top-ranked bug predictor as a traditional, non-adaptive method. We choose the analysis of Liblit et al. [27] as our non-adaptive reference. For each variant, we note the top bug predictor as identified by Liblit et al., then count sites explored before an adaptive analysis identifies this same predicate. We compare *TTest* with *BFS* and *Random* to test whether a carefully-selected heuristic can help in this task. We also consider an *Oracle* heuristic that has perfect knowledge of program behavior. It always selects the branch predicate that reaches the target predicate by monitoring the fewest instrumentation sites.

Figure 4 plots the percentage of sites explored for each program averaged across all variants. *BFS* and *Random* explore about 60% of sites before finding the top predicate. *TTest* explores just 40%. *Oracle* suggests that there is room for improvement but also establishes a lower bound of about 25% for any adaptive search that crosses CDG edges one at a time. Adaptive analysis performs very well in bash, bc, ccrypt, exif, and gcc, finding the top predictor while instrumenting less than 20% of sites on average.

Table 2 shows the mean number of sites instrumented during an iteration and the mean number of iterations required to find the top bug predictor. *BFS*'s wide fan-out reveals the top predictor in fewer iterations but instruments many sites. Other forward heuristics

Table 2: Mean number of sites instrumented per iteration and mean number of iterations

	Mean number of iterations to find top-ranked predictor							Mean number of sites instrumented per iteration						
Program	FailCount	Importance	MaxImp	TTest	Random	BFS	BwImp	FailCount	Importance	MaxImp	TTest	Random	BFS	BwImp
`bash`	181.0	314.0	316.0	545.0	766.4	12.0	-	3.9	3.4	3.2	2.8	2.6	179.2	-
`bc`	94.0	156.0	73.0	66.0	145.2	7.0	1.0	3.6	2.9	4.3	4.8	2.8	60.9	163.0
`ccrypt`	11.0	79.0	9.0	29.0	16.8	5.0	59.0	3.3	2.4	5.3	2.6	3.6	24.8	4.5
`exif`	73.0	99.0	70.0	33.0	186.6	16.0	22.0	1.9	2.4	1.9	3.0	2.3	24.6	2.7
`flex`	176.3	242.5	67.6	311.4	240.8	3.1	114.7	46.7	45.4	51.7	44.1	43.8	289.6	137.5
`gcc`	618.0	1227.0	662.0	645.0	863.3	10.0	1500.0	3.8	3.2	3.7	4.2	3.8	517.2	12.0
`grep`	180.2	388.2	107.5	299.2	546.3	10.6	190.2	5.8	6.1	6.3	5.9	5.1	110.4	9.7
`gzip`	57.7	42.6	50.8	127.8	84.3	3.1	87.6	5.5	10.1	6.5	9.8	6.5	105.3	10.6
Siemens	19.7	25.6	20.7	18.6	30.6	7.7	21.2	3.2	3.1	3.1	3.2	2.9	9.9	3.5
`space`	148.9	158.6	161.6	134.9	266.5	9.8	76.2	3.3	3.6	3.1	3.9	2.6	56.1	18.3

Table 3: Relative performance overheads

	Sampling				Adaptive	
Program	1/1	1/100	1/∞	Binary	TTest	BFS
`bash`	1.320	1.259	1.138	1.868	1.129	1.161
`bc`	1.174	1.149	1.130	1.306	1.006	1.050
`gcc`	3.689	2.438	1.655	7.290	1.001	1.037
`gzip`	3.587	2.017	1.570	3.464	1.023	1.110
`exif`	2.045	2.002	1.315	14.035	1.164	1.227
Overall	2.363	1.773	1.362	5.592	1.065	1.117

instrument roughly the same number of sites per iteration but differ in the number of iterations required. `flex` has a relatively flat CDG due to a large number of *switch* statements in its input scanner. This causes significantly many sites to be instrumented per iteration. The best predictor for `bc` is very close to the point of failure and hence backward analysis completes quickly. The number of iterations required is in the order of tens for Siemens programs and in the order of hundreds for large applications, which is not large for wide deployments that generate many feedback reports.

5.3 Performance Impact

Table 2 shows that very few sites are instrumented at any time. We might then expect lower overheads. We test this hypothesis by measuring the mean overhead for executing the monitoring plans suggested by the *TTest* and *BFS* heuristics. Table 3 shows measured overheads relative to 1 for non-instrumented code. We evaluate performance for `bash`, `bc` and `exif`, whose non-instrumented programs run for approximately 0.25, 4, and 0.01 seconds respectively. The test suites for other programs, being functionality tests rather than performance tests, execute for extremely short periods (order of a few milliseconds) and their performance cannot be reliably measured. For `gcc` and `gzip`, we evaluate performance using inputs in the SPEC benchmark suite. Non-instrumented programs take between 0.5 and 50 seconds on these inputs. We compare our technique against complete binary instrumentation and the sampling scheme of Liblit et al. [27]. We experiment with sampling rates of 1/∞ (the best case for sampling based instrumentation), 1/100 (suggested by Liblit et al. [26] for public deployments) and 1/1.

Adaptive instrumentation is at least an order of magnitude faster than complete binary instrumentation and significantly faster than all sampling variants. *BFS*, due to its wide fan-out, has a higher overhead than *TTest*. The overhead is minuscule for `bc`, `gcc` and `gzip`, where long running times amortize instrumentation costs. Overheads for `bash` and `exif` are easily affected by measurement noise as even a 0.01 second offset could change the overhead by at least 4%. If these short-lived programs are excluded, the average overhead for the remaining programs is 300% for complete binary instrumentation, 87% for sampling at a rate of 1/100 and 1% for adaptive instrumentation. An overhead of 1% is effectively imperceptible to an end-user. Selective instrumentation, if applied by a static instrumentor, can achieve order-of-magnitude improvements over sampling. But enacting new plans would require distributing executable patches: an impractically resource-intensive proposition.

5.4 Multiple Bugs

Confronted with multiple bugs, adaptive analysis might focus exclusively on the most frequent bug, or it might split its attention between several different, equally prevalent bugs. Neither is desirable. Infrequent bugs are ignored in the first case, while it takes longer to find good predictors for any of the bugs in the second case.

We explore this issue using `exif`, which contains three known bugs [27]. One bug was exhibited only once in our test suite of 10,000 runs and hence is ignored. The other two caused 228 and 180 runs to fail. Our technique finds the top bug predictor for the first bug in about 32 iterations, after instrumenting just 100 of 2,631 predicates. The best predictor for the second bug is not found until iteration 423, after instrumenting 817 predicates. Manual inspection shows that many sites instrumented between iterations 32 and 423 relate to the first bug. This affirms that adaptive instrumentation can stall in the presence of multiple bugs.

We propose a solution similar in spirit to that of Liblit et al. [27]: failures due to other bugs can be thrown away while pursuing a fix for a particular bug. Multiple instances of Procedure 1, each pursuing one bug, can be active simultaneously. Deployed programs can be randomly split to collect feedback data for these multiple instances. This requires grouping failing runs by cause. For crashes, one may use crash stacks or just the crashing program counter to label failures. We label each failing run for `exif` based on the failing PC and run two separate instances of Procedure 1. The best bug predictor for the first bug is found in 41 iterations and 120 instrumentation sites. The second bug is caught after 26 iterations and 68 sites. Thus, while Procedure 1 by itself is not designed for multiple bugs, it can be easily modified to handle them.

5.5 Comparison with Holmes

As noted earlier, HOLMES [7] introduced two orthogonal concepts: path-based instrumentation and adaptive predicate selection. We focus our comparison only on the later. We do not consider path predicates in this section for two reasons. First, our binary and source instrumentors do not yet support the selective path profiling used to efficiently implement path predicates. Second, HOLMES's use of path predicates is orthogonal to adaptivity, and could be added to our system in the same manner.

In effect, HOLMES defines *Vicinity* at the granularity of entire functions. HOLMES finds *weak predictors* instrumented in earlier iterations, defined as predicates with *Importance* scores between 0.5

and 0.75. It selects functions close to these predictors in the PDG, and instruments all predicates in these neighboring functions in the next iteration.

Weak predictors can be quite sparse. Because HOLMES explores only near weak predictors, this creates a risk that it can get stuck with no new sites available to explore. In our experiments, this was the case in 46 of the 130 Siemens experiments and 61 of the 97 larger experiments. In 111 of the 120 experiments in which HOLMES finds the top predictor, the sparsity of weak predictors is side-stepped because the top predictor is so close to the point of failure that it is instrumented in the very first iteration. While the definition of weak predictors seems to be the impediment here, if we remove that restriction and define *Vicinity* to explore near all predictors, then HOLMES reduces to doing a breadth-first search on the call graph, which is a coarser version of the *BFS* heuristic evaluated earlier. Our approach cannot get stuck, as noted in Section 4.4.

Exploring at the granularity of functions is even coarser than *BFS*. Hence, more sites will be instrumented per iteration, imposing more overhead, but requiring fewer iterations. Our approach is more flexible, allowing trade-offs between overhead and the number of iterations. We do not present a direct performance comparison because HOLMES performs only trivial (single-iteration) explorations of all programs in Table 3.

6. RELATED WORK

Renieris and Reiss [37] present a model for debugging where the programmer does a breadth-first search on the program-dependence graph. This model has been used in subsequent studies [8, 17, 18, 28, 45] to quantify the effectiveness of bug predictors. In this paper, we adopt a similar model where we use a heuristic search instead of breadth-first search. It should be noted that in earlier papers, the model was used as an independent metric to compare different analyses. Here, we use it for a different goal: to reduce the monitoring overhead. Also, whereas prior search models operate on program-dependence graphs, we search across control-dependence edges only. This is due to the difficulty of extracting precise data-dependence edges from binary code, although work by Balakrishnan et al. [3] may permit adaptive PDG exploration in the future.

In their execution classification tool, Haran et al. [12] select program behaviors to be monitored using weighted sampling. Like CBI, sampling is used to reduce monitoring overhead. In the presence of a large user community, weighted sampling can be combined with our technique for a non-uniform assignment of instrumentation sites to users using any of our heuristics as weights.

Several other dynamic program analysis tools alter their behavior adaptively. The dynamic leak detector of Hauswirth and Chilimbi [13] profiles code segments at a rate inversely proportional to their execution frequencies. Yu et al. [44] use dynamic feedback to control the granularity of locksets and threadsets in their data race detection algorithm. Dwyer et al. [9] make adaptive, online decisions to monitor just a subset of the program events in their dynamic finite-state property verifier. The AjaxScope platform for monitoring client side execution of web applications [20] provides mechanisms for specifying adaptive policies; the authors describe a performance profiling tool using this feature.

The Paradyn project [30] uses adaptive, dynamic instrumentation for performance profiling of large parallel programs. Roth and Miller [39] emphasize automated, on-line diagnosis of performance bottlenecks. Unlike Paradyn and the other online adaptive analyses [9, 13, 20, 44] our approach uses statistical bug detection with data being aggregated across many runs and analysis being performed offline. Paradyn's tools and techniques for managing and visualizing large data streams may be useful in our domain as well.

The GAMMA project represents one of the first practical systems for run-time monitoring of deployed software. Orso et al. [33] describe a data collection infrastructure, termed *software tomography*, that supports a variety of software evolution tasks and which allows post-deployment changes to data collection. However, while GAMMA automates the distribution of data-collection tasks among a user community, selection of those tasks is assumed to be human-directed. We propose an automatic, heuristically-guided system for bug-hunting that changes data-collection tasks in response to feedback. As with Paradyn, visualization techniques developed by the GAMMA group [19, 34] may prove useful to help programmers understand and interpret data collected using our adaptive approach.

7. CONCLUSIONS

Post-deployment bug hunting is a search for a needle in a haystack. Monitoring strategies that cannot respond to feedback incur large overheads and waste considerable computational resources. We use statistical analysis, static program structure, and binary instrumentation to develop an adaptive post-deployment monitoring system. Of several search heuristics considered, one (*TTest*) consistently performs well in the forward direction while another (*Importance*) shows promise when working backward from known points of failure. We find that this technique achieves the same results as an existing statistical method while monitoring, on average, just 40% of potential instrumentation sites in the programs we considered. Performance measurements show that our technique imposes an average performance overhead of 1% for a class of large applications as opposed to 87% for realistic sampling-based instrumentation. Monitoring overheads are so small as to be nearly immeasurable, making our adaptive approach practical for wide deployment.

8. REFERENCES

[1] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In J. N. Kok, J. Koronacki, R. L. de Mántaras, S. Matwin, D. Mladenic, and A. Skowron, editors, *ECML*, volume 4701 of *Lecture Notes in Computer Science*, pages 6–17. Springer, 2007.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, pages 168–179, 2001.

[3] G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum. CodeSurfer/x86–a platform for analyzing x86 executables. In R. Bodík, editor, *CC*, volume 3443 of *Lecture Notes in Computer Science*, pages 250–254. Springer, 2005.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, pages 46–57, 1996.

[5] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[6] K.-R. Chilakamarri and S. G. Elbaum. Reducing coverage collection overhead with disposable instrumentation. In *ISSRE*, pages 233–244. IEEE Computer Society, 2004.

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

[8] H. Cleve and A. Zeller. Locating causes of program failures. In Roman et al. [38], pages 342–351.

[9] M. B. Dwyer, A. Kinneer, and S. G. Elbaum. Adaptive online program analysis. In *ICSE*, pages 220–229. IEEE Computer Society, 2007.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[11] Free Software Foundation. GCC: The GNU compiler collection. http://gcc.gnu.org/.

[12] M. Haran, A. F. Karr, M. Last, A. Orso, A. A. Porter, A. P. Sanil, and S. Fouche. Techniques for classifying executions of deployed software to support software engineering tasks. *IEEE Trans. Software Eng.*, 33(5):287–304, 2007.

[13] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In S. Mukherjee and K. S. McKinley, editors, *ASPLOS*, pages 156–164. ACM, 2004.

[14] S. Horwitz and T. W. Reps. The use of program dependence graphs in software engineering. In *ICSE*, pages 392–411, 1992.

[15] S. Horwitz, B. Liblit, and M. Polishchuk. Better debugging via output tracing and callstack-sensitive slicing. *IEEE Transactions on Software Engineering*, 36(1):7–19, Jan./Feb. 2010.

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

[17] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE*, pages 184–193. ACM, 2007.

[18] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *ASE*, pages 273–282. ACM, 2005.

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

[20] E. Kiciman and V. B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In T. C. Bressoud and M. F. Kaashoek, editors, *SOSP*, pages 17–30. ACM, 2007.

[21] D. A. Levine. Algorithm 344: Student's t-distribution [s14]. *Commun. ACM*, 12(1):37–38, 1969.

[22] Z. Liang and R. Sekar. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In *ACSAC*, pages 215–224. IEEE Computer Society, 2005.

[23] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In V. Atluri, C. Meadows, and A. Juels, editors, *ACM Conference on Computer and Communications Security*, pages 213–222. ACM, 2005.

[24] B. Liblit. *Cooperative Bug Isolation (Winning Thesis of the 2005 ACM Doctoral Dissertation Competition)*, volume 4440 of *Lecture Notes in Computer Science*. Springer, 2007.

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

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

[27] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 15–26. ACM, 2005.

[28] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 286–295. ACM, 2005.

[29] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[30] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11): 37–46, 1995.

[31] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In Roman et al. [38], pages 156–165.

[32] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*. The Internet Society, 2005.

[33] A. Orso, D. Liang, M. J. Harrold, and R. J. Lipton. Gamma system: continuous evolution of software after deployment. In *ISSTA*, pages 65–69, 2002.

[34] A. Orso, J. A. Jones, M. J. Harrold, and J. T. Stasko. Gammatella: Visualization of program-execution data for deployed software. In *ICSE*, pages 699–700. IEEE Computer Society, 2004.

[35] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies–a safe method to survive software failures. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[36] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *OSDI*, pages 61–74. USENIX Association, 2006.

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

[38] G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors. *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, 2005. ACM.

[39] P. C. Roth and B. P. Miller. On-line automated performance diagnosis on thousands of processes. In J. Torrellas and S. Chatterjee, editors, *PPOPP*, pages 69–80. ACM, 2006.

[40] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. Software-artifact intrastructure repository. http://sir.unl.edu/portal/, Sept. 2006.

[41] M. M. Tikir and J. K. Hollingsworth. Efficient online computation of statement coverage. *Journal of Systems and Software*, 78(2):146–165, 2005.

[42] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In R. Yavatkar, E. W. Zegura, and J. Rexford, editors, *SIGCOMM*, pages 193–204. ACM, 2004.

[43] H. M. G. H. Wassel. An enhanced bi-clustering algorithm for automatic multiple software bug isolation. Master's thesis, Alexandria University, Egypt, Sept. 2007.

[44] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In A. Herbert and K. P. Birman, editors, *SOSP*, pages 221–234. ACM, 2005.

[45] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In W. W. Cohen and A. Moore, editors, *ICML*, volume 148 of *ACM International Conference Proceeding Series*, pages 1105–1112. ACM, 2006.