HOLMES: Effective Statistical Debugging via Efficient Path Profiling

Trishul M. Chilimbi
Microsoft Research Redmond
trishulc@microsoft.com

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

Krishna Mehra
Microsoft Research India
v-kmehra@microsoft.com

Aditya V. Nori
Microsoft Research India
adityan@microsoft.com

Kapil Vaswani
Microsoft Research India
kapilv@microsoft.com

Abstract

Statistical debugging aims to automate the process of isolating bugs by profiling several runs of the program and using statistical analysis to pinpoint the likely causes of failure. In this paper, we investigate the impact of using richer program profiles such as path profiles on the effectiveness of bug isolation. We describe a statistical debugging tool called HOLMES *that isolates bugs by finding paths that correlate with failure. We also present an adaptive version of* HOLMES *that uses iterative, bug-directed profiling to lower execution time and space overheads. We evaluate* HOLMES *using programs from the SIR benchmark suite and some large, real-world applications. Our results indicate that path profiles can help isolate bugs more precisely by providing more information about the context in which bugs occur. Moreover, bug-directed profiling can efficiently isolate bugs with low overheads, providing a scalable and accurate alternative to sparse random sampling.*

1 Introduction

Commercial software ships with undetected bugs despite the combined best efforts of programmers, sophisticated bug-detection tools and extensive testing. These software errors that go undetected can cause crashes that are disruptive at best and cost money and lives at worst. To fix the problem, a programmer often has to painstakingly try to work backwards from the crash to isolate and debug the root cause of the error. This process is tedious, error-prone and involves guesswork as typically the only information available is the stack trace and register values at the point where the program crashed.

Statistical debugging pioneered by the Cooperative Bug Isolation project (CBI) [21] aims to streamline and automate the process of isolating the bug responsible for the program failure. It achieves this goal by collecting information about program execution via *predicate profiles* (these are values of certain types of predicates at particular program points) from both successful and failing runs of a program and applies statistical techniques to pinpoint the likely cause of the software failure.

Prior work on statistical debugging has not fully investigated the impact of using richer profiles from user executions, such as *path profiles* [11], on the accuracy of bug isolation. Paths are a natural candidate for debugging as they capture more information about program execution behavior than point profiles such as predicate profiles. Indeed, recent research on the effectiveness of compound Boolean predicates and path information derived from simple atomic predicates for predicting program failures [17, 10] suggests that using path profiles for isolating program failures has merit. In addition to more precise bug isolation, paths offer two other advantages over predicates. While predicates can pinpoint where the bug occurs in the code, paths can additionally provide more context on how the buggy code was exercised which can aid and simplify the task of debugging. Moreover, path profiles are routinely collected in some systems for profile-guided performance optimization – in such cases, statistical analysis can be performed at no additional program instrumentation or change to the software build process. While this may seem like a minor point, large software is often compiled and tested with a brittle and complex build process that programmers are loath to change.

Figure 1 illustrates a code fragment derived from the `printtokens2` benchmark of the SIR benchmark suite [5] that illustrates the potential advantage of path profiles over predicate profiles. In this case, the root cause of the bug is a missing return statement on line 26. Statistical debugging using program paths localizes the bug to a path shown in blue and underlined (lines 14, 17, 19, 22-24, 27, 29, 31, 32, 34-36). A cursory examination of the

```
1 token get_token(tp)
2 token_stream tp;
3 {
4   int i=0,j;
5   int id=0;
6   char ch,ch1[2];
7   ch=get_char(tp);
8   ...
9   /* prepare for string */
10  if(ch =='"') id=1;
11  ch=get_char(tp);
12
13  /* until meet the end character */
14  while (is_token_end(id,ch) == FALSE) {
15    buffer[i++]=ch;
16    ch=get_char(tp);
17  }
18
19  ch1[0]=ch;
20
21  /* if end is eof token, put back eof */
22  if(is_eof_token(ch1)==TRUE) {
23    ch=unget_char(ch,tp);
24    if(ch==EOF)unget_error(tp);
25    /* BUG - missing return */
26    /* return(buffer); */
27  }
28
29  if(is_spec_symbol(ch1)==TRUE) {
30    ...
31  }
32
33  /* if end is ", hold second " in buffer */
34  if(id==1) {
35    buffer[i++]=ch;
36    return(buffer);
37  }
38  ...
39  return(buffer);
40 }
```

Figure 1. Code fragment from the printtokens2 **benchmark that has a code missing bug at line 26.**

branches along the path reveals that the variable `ch`, which contains the character at the end of the token, must be an EOF marker (from the branch at line 22). Also note that along this path, the condition `id==1` at line 34 evaluates to true, which implies that the token is a string delimited by `"` (i.e. `ch == "`) – this is a contradiction. One possible explanation for this contradiction is that perhaps this path should have been infeasible and the program was never intended to reach line 35 with `ch == EOF`. This reasoning directly leads to the actual root cause. Conventional statistical debugging using atomic predicates localizes this bug to the branch at line 34. While this branch is "close" to the root cause, the branch alone does not provide enough context for the programmer to find the root cause. More generally, statistical debugging using atomic predicates cannot localize subtle, context-sensitive bugs that require reasoning about multiple correlated predicates. This motivates the necessity for richer profiles such as paths.

We describe the design and implementation of a statistical debugging tool called HOLMES that uses path profiles instead of predicate profiles to perform bug isolation. In the spirit of traditional statistical debugging, HOLMES first instruments the program in order to perform path profiling [11, 25]. Next, the program is run and path profiles from successful and failing program runs are used to identify and rank a subset of paths that are good predictors of the program failure. We show that HOLMES is more effective at isolating bugs as well as explaining them than predicate-based statistical debugging techniques.

Since statistical debugging relies on statistical techniques to isolate the cause of software failures, it benefits from profiles obtained by way of a large number of user executions. To encourage user adoption, the monitoring overhead needs to be sufficiently low and not impact the user experience while running the instrumented program version. Previously published statistical debugging techniques [20, 21, 10] addressed this by using a combination of sparse random sampling and collecting predicates counters that summarize how often an application predicate was true or false in the user execution. These studies have found that sampling has little effect on the accuracy of bug isolation given a large number of user executions and an appropriate statistical analysis.

While the runtime overhead of instrumentation is important in most cases, the space overhead of instrumentation may also be important in some cases. Sampling schemes based on code-duplication exacerbate this problem by roughly doubling the size of the executable [9, 16]. With such space overheads it is not practical to deliver executables instrumented for statistical debugging, for large applications, such as operating systems, which ship to many end users on fixed size media such as DVDs. The following quote from a distinguished Microsoft engineer highlights this issue: "We did the math of going to a second DVD for [Windows] Vista. Basically a second DVD doubles the costs, because you not only need two pieces of media, you also need a slightly more expensive case [23]".

HOLMES employs an alternative approach to sampling that addresses both time and space overheads through iterative and adaptive bug-directed profiling that relies on the observation that often, *only small portions of a program are relevant to a given bug*. In this approach, a program runs without any instrumentation until it starts generating bug reports due to failures. A desirable consequence of this approach is that bug-free programs do not incur any profiling overheads. Once HOLMES collects a sufficient number of bug reports, it combines these reports with information from static analysis of the program to identify portions of the program that are most likely to contain the root causes of bugs observed in the field. Within these portions of the code, it identifies a set of functions, branches and paths that should be profiled in subsequent runs. HOLMES instruments the program to profile these selected parts and re-deploys the

instrumented program in the field in order to collect profiles and bug reports from subsequent runs of the program. After a sufficient number of bug reports and profiles have been collected, HOLMES uses statistical analysis to identify paths that are strong predictors of the reported failures. Based on the score assigned to these predictors, HOLMES may either decide that the root causes of reported bugs have been found and report it to the developer, or expand the search by profiling other parts of the program starting from the reported weak predictors and continuing to collect more detailed profiles from subsequent runs of the re-instrumented program. HOLMES repeats this process of analyzing existing profiles and extending the search until the root cause is found.

The iterative, bug-directed profiling approach relies on a mechanism that can instrument and redeploy parts of the program in the field. Most existing operating systems already support this feature [6]. HOLMES is naturally suited for server-side applications, where parts of code can be dynamically updated in place.

In summary, this paper makes the following contributions.

- We describe the design and implementation of a statistical debugging tool called HOLMES that uses path profiles from successful and failing program runs to isolate the root cause of program errors and provide contextual information that can aid in debugging the error (Section 3.1).
- We also present an adaptive version of HOLMES that uses an iterative, bug-directed profiling to isolate the causes of program failures with extremely low-overheads. The low overheads permit the collection of *full* profiles, allowing HOLMES to isolate bugs from a much smaller number program runs. Furthermore, bug free programs do not incur any profiling overheads in this mode (Section 3.2).
- Finally, we evaluate the non-adaptive and adaptive versions of HOLMES using programs from the SIR benchmark suite [5] and some large real-world applications. Our results indicate that path profiles provide more precise bug isolation and iterative, adaptive profiling can effectively isolate bugs with low overhead without requiring sampling (Section 5).

2 Background

In this section, we give a short overview of statistical debugging and path profiling to facilitate the exposition of the ideas in this paper.

2.1 Statistical Debugging

Statistical debugging collects information about program execution from both successful and failing runs of a program and applies statistical techniques to pinpoint the likely cause of the software failure [21]. First, a program is instrumented to collect data about the values of certain types of predicates at particular program points. There are three categories of predicates tracked:

1. **Branches:** For every conditional, two predicates indicating whether the true or false branch was taken are tracked.

2. **Returns:** At each scalar-returning function call site, six predicates indicating whether the return value is $< 0, \leq 0, > 0, \geq 0, = 0$, or $\neq 0$ are tracked.

3. **Scalar-pairs:** At each scalar assignment $x = \ldots$, identify each same-typed in-scope variable y_i and each constant expression c_j . For each y_i and each c_j , six predicates on the new value of x: $<, \leq , > , \geq, =, \neq$ are tracked.

This information is then aggregated across multiple runs of a program in the form of feedback reports. The feedback report for a particular program execution is formed as a bit-vector, with two bits for each predicate (observed and true), and one final bit representing success or failure. In the next step, the predicates are assigned numeric scores to identify the best predictor from the set of predicates. For details on the scoring process, see Liblit et al [21]. It is assumed that this predictor corresponds to one important bug, though other bugs may remain. This top predictor is recorded, and then all feedback reports where it was true are removed from consideration under the assumption that fixing the corresponding bug will change the behavior of runs in which the predictor originally appeared. The scores are recalculated for all remaining predicates in the remaining sets of runs. The next best predictor among the remaining reports is then identified, recorded, and removed in the same manner. This iterative process terminates either when no undiagnosed failed runs remain, or when no more failure-predictor predicates can be found. This process of iterative elimination maps each predictor to a set of failing program runs. The output of the analysis is a list of predicates with the highest scores at the end of the elimination algorithm.

In order to make this approach tractable, the monitoring overhead needs to be low and not impact user experience while running the instrumented program version. Statistical debugging techniques [20, 21, 10] address this by way of sparse random sampling and establish that sampling has little effect on the accuracy of bug isolation given a large number of user executions and appropriate statistical analysis.

2.2 Path Profiling

Path profiles are succinct and pragmatic abstractions of a program's dynamic control-flow behavior. Program path histories often serve as a valuable debugging aid by revealing the instruction sequence executed in the lead up to interesting program points. The efficient path profiling scheme proposed by Ball and Larus [11] forms the basis of most path profilers. Note that the paths considered here are intra-procedural and acyclic path segments in the control flow graph of the program. *In the rest of this paper, we refer to these intra-procedural and acyclic path segments as paths*. Several compiler optimizations perform better when trade-offs are driven by accurate path profiles [8]. Program paths are also a more credible way of measuring coverage of a test suite [14]. In this paper we examine how paths can be used profitably for the purpose of statistical debugging in a cost effective way.

3 HOLMES: Statistical Debugging Using Path Profiling

In this section we describe HOLMES, a statistical debugging tool that uses path profiles. HOLMES comes in two flavors.

- **Non-adaptive debugging.** In this version, HOLMES performs statistical debugging as described in Liblit et al [21] using paths instead of custom predicates.
- **Adaptive debugging.** To reduce execution time and space overheads, HOLMES can also perform statistical debugging using iterative, bug directed path profiling.

We will now describe the ideas underlying each of these approaches.

3.1 Non-Adaptive Debugging

For non-adaptive debugging, HOLMES implements Liblit et al.'s scalable statistical debugging algorithm using path profiles in place of predicate profiles. The program is instrumented so that path profiles are collected during execution and this information is aggregated across multiple runs through feedback reports. Conceptually, the feedback report for a single program execution is a bit-vector with two bits for every path: one bit that indicates whether the path was *observed* and the other bit that indicates whether the path was executed in that run. The feedback report also contains one bit that represents the success or failure of the execution. A path is said to have been *observed* when the node that corresponds to the start of the path is visited but the path is not necessarily executed.

In the next step, paths are assigned numeric scores to identify the best predictor of failure from the set of paths. We adapt the scoring process detailed in Liblit et al [21] to work with paths as follows. Predictors are scored based on *sensitivity* (accounts for many failed runs) and *specificity* (does not mis-predict failure in a successful run). These scores are balanced using a numeric importance score computed as follows. The events corresponding to a path p from all the runs can be aggregated into four values: (1) $S_o(p)$: the number of successful runs in which the path p was observed (2) $F_o(p)$: the number of failed runs in which the path p was observed (3) $S_e(p)$: the number of successful runs in which the path p was executed, and (4) $F_e(p)$: the number of failed runs in which the path p was executed. Using these values, three scores of bug relevance are calculated.

$$\text{Sensitivity}(p) = \frac{\log |F_e(p)|}{\log |F|} \quad (1)$$

$$\text{Context}(p) = \frac{F_o(p)}{S_o(p) + F_o(p)} \quad (2)$$

$$\text{Increase}(p) = \frac{F_e(p)}{S_e(p) + F_e(p)} - \text{Context}(p) \quad (3)$$

where F is the total number of failing runs. The harmonic mean of Sensitivity and Increase identifies predicates that are both highly sensitive and highly specific [21]:

$$\text{Importance}(p) = \frac{2}{\frac{1}{\text{Sensitivity}(p)} + \frac{1}{\text{Increase}(p)}} \quad (4)$$

The *Importance* score is calculated for each path, and the top results are selected and presented to the programmer as potential root causes.

3.2 Adaptive Debugging

Low runtime overheads are critical when programs are deployed in production environments where even a marginal slowdown in execution time or increase in memory consumption is unacceptable. Previous approaches for statistical debugging use sparse random sampling to reduce the overheads of collecting predicate profiles. When random sampling is enabled, predicates are evaluated and recorded intermittently. To compensate for the potential loss in accuracy due to random sampling, statistical debugging relies on a large number of both failing and successful runs of the program to build a representative profile. In addition, the sampling scheme used by Liblit et al. relies on a code duplication-based instrumentation scheme that effectively doubles the size of the program executable. Such a large increase in code size may not be acceptable in some situations.

To address these concerns, we propose an iterative, bug-directed version of HOLMES that relies on the observation

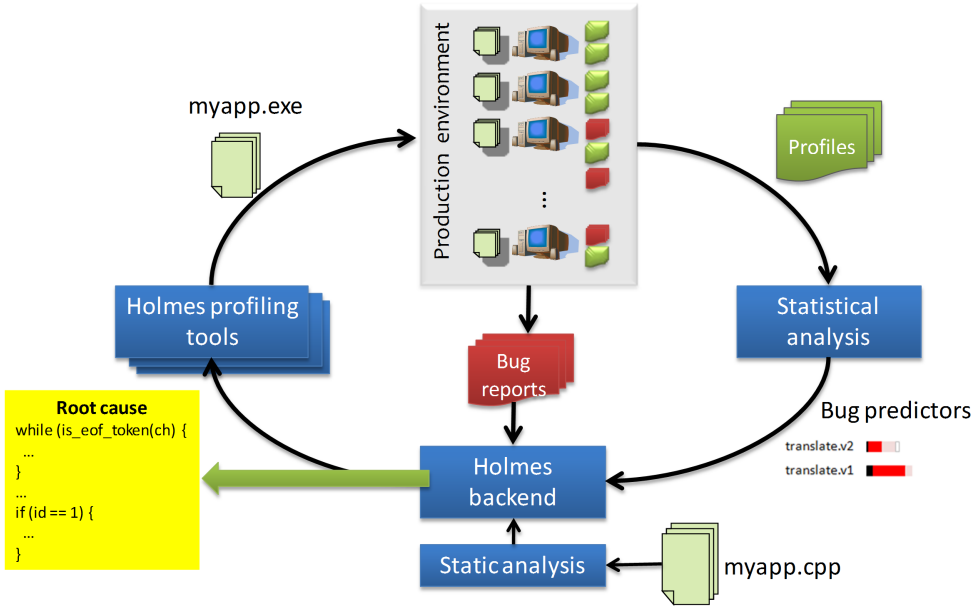

Figure 2. The HOLMES framework.

that in large programs, often only a small fraction of the code is relevant to a given bug. Figure 2 illustrates the HOLMES framework. The main components of this framework are: (a) bug report and profile collection, (b) static program analysis, (c) statistical analysis, and (d) selective instrumentation and deployment of instrumented code. We now discuss these components briefly and describe one implementation of this framework in the next section.

Initially, HOLMES monitors the uninstrumented program for failures and collects a set of bug reports, which contain a stack trace and partial state of the program at the point of failure. Once a sufficient number of bug reports have been collected, HOLMES's static analysis component uses these bug reports to identify parts of code that more likely to contain the root causes of the bugs. These portions of code are then instrumented to collect detailed profiles and redeployed in the field. Note that by focusing the instrumentation on portions of code relevant to the bug, HOLMES avoids the need for sparse random sampling.

Next, HOLMES analyzes the partial profiles using the statistical analysis described in Section 3.1. The analysis computes a statistical model of the program's outcome. The model consists of a set of bug predictors, each associated with the $Importance$ score. If the model identifies sufficiently strong bug predictors and explains all failures, HOLMES reports these predictors and terminates the iterative process. However, if the resulting model is inconclusive (perhaps because parts of code containing the root cause were not profiled), HOLMES expands its search by using static analysis and the model to identity other parts of code that closely *interact* with the weak predictors. This iterative process continues until strong predictors are obtained and all failures have been explained.

Often, software maintenance necessitates a scheme like HOLMES, especially when the bug occurs only on client machines and cannot be reproduced on test machines. In such scenarios, developers routinely selectively replace client binaries with instrumented versions in order to collect more information about the problem. However, in most cases, this is done manually and can be extremely tedious. HOLMES automates this iterative process of collecting and analyzing relevant profiles, letting developers focus on the more important task of fixing bugs.

4 HOLMES Implementation

We now describe an implementation of the HOLMES adaptive debugging framework that uses path profiles. Our implementation consists of two phases. In the *bootstrapping* phase, HOLMES is bootstrapped using bug reports from the field. This phase is followed by an *iterative* phase, which, in turn, consists of one or more data collection, statistical analysis, static analysis and redeployment phases. *Note that the non-adaptive version of* HOLMES *corresponds to a single instance of the iterative phase with full profiling.*

4.1 Bootstrapping Holmes

In order to bootstrap the process of finding the root cause, HOLMES exploits the common knowledge that often,

the root cause of bugs can be found in functions present on the stack trace at the point of failure. Based on this assumption, HOLMES processes bug reports to compute a set S of functions that appear in one or more stack traces. All functions in this set are candidates for instrumentation. However, to control the number of functions in this set, HOLMES computes an effective score $\Theta(f)$ for each function $f \in S$ that is defined as follows.

$$\Theta(f) = \sum_{i=1}^{n} \frac{1}{\Delta_i(f)} \quad (5)$$

where n is the number of bug reports and $\Delta_i(f)$ is the number of functions that lie between the function f and the function containing the location of failure in the stack trace of bug report i. Intuitively, $\Theta(f)$ is high for functions that appear often on the stack trace and/or appear close to the location of failure. HOLMES selects the top k functions from the set S based on this score for profiling in the next iteration) where k is a user-defined threshold that can be used to control the overheads of profiling). In case there are no stack traces, we bootstrap HOLMES using full branch profiles.

4.2 Iterative Profiling

We will now describe the details of each sub-phase in HOLMES's iterative profiling phase.

(a) Data Collection: In this phase, HOLMES deploys the instrumented version of the program and collects bug reports and profiles, if any, from all executions of this program. In contrast to CBI [21], HOLMES does not use temporal sampling and all instrumented functions are fully profiled. In spite of this, the profiling overheads are still low as the function selection phase ensures that only a small number of functions are instrumented.

(b) Statistical Analysis: After the bug reports and profiles are obtained, HOLMES uses its statistical analysis to analyze profiles for all runs since its previous iteration. The analysis returns a set of bug predictors, their scores, the set of failing runs each predictor explains and a set of unexplained failures. HOLMES classifies a predictor as *strong* if the predictor's score exceeds a user-defined threshold and *weak* otherwise. All strong predictors are marked as potential root causes and reported to the developer. If all failing runs are explained by strong predictors, HOLMES reinitializes the iterative process by deploying an uninstrumented version of the program, after which it needs to be bootstrapped again. On the other hand, weak predictors, if any, are passed to the function selection phase (described next) which uses them to identify the set of functions to be profiled in the next iteration.

(c) Function Selection: This phase takes as input a set of weak predictors and outputs a set of functions that should be considered for instrumentation in the next iteration. By definition, a weak predictor is one that is exercised in some failing runs but also exercised in some successful runs. There are two possibilities that HOLMES needs to be taken into account.

Interactions. Weak predictors often point to parts of code that may contain other stronger predictor. This is because a weak predictor may *interact* with other parts of the program via control and/or data dependencies, caller/callee relationships etc. Often, some of these interactions are more strongly correlated with failure than the predictor itself. We illustrate this scenario using an example that we encountered when analyzing a function from the `replace` benchmark.

Figure 4.1 shows a function `subline` in the `replace` benchmark (v3). This function contains a bug on line 9 due to a missing conditional. After bootstrapping, HOLMES identifies and profiles a function `patsize` (shown in figure) and finds that one of the paths through `patsize` is a weak bug predictor. An analysis of this function reveals that this weak path predictor interacts with the function `amatch` through a callee-caller relationship at line 51. Furthermore, one of the paths in `amatch` is a stronger bug predictor because it calls `patsize` in a restricted context (when `pat[j] == CLOSURE`). Similarly, we find that this path in `amatch` interacts with the function `subline` via a caller-callee relationship and the function `subline` contains an even stronger predictor, which also happens to be the root cause.

HOLMES uses static analysis to identify interactions or *coupling* [7, 13] between weak predictors and other parts of code. Since static analysis is conservative, it is possible that we may find a large fraction of the program interacting with the predictor and profiling all such interacting functions may be expensive. HOLMES addresses this problem by quantifying the degree of interaction between a predictor and a function using a custom code coupling measure, which simply counts the number of pairs of (data or control) dependent instructions between the predictor and any given function, including dependencies due to function arguments. HOLMES ranks functions based on the degree of coupling and again, selects the top k functions, where k is a user-defined threshold.

Strengthening weak predictors. Often, weak predictors can be *strengthened* by profiling the function containing the weak predictor using richer profiles. Consider a scenario in which a bug manifests occasionally when a specific branch in the program is traversed. However, the bug manifests almost always when one specific path through this branch

```
1 void subline(lin, pat, sub)
2 {
3   char  *lin, *pat, *sub;
4   int   i, lastm, m;
5   lastm = -1;
6   i = 0;
7   while ((lin[i] != ENDSTR))
8   {
9     m = amatch(lin, i, pat, 0);
10    /* BUG missing condition */
11    if ((m >= 0) /*&&(lastm!=m)*/){
12      putsub(lin, i, m, sub);
13      lastm = m;
14    }
15    if ((m == -1) || (m == i)) {
16      ...
17    }
18  }
19 }
```

(a)

```
20 int patsize(pat, n)
21 {
22   char* pat;
23   int n;
24   int size;
25   if (!in_pat_set(pat[n])) {
26     ...
27   }
28   else
29     switch (pat[n])
30     {
31       ...
32       case CLOSURE:
33         size = CLOSURE;
34       break ;
35           ...
36     }
37   return size;
38 }
```

(b)

```
39 int amatch(lin, offset, pat, j)
40 {
41   char* lin;
42   int offset;
43   char* pat;
44   int j;
45   int i, k;
46   bool   result, done;
47
48   done = false;
49   while ((!done) && (pat[j] != ENDSTR))
50     if ((pat[j] == CLOSURE)) {
51       j = j + patsize(pat, j);
52       i = offset;
53     while ((!done) && (lin[i] != ENDSTR)) {
54       result = omatch(lin, &i, pat, j);
55       if (!result)
56         ...
57     }
58       ...
59     }
60     ...
61 }
```

(c)

Figure 3. (a) Buggy version of the function `subline` in the `replace` benchmark. (b) Function `patsize` where Holmes finds a weak path predictor in the first iteration. (c) Function `amatch` where Holmes finds a weak path predictor in the second iteration.

is traversed. In such a scenario, if we used branch profiles and profiled the function containing the branch, HOLMES would have found a weak branch predictor. Instead, if we used path profiles, HOLMES would find a stronger path predictor. Similarly, if an atomic predicate appears as a weak predictor, a stronger predictor may be obtained if we used compound predicates [10]. Therefore, when a weak predictor is found in a function, HOLMES marks the function for more detailed profiling in the next iteration. Our current implementation of HOLMES only considers branches and paths for the purposes of profiling.

5 Experimental Evaluation

We conducted experiments to evaluate two aspects of our approach. First, we evaluate the effect of different profiling techniques on the accuracy of statistical debugging. In the second set of experiments, we evaluate the effectiveness of HOLMES in isolating bugs, both in terms of accuracy and runtime overheads.

Benchmarks and setup. We performed experiments using two sets of benchmark applications (Table 1). We picked 6 benchmarks from the Software Infrastructure Repository (SIR) [5]. These are the only SIR benchmarks we could compile using the Microsoft Phoenix compiler [4]. Each SIR benchmark contains several buggy versions and each of these versions contains a single bug. We also considered four large, real world applications, `gcc` [3], `apache` portable library (apr) [1], two versions of a binary translator being developed internally in Microsoft (`translate.v1` and `translate.v2`) and the EDG C++ compiler. We use the test suites provided with these applications to generate the set of successful and failing runs. We evaluate the adaptive profiling mechanism by re-running the test suite in each iteration. This methodology mirrors our assumption that a deployed application is likely to generate reasonably similar profiles until changes are made or the bugs are fixed. We performed our experiments on a system with the Intel Pentium Core 2 Duo CPU (1.6Ghz) and 3 GB of RAM running Windows Vista. While measuring overheads, we estimate the execution time of a benchmark by running the benchmark 5 times, ignoring the first run and computing the average of the other 4 runs.

Profiling tools. We used the Microsoft Phoenix compiler framework (July 2007 SDK) [4] to develop profiling tools that can instrument C/C++ programs to generate branch and path profiles. For predicate profiles, we rely on the instrumentation tool that accompanies CBI. This tool supports four different predicate instrumentation strategies namely scalar pairs, return variables, floating point operations and branches. We enabled all these strategies for our experiments. We were unable to evaluate the effectiveness of CBI over the 4 large applications because the current implementation of the instrumentation tool does not support these benchmarks.

Edges, Paths or Predicates? In Section 3, we hypothesized that bug isolation would benefit if path profiles are available. To test this hypothesis, we used branch and path profiles to drive HOLMES and predicate profiles (profiles of predicates from the family described in Section 2.1) to drive CBI. To ensure a fair comparison, we do not use sam-

Benchmark	Version	Description	LOC	# tests
printtokens	v1-v10	Lexical analyzer	726	4000
printtokens2	v1-v10	Lexical analyzer	570	4000
replace	v1-v10	Performs pattern matching and substitution	564	4000
space	v1-v10	An interpretor for an array definition language	6199	4000
schedule	v1-v10	Priority scheduler	412	2650
totinfo	v1-v10	Computes statistics for input data	565	1052
gcc	2.95.3	GNU C compiler	222196	812
apache	2.02	Web server	85661	269
translate	1	Binary to source translator	51987	133
translate	2	Binary to source translator	56604	260
EDG compiler	-	C++ compiler front end	466342	186

Table 1. Description of benchmarks used in our experiments.

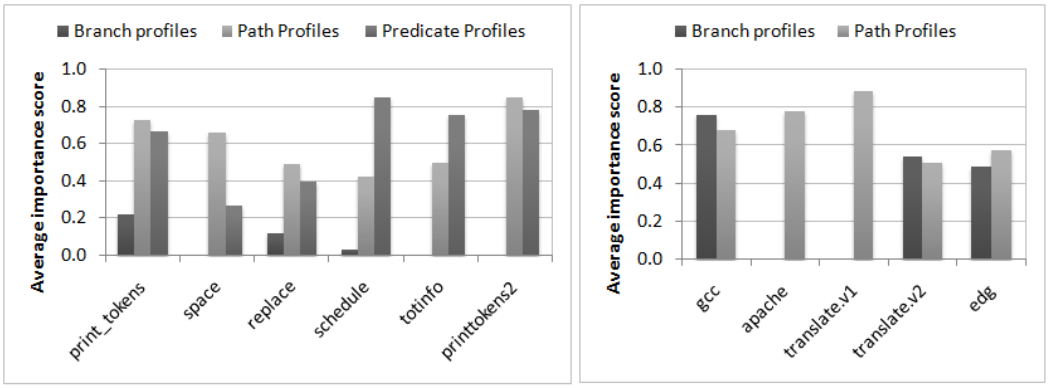

Figure 4. Average importance scores assigned to bug predictors for various profiling schemes. High scores are better.

pling for CBI. For each benchmark-profile pair, HOLMES and CBI output a set of bug predictors along with their importance scores (Section 3.1).

Figure 4 shows the importance scores for predictors obtained using branch, path and predicate profiles. These scores are averages across all versions of the same benchmark. The figure shows that predictors based on branch profiles have the lowest scores across most benchmarks. We investigated these scores further using bug thermometers [21] for both branch and path profiles (Figure 5). The thermometers show that for several benchmarks (like `apache` and `translate.v1`), the statistical analysis does not find any statistically significant bug predictors using branch profiles. On the other benchmarks, the top ranked bug predictors based on branch profiles either occur too often in successful runs (indicated by the long white portion at the end of the thermometer) or have low increase. On the other hand, the analysis using path profiles finds strong bug predictors (with high increase) for almost all benchmarks. The only exception is `gcc`, where branch profiles do as well as path profiles. These findings suggest that branches alone may not have sufficient discriminatory power to explain failures.

Perhaps one reason why branch profiles are not as effective is that large applications often go through rigorous test passes before they are released and it is not unusual for these tests passes to cover a reasonable fraction of the program's branches. Hence, if some of the branches were the root cause of bugs, such bugs are likely to be detected during internal testing.

The comparison between path and predicate based predictors is more interesting. On average, bug predictors obtained using path profiles outscore predictors obtained using predicate profiles in 4 of the 6 SIR benchmarks. However, the lack of a clear winner suggests that relying on one profiling scheme may not suffice and statistical debugging tools should adapt and use multiple profiling techniques.

Although the scores assigned by statistical debugging estimate the quality of bug predictors, the real measure of a predictor's effectiveness is the amount of programmer effort required to go from the predictor to the actual root cause. In general, quantifying programmer's effort to find the root cause starting with a given predictor is hard. However,

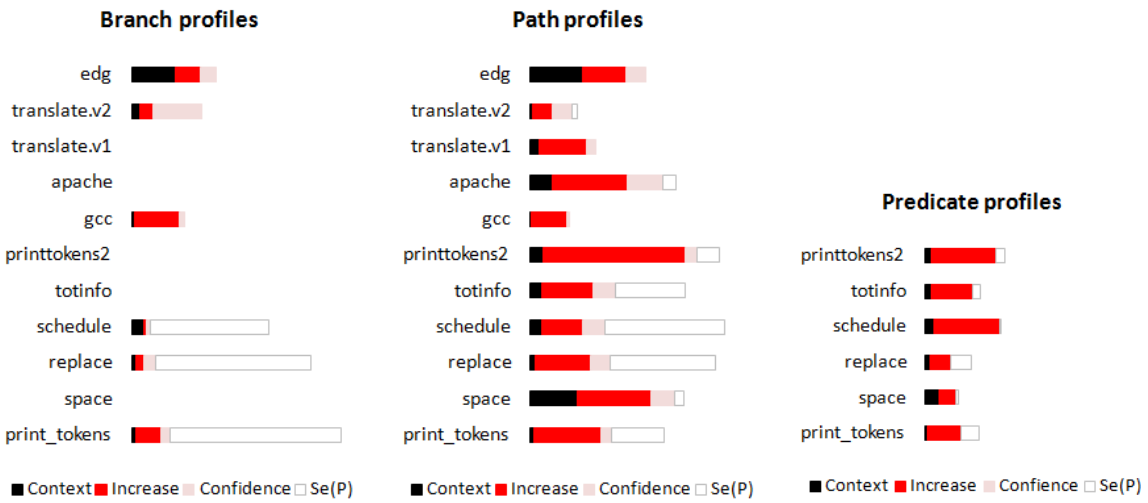

Figure 5. Bug thermometers used to visualize information about predictors. The figure shows the average context, the increase, the confidence in increase and the number of times the predictor is executed in successful runs for branch and path predictors.

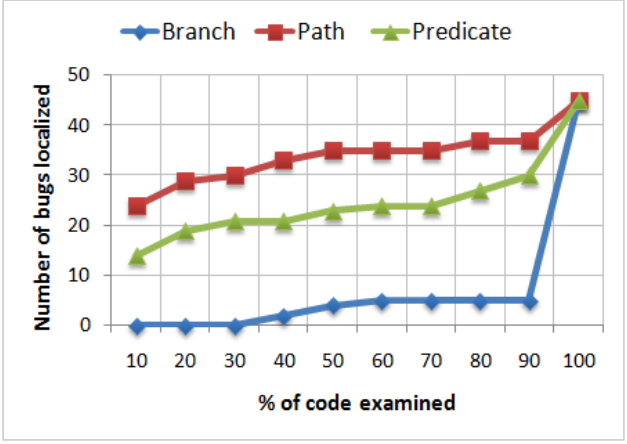

Figure 6. The distribution of the number of bugs localized vs. % code examined for three types of profiles.

researchers have proposed metrics that estimate programmer effort and can be measured automatically. Reineris et al. [24] propose a metric based on the program dependence graph that can be used to estimate a programmer's effort. This metric is defined as the number of nodes in the program's dependence graph traversed during a BFS search from one of the nodes corresponding to the predictor to a node that corresponds to the root cause. Based on this metric, we can compute for each predictor, the fraction of the program that a programmer would have to traverse to get to the root cause from the predictor. This metric T is defined as $T = \frac{N_{bfs}}{N_{pdg}} \times 100$, where N_{bfs} represents the distance from the root cause and N_{pdg} is the total number of nodes in the program dependence graph. We computed this metric using CODESURFER [2] and use this metric to evaluate the quality of predictors generated using three types of profiles. We conducted this experiment only for the SIR benchmarks since the actual root causes in the other benchmarks are not known.

Figure 6 illustrates the distribution of distances from the root cause vs. the fraction of bugs localized using each of the three profiling schemes. In this graph, a point (x, y) can be interpreted as"for x bugs, the programmer had to traverse less than or equal to y% of the program to find the root cause". As expected, branch profiles provide no value during the debugging process. Predictors obtained using path profiles are generally closer to the actual root cause than predictors obtained using branch or predicate profiles. For example, a programmer can root cause 24 of the 45 bugs using path-based predictors and 14 bugs using predicate-based predictors by examining $< 10\%$ of the code. On the whole, using paths as potential root causes tends to increases the accuracy of statistical debugging. This observation, coupled with the fact that paths can be profiled efficiently even in very large programs, suggests that path profiles are a natural profiling mechanism for scalable statistical debugging.

Adaptive debugging. As aforementioned, HOLMES's bug directed profiling technique iteratively instruments parts of

Benchmark	Path profiling	HOLMES$_1$	HOLMES$_2$	HOLMES$_3$
print_tokens	0.679 / 100.0%	**0.423 / 100.0%**	0.423 / 100.0%	0.423 / 100.0%
replace	0.570 / 98.9%	0.270 / 96.3%	**0.529 / 98.2%**	0.529 / 98.2%
gcc	0.679 / 66.6%	0.576 / 66.6%	**0.679 / 66.6%**	0.679 / 66.6%
translate.v1	0.534 / 58.3%	0.236 / 66.6%	0.471 / 25.0%	**0.534 / 58.3%**
translate.v2	0.825 / 93.3%	0.472 / 26.6%	**0.891 / 80.0%**	0.891 / 80.0%
edg	0.648 / 98.0%	0.651 / 97.0%	0.637 / 96.1%	**0.656 / 96.1%**

Table 2. Importance scores and percentage of failures explained for the first 3 iterations of HOLMES. Scores are averages across all versions in case of the SIR benchmarks. Bold text indicates the iteration where HOLMES reaches a fixed point.

	Benchmark	Branch profiling	Path profiling	HOLMES$_1$	HOLMES$_2$	HOLMES$_3$
Execution time overheads (%)	gcc	75.3	181.3	2.614	9.647	NA
	translate.v1	3.449	4.747	0.256	2.135	3.449
	translate.v2	8.770	2.787	0.842	0.008	0.386
	Benchmark	**Branch profiling**	**Path profiling**	HOLMES$_1$	HOLMES$_2$	HOLMES$_3$
Space overheads (%)	gcc	84.079	170.213	7.303	46.520	NA
	translate.v1	25.215	41.127	4.144	2.971	21.602
	translate.v2	25.099	43.232	3.147	1.833	0.790
	apache	35.620	43.440	3.010	NA	NA
	edg	168.063	374.686	240.398	251.440	247.016

Table 3. Execution time and space overheads for branch profiling, path profiling, and all HOLMES iterations.

code that are most likely to be relevant to a bug until HOLMES a sufficiently strong root cause is found. Our current implementation of HOLMES supports both branch and path profiling. HOLMES selects the type of profiling using the following policy. After bootstrapping, HOLMES uses selective path profiling in subsequent iterations. HOLMES terminates the iterative process if it finds a predictor with a score > 0.8. During each iteration, HOLMES ranks functions using the coupling measure and selects the top 5 for instrumentation. If the same predictors are seen in successive iterations, HOLMES switches to *full* branch profiles, identifies weak branch predictors and instruments functions that are coupled with these predictors using path profiling.

First, we measured the quality of predictors (represented by the importance score) obtained in successive iterations of HOLMES (Table 2). We find that in the first iteration, HOLMES generates weaker predictors that explain a small fraction of failures. In most cases, HOLMES is able to localize the bug with the same accuracy as path profiles in 2-3 iterations. HOLMES is unable to localize bugs in `apache` and `space` because the failing runs in these benchmarks do not generate stack traces.

Table 3 shows the runtime overheads for each iterations of HOLMES. We do not report execution time overheads for `apache` and the EDG compiler because the test suite has a large number of very short tests and the execution time of the test suite is dominated by the test harness. We find that the slowdown observed in each iteration of HOLMES is negligible because HOLMES only profiles a small fraction of code in each iteration, whereas full path profiling can potentially slow down the program by a factor of 2. We also find that HOLMES reduces the results code size significantly in most of the benchmarks.

6 Related Work

The problem of explaining software failures has generated considerable interest and research ideas in recent times. Various solutions to this problem have been proposed and these include static, dynamic and statistical analyzes of programs as well as their combinations. Ball, Naik and Rajamani [12] use a software model checker to generate error traces as well correct traces of programs and compute the differences in these traces to detect and localize defects in Windows device drivers. Delta-debugging of programs [15] identifies state differences between failing and passing runs of a program to detect causes in the program state that lead to a failure. This information is combined with "cause transitions" to isolate statements in the program that are potential causes for failure. Since this technique relies on collecting state information while the program runs, the process of isolating the root cause for a failure could potentially have

high runtime overheads. Jones et al. [18, 19] propose techniques that rank statements in a program based on their occurrence in failing and passing runs of the program. This ranking is subsequently used along with program visualization methods to aid the programmer in locating the statements that are root causes for failure in an application. In contrast, we show that paths are a more expressive and effective way of isolating program failures.

Our work is closely related to recent advances in statistical bug isolation implemented in the CBI project [20, 21] and [22]. CBI analyzes bug reports collected from deployed software in order to isolate root causes for failures in the software. Specifically, the program is instrumented to collect information about values of certain types of predicates at various program points and this information is passed on to a statistical engine in order to compute predicates that are highly correlated with failures. We use path profiles that (a) capture richer information about program executions than basic predicate profiles and, (b) can be computed at lower runtime and space overheads. Recent work on using compound Boolean predicates for predicting program failures in CBI [10] shows improvement in the effectiveness of bug isolation and indeed, paths are an instance of compound predicates. Jiang and Su [17] also layer a path learning algorithm on top of CBI in order to provide more context and information about the nature of a bug. In addition to showing the usefulness of paths in statistical debugging, we also show how iterative and adaptive bug-directed profiling provides a low overhead alternative to sampling.

7 Conclusions

We have implemented a statistical debugging tool called HOLMES that uses paths in place of predicates to perform bug isolation. We have also demonstrated an iterative and adaptive version of HOLMES that can effectively isolate bugs with low time and space overheads without resorting to sampling. Our results show that path profiles provide more precise and effective bug isolation as well simplify the task of debugging for real world applications.

References

[1] The Apache Software Foundation. http://www.apache.org.

[2] Codesurfer. http://www.grammatech.com/products/codesurfer/.

[3] GCC: The GNU Compiler Collection. http://gcc.gnu.org.

[4] Phoenix. http://research.microsoft.com/Phoenix.

[5] SIR: Software-artifact Infrastructure Repository. http://sir.unl.edu/portal/index.html.

[6] Windows update. http://windowsupdate.microsoft.com.

[7] E. B. Allen, T. M. Khoshgoftaar, and Y. Chen. Measuring coupling and cohesion of software modules: an information-theory approach. In *METRICS: Symposium on Software Metrics*, pages 124–134, 2001.

[8] G. Ammons and J. R. Larus. Improving data-flow analysis with path profile. In *PLDI: Programming Language Design and Implementation*, pages 72–84, 1998.

[9] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *PLDI: Programming Language Design and Implementation*, pages 168–179, 2001.

[10] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound Boolean predicates. In *ISSTA: International Symposium on Software Testing and Analysis*, pages 5–15, 2007.

[11] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO: International Symposium on Microarchitecture*, pages 46–57, 1996.

[12] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL: Principles of Programming Languages*, pages 97–105, 2003.

[13] A. Beszedes, T. Gergely, S. Farago, T. Gyimothy, and F. Fischer. The dynamic function coupling metric and its use in software evolution. In *Conference on Software Maintenance and Reengineering (CSMR)*, pages 103–112, 2007.

[14] T. M. Chilimbi, A. V. Nori, and K. Vaswani. Quantifying the effectiveness of testing via efficient residual path profiling. In *FSE: Foundations of Software Engineering*, pages 545–548, 2007.

[15] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE: International Conference on Software Engineering*, pages 342–351, 2005.

[16] M. Hirzel and T. M. Chilimbi. Bursty tracing: A framework for low overhead temporal profiling. In *FDDO: Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[17] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *ASE '08: Automated Software Engineering*, pages 184–193, 2007.

[18] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05: Automated Software Engineering*, pages 273–282, 2005.

[19] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02: International Conference on Software Engineering*, pages 467–477, 2002.

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

[21] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI: Programming Language Design and Implementation*, pages 15–26, 2005.

[22] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. Sober: Statistical model-based bug localization. In *FSE: Foundations of Software Engineering*, 2005.

[23] M. C. M. Fortin, Distinguished Engineer. Personal communication. 2007.

[24] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries, 2003.

[25] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *POPL '07: Principles of Programming Languages*, pages 351–362, 2007.