Statistical Debugging with Elastic Predicates

Ross Gore, Paul F. Reynolds, Jr., David Kamensky
{rjg7v, pfr, dmk3d}@virginia.edu

University of Virginia*

Tech Report
CS-2011-02

ABSTRACT
An important class of software, including simulations and computational models, employs stochastic distributions to represent, or support evaluation of, uncertainty in an underlying model. This class of software presents three interesting analysis challenges: 1) effective localization of the sources of unexpected outcomes; 2) effective treatment of stochastic distributions and the floating-point computations that generally accompany them and 3) separation of unexpected outcomes: disassociating valid, but unexpected, results from those reflecting software failure. Traditional debugging and fault localization methods [2-6, 11-15, 17-20, 23, 28, 30-32] have addressed primarily the first challenge, namely localization of sources of faults while assuming character, Boolean, integer and pointer operations. While these methods are effective in general, they are not tailored to software that uses stochastic distributions and floating-point computations. Thus there is opportunity. We introduce ESP, a novel approach to predicate-based statistical debugging. ESP localizes sources of unexpected outcomes in software using stochastic distributions and floating-point operations, thus addressing the first two aforementioned challenges. ESP predicates are elastic rather than uniform and static; each predicate adapts to variable values observed at its program instrumentation point. We present experimental results for established fault localization benchmarks and widely used simulations. For benchmarks employing stochastic distributions, ESP outperforms the best alternatives: Interesting Value Map Pair (IVMP), Cooperative Bug Isolation (CBI) and Tarantula. For traditional benchmarks, ESP performs similar to IVMP and outperforms CBI and Tarantula.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *Debuggers.*

D.2.5 [**Software Engineering**]: Testing and Debugging – *Debugging aids, Testing tools, Tracing.*

General Terms
Algorithms, Measurement, Experimentation

Keywords
automated debugging, fault localization, exploratory software

1. INTRODUCTION
Our interest is in exploratory software. Exploratory software includes stochastic distributions, and the floating-point computations that generally accompany them. Typically exploratory software is used for exploration of uncertainties in an underlying model. Simulations and computational models are examples of exploratory software and have become a common tool for subject matter experts (SMEs) in a variety of disciplines [8]. Predictions based on exploratory software outcomes have entered the mainstream of critical public policy and research decision-making practices, often affecting large numbers of people and valuable resources. Unfortunately SMEs can struggle for decades with the resolution of unexpected exploratory software outcomes. Their methods are generally manual and do not scale. Automated analysis to localize the sources of unexpected exploratory software outcomes will be beneficial.

Prior work in automatically localizing sources of unexpected software outcomes has focused on fault localization. Fault localization is the process of narrowing or guiding the search through source code to help a SME or developer find statements containing faults that cause software failures. The use of stochastic distributions defies fault localization approaches that depend on repeatable execution traces and outcomes [6, 12, 13]. Furthermore, floating-point operations that typically accompany stochastic distributions exceed the capabilities of other existing fault localization methods [2, 3, 17-19, 30]. Unexpected exploratory software outcomes can reflect new knowledge about the underlying model, or a fault. Currently there is no known automated analysis method for separating them. These challenges represent the basis of our motivation.

Separating unexpected valid outcomes from software failures is an interesting, difficult problem. We have not solved it. However, our predicate-based statistical debugger, Exploratory Software Predictor (ESP), does localize sources of the broader class of unexpected outcomes effectively in software employing stochastic distributions and floating-point operations. ESP is novel, and it extends the domain of programs for which fault localization analysis is effective.

Our interest extends to software where model uncertainty is not a factor, but stochastic distributions and floating-point computations are present. Our interest is due to the challenge their presence creates for existing fault localization approaches. ESP performs effectively for these programs and for the traditional programs fault localization techniques have targeted.

ESP is a predicate-based statistical debugging approach focused on identifying single or multiple sources of unexpected outcomes in exploratory software. Predicate-based statistical debugging approaches, such as ESP, represent a class of fault localization techniques that share a common structure. Each approach consists of a set of conditional propositions, or predicates, tested at particular program points. The predicates are given an importance score based on how frequently they are true in the passing and failing test cases for a failing program. A single predicate can be thought of as partitioning the space of all test cases into two subspaces: those satisfying the predicate and those not. The more closely the partitions created by a predicate match the subspaces where the fault is and is not expressed, the better fault predictor the predicate is [17, 19]. The predicates are ranked, based on their importance score. The rankings and scores are provided to SMEs to help in finding and fixing the faults.

In existing predicate-based statistical debugging approaches, predicates are uniform and static. The predicates are uniform in the sense that the same set of conditional propositions is tested at each program point. The predicates are static because each conditional proposition being tested is determined before the execution of the program. ESP moves beyond uniform and static predicates to *elastic predicates*. In ESP, as the program is executed the variable values at each program point are observed. For each program point, similar observed values are clustered to create unique conditional propositions. The unique conditional propositions for each program point are elastic predicates. Elastic predicates create partitions of test cases that more closely match the subspaces where an unexpected outcome is and is not expressed. The result is improved effectiveness. ESP offers improved effectiveness over existing fault localization techniques in terms of: (1) statement based rankings over a set of established benchmarks and several widely used simulations and (2) the importance scores associated with each predicate.

* Dept. of Computer Science, UVA, Charlottesville, VA 22904

2. RELATED WORK

We begin with a review of currently accepted definitions for the terms *failure*, *error* and *fault*. A *failure* occurs when the observed behavior of a program differs from the expected behavior. An *error* represents part of the program state that may lead to the failure. A *fault* is the cause of the error in the program [22].

There is no body of literature that specifically addresses the analysis of exploratory software outcomes; exploratory software possesses attributes that have typically been neglected or avoided. Software debugging methods come the closest to addressing the needs for analyzing unexpected exploratory software outcomes. Here, we review portions of that literature related to ESP.

2.1 Automatically Repairing Software Faults

Several approaches automatically assist developers in fixing failing software. He and Gupta use path-based weakest preconditions to automatically generate program modifications to correct a fault within a function provided the function has formal pre and post conditions [11]. More approaches employ machine-learning algorithms to automatically repair faults in off-the-shelf legacy software [28]. While these approaches do not rely on formal specifications, they do require a failed test case that demonstrates the bug and a number of other test cases that encode the required functionality of the program. Due to the nature of exploratory software, this is not generally possible for SMEs. Furthermore, fault location is a prerequisite to repair. Improvements in locating faults will improve repair solutions.

2.2 Slicing-based Fault Localization

Slicing-based fault localization identifies faulty statements through static, dynamic and relevant slicing. Each of these slicing approaches is defined and described. Static slicing identifies a subset of program statements that *may* influence the value of a variable at a program location [27, 29]. Static slices are used in static fault localization approaches, such as FindBugs, to help identify predefined faulty patterns [4]. Dynamic approaches identify a subset of program statements that *do* influence the value of a variable at a particular point in a given dynamic program execution [1, 16, 30]. The concept of relevant slicing has also been studied to incorporate potential dependencies. Relevant slices computed from the point of an incorrect variable value can identify the subset of statements that could have contributed to the incorrect value. This subset is likely to contain a fault and is significantly smaller than the set of all program statements [27].

2.3 Statistical Debugging

Statistical debugging uses data collected during the execution of passing and failing test cases to rank predicates and statements based on the likelihood they contain a fault. Jiang and Su [14] construct control flow paths linking high ranking predicates to explain software failures. The Nearest Neighbor approach searches for a passing test case execution that is most similar to a failing test case execution, contrasts the two executions, and uses the information to identify the most suspicious parts of the program [23]. SOBER models the evaluation patterns of predicates in passing and failing test cases, and considers a predicate to be relevant to a fault if its evaluation pattern in failing test cases significantly differs from the observed pattern in passing test cases [20]. In recent fault localization evaluations using the Siemens Benchmark Suite [25], two approaches stand out in terms of efficiency and effectiveness: Tarantula and CBI [2, 5, 6, 12, 15, 20]. We describe each further.

Tarantula is a statistical debugging approach that analyzes how frequently a program statement is executed in passing and failing test cases. It has been shown to be one of the most effective and efficient fault localization approaches for identifying faulty statements in the Siemens Benchmark Suite [15]. The approach is efficient because it only takes into account statement coverage information of passing and failing test cases. However, its effectiveness is limited due to its inability to analyze values of variables within the program [6, 12].

CBI employs predicate-based statistical debugging [17]. CBI uses instrumentation to collect feedback reports that describe which predicates are observed in a test case as well as the outcome of the test case. CBI assigns an importance score to all predicates and outputs a complete list of predicates and their importance scores in descending order. A developer or SME uses the list to identify areas of the program related to the observed failure. Section 3.2 describes how importance scores for predicates are computed in CBI. Using predicates ranked by importance scores, CBI has identified previously unknown faults in several widely used applications [17-19, 32]. SOBER was shown to be slightly more effective than a variant of CBI [20]. However, this was prior to CBI's use of importance scores, which significantly improve its effectiveness.

Several enhancements have been studied for CBI and other predicate-based statistical debugging approaches. HOLMES and Adaptive Bug Isolation can significantly reduce the number of program points that need to be instrumented and monitored to identify predicates with high importance scores [2, 5]. Reduction in the number of instrumented program points can improve the overall efficiency of CBI but it does not improve the effectiveness. Another enhancement, compound Boolean predicates, combines predicates with Boolean formulae to create a richer predicate vocabulary [3]. Compound Boolean predicates and elastic predicates improve the effectiveness of predicate-based statistical debugging approaches in a complementary fashion. Compound Boolean predicates offer a richer vocabulary while elastic predicates offer more precise predicates. Their possible combination has not yet been explored.

2.4 State-altering Fault Localization

State alteration approaches modify the program state of an executing program in an attempt to isolate faults. In the Delta Debugging framework, failure-inducing input is identified that allows for the computation of cause-effect chains for failures that are linked to faulty statements [6]. This is accomplished by swapping the values of variables between one passing and one failing test case. Predicate Switching is another state alteration approach which attempts to isolate faulty statements by identifying predicates whose outcomes can be altered during a failing test case to cause the test case to pass [31]. The most effective state-alteration approach, when evaluated with the Siemens Benchmark Suite, is IVMP. IVMP is a value profile based approach that involves searching for the program statements that can be shown to affect the output of a failing execution such that the incorrect output becomes correct. This is done by replacing the values used at a statement during the execution of a failing test case with an alternate set of values, then determining whether the altered execution passes the test case. The successful changes often occur at statements containing faults or statements that are directly linked to statements containing faults through data or control-flow dependencies [12, 13].

State-altering approaches are not always effective for programs that use stochastic distributions. These approaches require an unaltered failing test case to fail every time it is executed [6, 12]. This requirement enables variable values in a failing test case to be identified and altered to attempt to create a passing test case. However, stochastic distributions cause some variable values to vary from execution to execution. These values can result in a test case that passes one time it is executed and fails another violating the requirement and diminishing the effectiveness of the method as a result.

3. ELASTIC PREDICATES

Numerous statistical debugging approaches have been proposed to identify predicates that are good failure predictors [14, 17-20, 31, 32]. The most effective of these approaches analyze variable values within a program. However, accounting for all possible values at each assignment to a given variable is impractical [17]. Instead, these approaches utilize an instrumentation scheme with uniform and static predicates at each variable assignment statement. This instrumentation scheme is referred to as *single variable* and is an extension of the returns scheme in CBI. The most common single variable scheme for an assignment to a variable x uses these predicates [17]:

1. $x < 0$
2. $x = 0$
3. $x > 0$

The elastic predicates used in ESP replace uniform and static predicates. ESP provides a single variable instrumentation scheme that records the mean, μ_x, and standard deviation, σ_x, for a given variable x for all test case executions. Using μ_x and σ_x these predicates are constructed:

1. $(\mu_x - l\sigma_x) > x$
2. $(\mu_x - k\sigma_x) \le x < (\mu_x - j\sigma_x)$
3. $\mu_x = x$
4. $(\mu_x + j\sigma_x) \le x < (\mu_x + k\sigma_x)$
5. $(\mu_x + l\sigma_x) > x$

For the predicates $0 \le j < k \le l$ is assumed. In ESP the default nine elastic predicates for the single variable instrumentation scheme are: j={0,1,2}, k={1,2,3}, l={3}. These predicates partition the values for an instrumented program point for three standard deviations above and below the mean of x.

Multiple variables within a program can have important relationships that cannot be captured with a single variable instrumentation scheme. The Daikon project identifies implicit invariants to aide program evolution and understanding [7]. ESP and existing predicate-based statistical debugging approaches identify near-invariants that are only violated when the program fails test cases. This instrumentation scheme is called *scalar pairs* [17]. Within CBI the scalar pairs scheme instruments assignments to character, integer and pointer typed variables. ESP extends the scheme to floating-point typed variables. The name *scalar-pairs* refers to the data type of the variables.

In existing predicate-based statistical debugging approaches, the scalar pairs scheme examines possible invariants from a uniform, static set. At each assignment to a variable x, these approaches identify all other same-typed local or global variables $y_1, y_2, \ldots, y_n$ that are currently in scope. For each pair of variables (x, y_i) the scheme compares the new value of x with the existing value of y_i with these predicates [17]:

1. $x < y_i$
2. $x = y_i$
3. $x > y_i$

In ESP, the mean μ_{x-y_i} and standard deviation σ_{x-y_i} of the difference between the new value of x and the existing value of y_i, is computed for each pair of variables (x, y_i). Using μ_{x-y_i} and σ_{x-y_i} these elastic predicates are constructed:

1. $(\mu_{x-y_i} - l\sigma_{x-y_i}) > x - y_i$
2. $(\mu_{x-y_i} - k\sigma_{x-y_i}) \le x - y_i < (\mu_{x-y_i} - j\sigma_{x-y_i})$
3. $\mu_{x-y_i} = x - y_i$
4. $(\mu_{x-y_i} + k\sigma_{x-y_i}) \le x - y_i < (\mu_{x-y_i} + j\sigma_{x-y_i})$
5. $(\mu_{x-y_i} + l\sigma_{x-y_i}) < x - y_i$

Again, for the elastic predicates $0 \le j < k \le l$ is assumed. The default values for j, k and l are the same as for the single variable scheme. These default elastic predicates partition the values for an instrumented program point for three standard deviations above and below the mean of $x - y_i$.

In the elastic predicate single variable and scalar pairs instrumentation schemes, variable values are observed throughout executions of the failing program to create unique predicates at each instrumented program point. While elastic predicates require more space and time to compute than uniform and static predicates, the expectation is that these predicates will have higher importance scores and be better failure predictors because they are more tailored to the failing program. Experimental data about the space and time used by ESP is provided in Section 4.

3.1 Importance Scores

Elastic predicates and uniform and static predicates require two data structures for each executed test case to compute importance scores. Importance scores are used to rank predicates. The two data structures are: (1) a one bit feedback report, *R*, indicating if the test case was passed or failed and (2) a bit vector, *BV*, with one bit for each instrumented predicate. Within *BV* each bit indicates if the corresponding predicate is observed to be true at least once during execution of the test case [17-19, 32]. Once all test cases have been executed, the importance scores for each predicate can be calculated from these data structures.

Throughout the development of CBI, Liblit et al. explored several different formulas to compute importance scores for predicates [17-19, 32]. They determined that predicates that are sensitive and specific should yield high importance scores. Sensitive predicates account for a high percentage of failed test cases and specific predicates do not predict failure for successful test cases. The sensitivity and specificity are computed as we describe next. The data from each feedback report *R* and each corresponding bit vector *BV* is aggregated into four measures for a predicate *p* [17, 19, 32]:

1. *S*(*p* obs) and *F*(*p* obs) the number of successful and failed test cases in which *p* was evaluated.
2. *S*(*p*) and *F*(*p*) the number of successful and failed test cases in which the value of *p* was evaluated and found to be true.

Sensitivity: log(*F*(*p*))/log(*NumF*). *NumF* is the total number of failing runs. This ratio describes the percentage of the failing test cases the predicate accounts for.

Specificity: *Increase*(*p*). *Increase*(*p*) is the amount by which *p* being true increases the probability of failure over simply reaching the statement where *p* is defined.

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

Sensitivity and specificity are combined via their harmonic mean. This metric is the importance score for the predicate [19, 32].

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

While the formula to compute importance scores for uniform and static predicates and elastic predicates is the same, the scores are not. The example in Section 3.2 highlights the differences.

3.2 Example

Applying CBI and ESP to an example program used in previous fault localization studies helps elucidate the differences between uniform and static predicates (CBI) and elastic predicates (ESP). Figure 1 shows the source code of the `more_arrays()` function in the 1.06 version of the GNU implementation of BC, a basic command-line calculator tool [17, 20].

```
152 void
153 more_arrays()
154 {
155   int indx;
156   int old_count;
157   bc_var_array **old_ary;
158   char **old_names;
159
160   /* Save the old values. */
161   old_count = a_count;
162   old_ary = arrays;
163   old_names = a_names;
164
165   /* Increment by a fixed amount and allocate. */
166   a_count += STORE_INCR;
167   arrays = (bc_var_array **) bc_malloc (a_count*sizeof(bc_var..
168   a_names = (char **) bc_malloc (a_count*sizeof(char *));
169
170   /* Copy the old arrays. */
171   for (indx =1; indx < old_count; indx++)
172     arrays[indx] = old_ary[indx];
173
174
175   /* Initialize the new elements. */
176   for (; indx < v_count; indx++)
177     arrays[indx] = NULL;
178
179   /* Free the old elements. */
180   if (old_count != 0)
181     {
182       free (old_ary);
183       free (old_names);
184     }
185 }
```

Figure 1: The source code of the more_arrays() function in the 1.06 version of the GNU implementation of BC.

Table 1: The top ranked CBI predicates for more_arrays()

Filename	Line #	Function	Predicate	Importance Score
storage.c	166	more_arrays	a_count>0	0.1312
storage.c	161	more_arrays	old_count > 0	0.1242

Table 2: The top ranked ESP predicates for more_arrays()

Filename	Line #	Function	Predicate	Importance Score
storage.c	176	more_ arrays	indx > μ+3σ	0.988
storage.c	176	more_ arrays	μ+2σ< indx ≤ μ+3σ	0.846

The `more_arrays()` function in BC is responsible for increasing the number of arrays needed for computing. The logic within the function is an example of buffer reallocation. Line 167 allocates a larger chunk of memory. Line 171 is the top of a loop that copies values over from the old, smaller array. Line 176 completes the resize by zeroing out the new extra space. However, there is a fault in the function. In this example, we apply the single variable instrumentation scheme for CBI and ESP to identify the fault in `more_arrays()`. We use 1,000 randomly generated valid BC programs with various sizes and complexities as test cases. This approach to generating test cases is modeled after Liu et al.'s case study of BC [20]. The top two ranked predicates for the methods are shown in Table 1 and Table 2.

The top ranked CBI predicates do not make the fault within `more_arrays()` easy to discern. The predicates direct the user to two different line numbers with similar importance scores. These line numbers do not represent the program statement containing the fault and do not have importance scores that are high. They are not good failure predictors. In contrast, the location of the fault and the cause of the failure are evident in the top two ranked ESP predicates. The top ranked predicates show that `indx` is unusually large in failing test cases. The predicate suggests that failures occur when the input to BC defines an unusually large numbers of arrays.

Examination of `more_arrays()` reveals the hypothesis to be true. The allocation on line 167 requests space for `a_count` items. The copying loop on line 171 ranges from 1 through `old_count - 1`. The zeroing loop on line 176 continues on from `old_count` through `v_count - 1`. And here ESP finds fault: the new storage buffer has room for `a_count` elements, but the second loop is incorrectly bound by `v_count` instead.

The location of the fault and the cause of the failure are clear after identification and explanation. However, this fault was present and undiscovered for ten years in BC [17]. In this example, ESP is more effective than CBI because of the elastic predicates it employs. The elastic predicates are based on observed variable values creating partitions of test cases that more closely match the subspaces where the fault is. CBI performs poorly in this example because most of the values assigned to the variables in `more_arrays()` are greater than zero. Since the predicates used in CBI are uniform and static, most variable values at each program point satisfy the same predicate. The result is predicates with low specificity and importance scores. Specifically, the value of `indx` in line 176 is always greater than zero resulting in a predicate without specificity.

The improved effectiveness provided by elastic predicates is not achieved without cost. Elastic predicates require the mean and standard deviation of each instrumented program point to be computed. Within ESP, these calculations take more space and time to compute than the uniform and static predicates used in CBI. Further analysis of the space and time used by ESP for the programs in our evaluation is provided in Section 4.

4. EVALUATION

4.1 Experimental setup

Recent research uses adaptive sampling to reduce the number of predicates that need to be instrumented to achieve effective statistical debugging results [2, 5]. Parallelization of each approach included in our evaluation is also possible. While this work can improve the efficiency of each approach, here it is ignored and only naïve sequential implementations are considered. Exploring the extent to which the efficiency of ESP can be improved by adaptive sampling approaches and parallelization is an avenue for future work.

4.1.1 Implementation and Hardware

Our implementation of ESP closely mirrors the published implementation of CBI with several exceptions. First, ESP does not use random sampling of test case executions to reduce overhead; it always employs complete monitoring of each test case execution. This difference reflects the different goals of ESP and CBI. ESP is deployed as a stand-alone fault localization tool for a single SME. Its goal is to identify failure predicting predicates as effectively and efficiently as possible for the test cases provided. Second, ESP employs nine elastic predicates described in Section 3.1 and one uniform and static predicate at each instrumented program point. The uniform and static predicate tests the instrumented program point to determine if it equals zero. To ensure non-overlapping predicates it is always tested first. The inclusion of a uniform and static predicate in ESP is a reflection of how frequently used and important the value zero is in software, exploratory or not [4]. Finally, ESP uses an additional data structure to compute importance scores. Recall, for each test case in a test suite both ESP and CBI require: (1) a one bit feedback report, R, indicating if the test case was passed or failed and (2) a bit vector, BV, with one bit for each instrumented predicate. However, ESP implements elastic predicates with a third data structure, an m-by-n matrix, composed of 64-bit entries.

Within the m-by-n matrix, m represents the number of instrumented program points in the simulation and n represents the number of times a value is assigned to a program point within a test case. Each entry in the matrix is 64-bits to enable double precision floating-point numbers to be recorded. As each test case is executed, each m-by-n matrix and each feedback report R is filled. Once all test cases have been executed the mean and standard deviation for each instrumented program point has been calculated. Then, the matrix for each test case is traversed and the corresponding bit vector BV is filled and importance scores are calculated as they are in CBI.

In practice n is small for the majority of instrumented program points. The exceptions are program points within loops. As a result, ESP employs resizable rows within the m-by-n matrix, where each size increase is an order of magnitude (e.g. 10, 100, 1,000, 10,000, 100,000). This approach manages space well for the benchmarks and simulations used in our evaluation. Experimental data summarizing the distribution of n for our evaluation is provided in Section 4.4.

It is important to note that employing elastic predicates does not require an m-by-n matrix. Instead, this data structure can be eliminated in a naïve implementation at the cost of additional execution time. In this space efficient implementation, each test case is executed and the mean and standard deviation of each predicate are computed in an online manner. With the elastic predicates defined each test case can be executed again and the feedback reports and bit vectors can be filled as they are in CBI.

Along with our implementation of ESP, we implemented versions of Tarantula, IVMP and CBI. These implementations reflect [15], [12, 13], and [17] respectively. They are discussed in more detail in Section 4.1.3. Our experiments for each approach were run on a server with two Intel Xeon quad-core processors at 3.00 GHz and 48 GB of RAM.

4.1.2 Subject programs and test suites

The subset of programs from the Siemens Benchmark Suite used for our experiments is listed in Table 3. The programs, along with their corresponding faulty versions and test cases, were obtained from [25]. All Siemens faulty versions contain seeded faults [10]. These faults are computation-related (as opposed to memory-related), involving fault types such as operator and operand mutations, missing and extraneous code, and constant value mutations. Most faulty versions are seeded with a single fault in a single statement, but some faulty versions involve several statements. Several faulty versions were excluded because they did not yield any failing test cases from the provided test cases. These versions have been excluded in previous fault localization evaluations with the Siemens Benchmark Suite [12, 13]. The programs in Table 3 were chosen from the Siemens Benchmark Suite because they (1) contain floating-point computations or (2) could be easily modified to utilize stochastic distributions in a meaningful way. These characteristics make them similar to exploratory software and good candidates to test the effectiveness and efficiency of ESP against the best fault localization alternatives. The modifications we made are described after each program is introduced.

Table 3: Siemens Benchmark Suite evaluation programs

Program Name	Lines of Code	Num. of Versions	# of Test Cases	Program Description
tcas	138	41	1608	Altitude separator
totinfo	396	23	1052	Statistic computation
sched	299	9	2650	Priority scheduler
sched2	297	9	2710	Priority scheduler

Within our subset of the Siemens Benchmark Suite are four programs: tcas, totinfo, sched and sched2. The tcas program contains no loops and represents one conditional check spread across several functions; it takes as input a set of integer parameters and reports one of three output values. totinfo reads a collection of numeric data tables as input and computes statistics for each table as well as across all tables. Programs sched and sched2 are priority schedulers for processes, taking as input a number of processes and a list of scheduling commands, and outputting the processes as they complete in priority order. For each faulty version of each program, we also created a stochastic version. For the totinfo and tcas programs we modified the value of constants to be sampled from a uniform distribution with minimum value of half the constant and a maximum value of one and a half times the constant. These modifications represent the uncertainty of surrounding measurements used in exploratory software and also reflect the seeded constant mutation faults in the Siemens Benchmark Suite [25]. In the two priority scheduler programs we modified the programs to include arrival times drawn from a normal distribution for the processes. This modification is consistent with existing queueing simulations [9]. These stochastic versions represent benchmark programs that are similar to exploratory software. Our stochastic Siemens Benchmark Suite is available [26].

4.1.3 Fault localization approaches and scoring

In our experiments we compare the fault localization effectiveness of the following four approaches that rank program statements.

IVMP. Within IVMP, each statement's ranking is based on the number of failing executions in which a state alteration within the statement results in a passing execution. Any ties within this scheme are broken using the Tarantula suspiciousness formula described in this section.

ESP. ESP is a predicate-based statistical debugging approach that employs elastic predicates. In this evaluation both the single variable and scalar pairs instrumentation schemes for ESP are considered. Given a list of predicates ranked by importance scores, ESP and CBI rank statements according to the following:

1. For each statement identify the corresponding predicate with the highest importance score and move the statement and its importance score to set *ST*.
2. Rank the statements in *ST* by importance score.

This statement ranking strategy is consistent with existing predicate-based statistical debugging approaches [20].

CBI. CBI is a predicate-based statistical debugging approach that employs uniform and static predicates. In our evaluation both the single variable and scalar pairs instrumentation schemes for CBI are considered. Existing implementations of CBI only consider Boolean, character, integer, or pointer typed variables. However, in the interest of a fair evaluation we extend our implementation of CBI to consider floating-point type variables. Furthermore, our implementation of CBI does not use random execution sampling to reduce overhead. Instead it completely monitors each test case execution enabling the best possible effectiveness results for CBI. CBI ranks statements using the same process as ESP.

Tarantula. Statements within Tarantula are ranked in descending order of suspiciousness. The suspiciousness, *susp*, of a statement *s* is defined as:

$$susp(s) = \frac{\dfrac{failed(s)}{totalFailed}}{\left(\dfrac{failed(s)}{totalFailed} + \dfrac{passed(s)}{totalPassed}\right)}$$

Here, *failed(s)* and *passed(s)* are the number of failing and passing executions in which *s* is included. *totalFailed* and *totalPassed* are the total number of failing and passing executions.

In our experiments, we rank only those program statements that are executed by failing test cases using the test suite associated with each faulty version of each program. When multiple statements are tied for a particular rank, all tied statements are given a rank value equal to the maximum rank value from among the tied statements. This reflects the conservative assumption that a SME will examine all tied statements before any faulty statement within the set of tied score statements can be found.

To evaluate each approach we assign a score to each ranked set of statements that is the percentage of program statements executed by failing test cases in the test suite that need not be examined given the rank order of the statements. Given a ranked list of statements *S*, where the faulty statement occurs at rank *r* and *n* total statements are executed by failing test cases the *score* for the approach is: $score(S) = (n-r)/n * 100$.

Finally, there are two details pertaining to certain types of faults. First, for IVMP, faults in constant assignment statements (15 out of a total of 82 faulty versions), cannot be found. However, IVMP can detect faults in the statements where the constant representing the fault is used. To conservatively evaluate our effectiveness against IVMP we consider a constant assignment statement to be examined by a SME when the assignment statement is examined or when a statement explicitly using the constant is examined. Second, faults that involve missing statements (16 of the 82 faulty versions) cannot be ranked and examined by a user because they are missing. For these versions the statements directly adjacent to the missing code qualify as the faulty statement. These two issues are not unique and are handled the same way in previous evaluations [12, 13].

4.1.4 Identifying important predicates and scoring

In our experiments we also compare the effectiveness of CBI and ESP in terms of identifying failure predicting predicates. We measure the highest importance score returned by each approach for each faulty version of each program in both versions of the Siemens Benchmark Suite. This is a traditional evaluation of effectiveness for predicate-based statistical debugging approaches and is described further in Section 4.3 [2, 6, 17, 19, 32].

4.2 Statement Ranking Effectiveness

Our experimental results for the Siemens Benchmark Suite are shown for each of the statement ranking approaches in Figures 2-5. In the figures the x-axis represents the lower bound of each score range, and the y-axis represents the percentage of faulty versions with a score greater than or equal to the lower bound. Figures 2 and 3 show the percentage of faulty versions in which each approach computes a ranked list of statements in the specified score range for the traditional Siemens Benchmark Suite. Figure 2 shows these results for IVMP, Tarantula, ESP and CBI under the single variable instrumentation scheme. Figure 3 shows the results for IVMP, Tarantula, ESP and CBI under the scalar pairs instrumentation scheme. Figures 4 and 5 show the same data as Figure 2 and 3 respectively, except they reflect the Siemens Benchmark Suite that uses stochastic distributions. These presentations of data follow the convention of Jones et al. [15]. However, whereas Jones et al. computes scores with respect to the total number of program statements, we compute scores with respect to the total number of statements executed by failing test cases in the suite. Figures 2 and 3 show that for the traditional Siemens Benchmark Suite the IVMP approach and ESP overall perform much better than Tarantula or CBI. However, the results significantly change for the Siemens Benchmark Suite that uses stochastic distributions, as shown in Figures 4 and 5. IVMP is no longer an effective approach because failing test cases executions are not necessarily repeatable. As a result ESP outperforms any alternative. We examine the data for each approach.

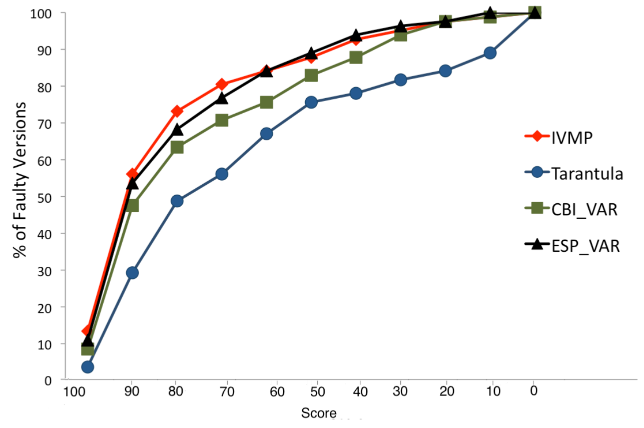

Figure 2: Statement ranking approaches with ESP and CBI single variable for the traditional Siemens Benchmark Suite.

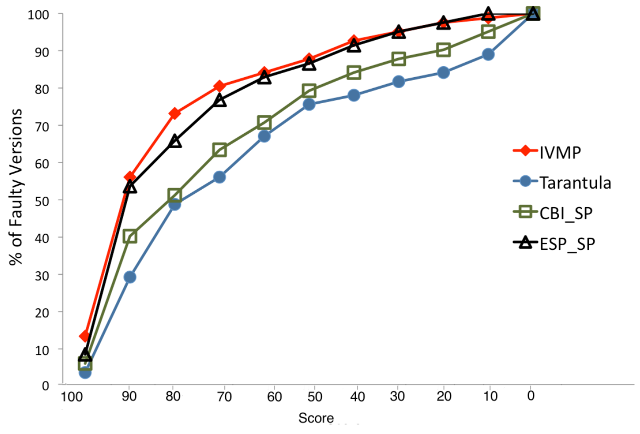

Figure 3: Statement ranking approaches with ESP and CBI scalar pairs for the Siemens Benchmark Suite that does not use stochastic distributions.

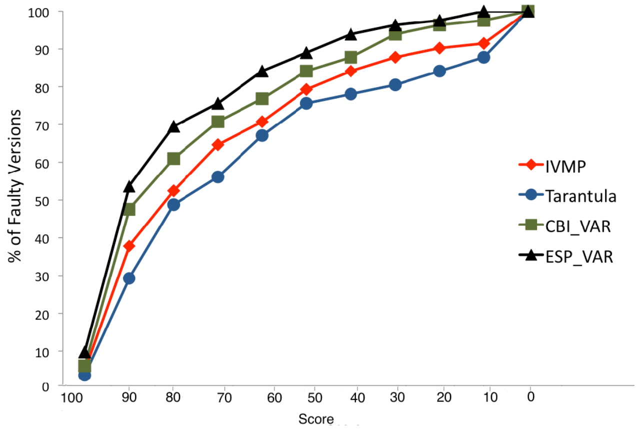

Figure 4: Statement ranking approaches with ESP and CBI single variable for the traditional Siemens Benchmark Suite.

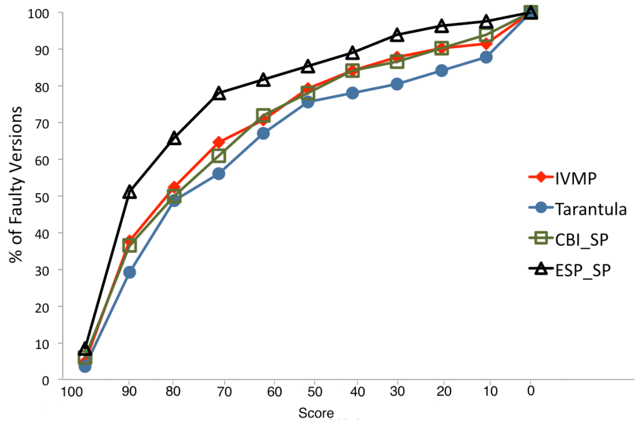

Figure 5: Statement ranking approaches with ESP and CBI single variable for Siemens Benchmark Suite that does use stochastic distributions.

4.2.1 ESP vs. IVMP

Within the traditional Siemens Benchmark Suite that does not use stochastic distributions IVMP performs better than ESP for tcas, sched and sched2. In these programs IVMP had a score of 90% or higher 40 times while ESP only had a score of 90% or higher 32 times. However, ESP performed well for the totinfo program, which frequently employs floating-point computations, and IVMP did not. Within totinfo it is very difficult for IVMP to perform state alterations that cause a failing test case to pass. This difficulty is due to the level of precision in the floating-point computations that generate the program's output. As a result the approach resorts to the ranking system in Tarantula for most statements. In the evaluation IVMP performs the same as or worse than Tarantula for 15 of the 23 faulty versions of totinfo.

Within the Siemens Benchmark Suite that does use stochastic distributions IVMP performs poorly. This is because the suite does not meet the state-alteration approach requirement, which requires a failing test case to fail each time it is executed. When applying IVMP to software that does not meet this requirement state-alterations that cause failing test cases to pass are harder to identify and not as meaningful [12, 13]. They cannot be isolated from variations in the execution trace that are caused by the stochastic distributions. Furthermore, the use of stochastic distributions introduces more floating-point computations into the programs, which IVMP does not always handle well.

4.2.2 ESP vs. CBI

ESP performs better than CBI under both instrumentation schemes for the traditional Siemens Benchmark Suite and the suite that uses stochastic distributions. Under the single variable instrumentation scheme there are only 21 out of the total 164 faulty program versions where ESP assigned a lower rank to the statement containing the fault than CBI. These instances were a result of two different scenarios. In the first scenario the fault is triggered in a small number of test cases by several values for a program point that are greater than zero but only slightly larger than all other observed values. In the second scenario the fault is triggered in a small number of test cases by several values for a program point that are less than zero but only slightly smaller than all other observed values. In both scenarios a uniform and static predicate which tests if the values for a program point are greater or less than zero is more specific to the failure than the elastic predicate used in ESP. The elastic predicate clusters all the similar values together while the uniform and static predicate separates those values that trigger the fault from the other similar values.

Under the single variable instrumentation scheme ESP and CBI assigned the same rank to the statement containing the fault in 36 other cases. In theses cases the uniform and static predicate used in both approaches identifies the fault. In the remaining 107 cases ESP outperforms CBI. ESP's performance is attributed to elastic predicates, which result in predicates with higher importance scores and better fault localization capabilities. The importance scores for CBI and ESP are examined further in section 4.3.

CBI's poor performance under the scalar pairs instrumentation scheme is noteworthy. For the traditional Siemens Benchmark Suite it only slightly outperforms Tarantula, an efficient approach that only analyzes statements, not variable values. This poor performance is consistent with the Liu et al. evaluation of CBI for the Siemens Benchmark Suite [20]. In this evaluation the scalar pairs instrumentation scheme did not add any effectiveness to CBI compared to the other implemented instrumentation schemes. ESP offers a significant improvement over CBI for the scalar pairs scheme, which we discuss in Section 4.3.

4.2.3 ESP vs. Tarantula

Along with ESP and CBI, the Tarantula approach did not experience a significant decrease in effectiveness when applied to the Siemens Benchmark Suite that uses stochastic distributions. However, compared to ESP, Tarantula is not effective for either version of the Siemens Benchmark Suite. For the traditional suite the single variable instrumentation scheme of ESP was able to uniquely identify the statement containing the fault (assign it rank 1) in 14 cases. Tarantula was able to do so in only 3 cases. Even though the ESP approach was able to uniquely identify the faulty statement in 14 cases, only 9 cases yielded scores of 99% or more because in the tcas program the number of statements executed in failing test cases was too few to yield a score of 99%.

4.3 Importance Score Ranking Effectiveness

Existing studies show that when SMEs or developers are provided with a good failure predicting predicate they are able to quickly identify the fault within a program whether the predicate corresponds to the statement containing the fault or not [17]. The importance score of the highest ranked predicate reflects this measure of effectiveness. Figure 6 shows the importance score of the highest ranked predicate for both ESP and CBI for the traditional Siemens Benchmark Suite. Figure 7 shows the same data for the suite that uses stochastic distributions. In the figures the x-axis represents the lower bound of each importance score range, and the y-axis represents the percentage of faulty versions with a score greater than or equal to the lower bound.

The trends in the importance scores in Figures 6 and 7 are similar to the trends in the statement ranking scores in Figures 2-5; ESP outperforms CBI by a similar margin. They are related. The elastic predicates do a better job of creating partitions of test cases where the fault is and it not expressed than the uniform and static predicates. This leads to a better ability to identify faults. The importance scores and statement ranks of the elastic predicates in ESP reflect this for both versions of the Siemens Benchmark Suite. The figures also explain the poor performance of CBI's scalar pairs scheme in our statement ranking evaluation. The uniform and static predicates are unable to identify relationships between variables that are only violated in failing test cases yielding few high scoring predicates. In contrast, the elastic predicates adapt to observed differences in variable values yielding many high scoring predicates and an effective approach.

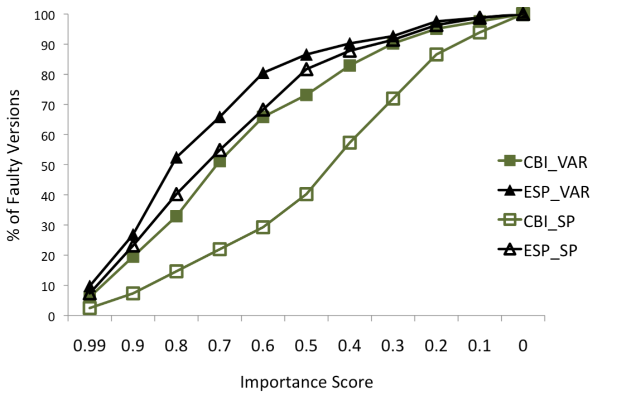

Figure 6: Highest importance score associated with a predicate for ESP and CBI for the traditional Siemens Benchmark Suite.

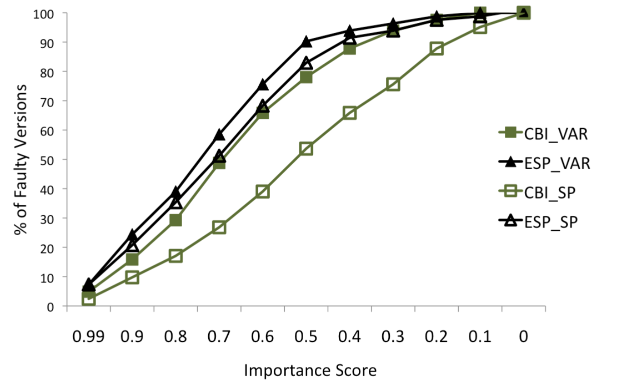

Figure 7: Highest importance score associated with a predicate for ESP and CBI for the Siemens Benchmark Suite that uses stochastic distributions.

Overall, the results of Section 4.2 and 4.3 demonstrate that ESP is more effective than CBI for both the traditional Siemens Benchmark Suite and the version that uses stochastic distributions. Furthermore, for the version that uses stochastic distributions ESP is the most effective fault localization approach. Also, ESP approaches the effectiveness of the best available approach for the traditional Siemens Benchmark Suite, which existing fault localization tools target. Next we explore how ESP performs compared to the other approaches in terms of efficiency.

4.4 Efficiency

Figures 8 and 9 summarize the efficiency results for our evaluation. Tarantula is the most efficient approach in our study because it only takes into account statement coverage information of passing and failing test cases. The other approaches also analyze the values of variables. As a result, we compute the efficiency of each approach relative to Tarantula's. This relative performance measure is: $RP = \dfrac{time_{Tarantula}}{time_{other}}$.

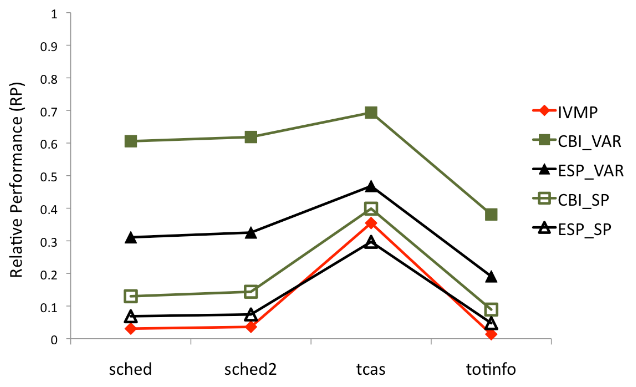

Figure 8: *RP* for each statement ranking approach for the programs in the traditional Siemens Benchmark Suite.

Figures 8 and 9 reveal several trends in the relative performance of the approaches in our evaluation. Each approach improves for the tcas benchmark and degrades for the totinfo benchmark. The tcas and totinfo programs are the least and most computationally intensive programs in the suite respectively. Tarantula does not analyze variable computations so it does not reflect these factors. However, the other approaches do analyze variable computations and reflect the computational characteristics of tcas and totinfo.

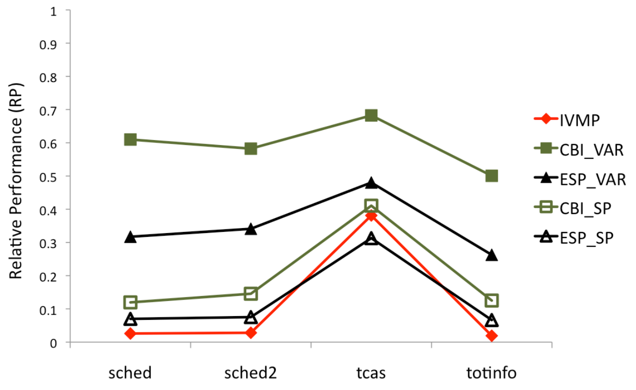

Figure 9: *RP* for each statement ranking approach for the programs in the Siemens Benchmark Suite that use stochastic distributions.

Given these expected fluctuations, the relative performance of ESP and CBI is independent of the programs in the evaluation to which they are applied. IVMP is not. For the the traditional and stochastic versions of the totinfo program the relative performance of IVMP approaches zero. This drastic degradation is due to several failing totinfo test case executions that IVMP repeatedly re-executes in an attempt to find state alterations resulting in a passing execution. IVMP is the only approach in our evaluation that re-executes failing test cases. As a result, for some programs, such as totinfo, repeated re-executions of failing test cases will be required, making IVMP inefficient.

Figures 8 and 9 also highlight the extra time required when ESP and CBI are used in the scalar pairs instrumentation scheme as opposed to the single variable scheme. While the extra time required for these approaches does not result in improved effectiveness for the Siemens Benchmark Suite, the scalar pairs scheme has been shown to be effective in CBI for several widely used programs [17]. Several performance optimizations for the scalar pairs scheme are also available in [17]. These are not implemented for ESP or CBI in our evaluation.

We expected the difference in the relative performance of ESP and CBI to be approximately constant throughout the evaluation. ESP and CBI both employ a set number of predicates at each instrumented program point resulting in relative performance that is independent of the faulty program. Recall, the difference in the implementation of CBI and ESP is the additional data structure, an *m*-by-*n* matrix, employed by ESP to construct elastic predicates. Within ESP, the matrix is generated for each executed test case. The rows in the matrix hold the values assigned to instrumented program points. Figure 10 shows the distribution of the length of each row in the matrices for the Siemens Benchmark Suite that uses stochastic distributions. The x-axis represents the length of each matrix row and the y-axis represents the percentage of matrices with a row of the specified length. Each axis is at log scale. The distribution of the row length for the traditional Siemens Benchmark Suite is similar.

Figure 10 shows that for more than 99% of the matrices the length of each row is 50 or less. Furthermore, for ~95% of the matrices the length of each row is 10 or less. In future work we will explore more efficient approaches for storing the values assigned for an instrumented program point. However, for the Siemens Benchmark Suite the additional matrix required by ESP is a compact data structure for most program points.

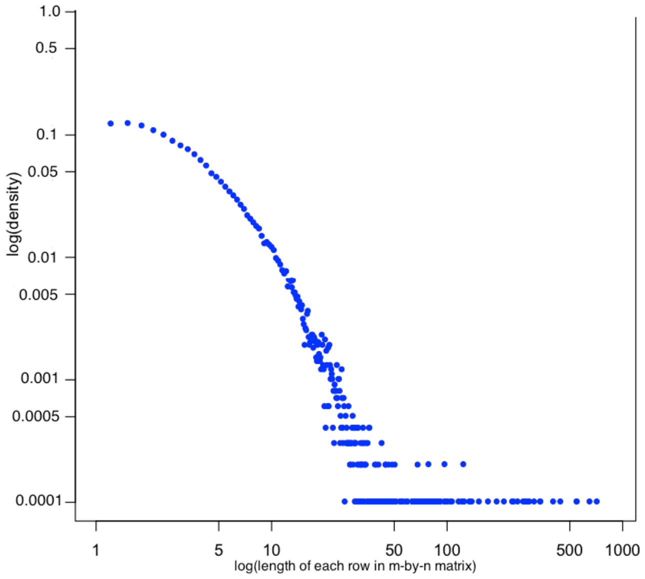

Figure 10: The distribution of the length of each row in each of the m-by-n matrices used by ESP. Note the log scales.

ESP approaches the effectiveness of the best available fault localization approach, IVMP, for the traditional Siemens Benchmark Suite in less time. Furthermore, ESP outperforms all other approaches for a version of the suite that shares characteristics of ESP's target domain, exploratory software. ESP achieves this effectiveness with only a constant decrease in efficiency relative to the most efficient approach in our evaluation, Tarantula. Next we will evaluate ESP against the best fault localization alternatives for two widely used simulations.

4.5 Widely Used Simulations

We conducted additional experiments to determine how effectively and efficiently ESP localizes faults for two widely used simulations. The simulations in the evaluation are shown in Table 4. One fault, similar to those in the Siemens Benchmark Suite, is seeded in each simulation.

Table 4: Evaluated simulations and simulators

Sim. Name	Modified Simulator	Total Lines of Code
TCP Protocol	ns-2 Network Simulator [21]	11458
CPU Scheduling	Queueing Toolkit [24]	2561

For each simulation, the rank of the statement containing the fault for each of the approaches is shown in Table 5. The *RP* and importance score of the highest ranked predicate in ESP and CBI are also provided. The best rank and highest importance score among the approaches for each simulation are shown in bold and italicized. For this portion of the evaluation ESP and CBI used only the single variable instrumentation scheme.

Table 5: Effectiveness and efficiency results

Sim. Name	Faulty Statement Rank (ESP, CBI, IVMP, Tar)	Importance Score (ESP, CBI)	*RP*
TCP Protocol	(***1***, 16, 138, 144)	(***1.00***, 0.925)	.30951
CPU Disk Scheduling	(***3***, 96, 64, 213)	(***0.985***, 0.765)	.32508

Table 5 shows that ESP is capable of significant improvements in fault localization effectiveness over the best available alternatives for analyzing widely used exploratory software. Moreover, the decrease in relative performance for ESP remained constant when compared to Tarantula. These efficiency results seem reasonable in an automated context considering: (1) the significant improvement in the rank of the statements containing the fault and (2) the ability to find better failure predicting predicates with higher importance scores than existing approaches. Even though these simulations are significantly larger than the Siemens Benchmark Suite, ESP's performance relative to Tarantula is approximately the same as it was in the Siemens Benchmark Suite. These results illustrate that it is not program size that determines the efficiency of our approach, but the number of test cases included with the program. This has been shown to be the limiting factor of efficiency for existing predicate-based statistical debugging approaches [20].

4.6 Multiple Faults

Several of the cases in the Siemens Benchmark Suite contain multiple faults. This is not uncommon; many examples of exploratory software also contain multiple faults. In our evaluation, each fault localization approach was only required to identify one fault per program. This allowed us to conservatively evaluate the effectiveness of ESP against the best available alternative for the Siemens Benchmark Suite, IVMP. However, in programs with multiple faults IVMP's effectiveness can diminish. It is difficult for IVMP to differentiate among multiples faults in a program because it has trouble identifying state alterations in failing test cases that have different effects on the program output [12]. Several modifications to IVMP have been suggested to address this issue. However, these modifications can make IVMP even more inefficient and do not guarantee multiple faults are distinguished from one another [13]. In contrast, the effectiveness and efficiency of ESP, CBI and other existing predicate-based statistical debugging approaches does not diminish. ESP uses the following established algorithm that guarantees a SME a failure predicting predicate for each fault in a program [17, 32].

1. Rank each predicate in descending by importance score.
2. Remove the top-ranked predicate *p* and discard all test cases where the *p* was found to be true.
3. Repeat steps 1 and 2 until the set of test cases is empty or the set of predicates is empty.

5. CONCLUSION

SMEs can struggle for decades with separating valid, but unexpected, exploratory software outcomes from failures. This remains an open probelm. However, we have developed a predicate-based statistical debugging method, ESP, which localizes sources of unexpected outcomes. ESP replaces the uniform and static predicates used in existing approaches with elastic predicates. The result is improved effectiveness. ESP outperforms the best alternatives in exploratory software applications and performs as well as the best alternative for traditionally targeted software. In future work, we will explore adaptive sampling approaches to reduce the program points instrumented with elastic predicates in ESP and automate the selection of parameters within ESP's elastic predicates.

6. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. of the ACM SIGPLAN 1990 Conf. on Programming Language Design and Implementation*, 246-256, 1990.

[2] P. Arumuga Nainar and B. Liblit. Adaptive bug isolation. In *Proc. of the 32nd Int. Conf. on Software Engineering*, 255-264, 2010.

[3] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound Boolean predicates. In *Proc. of the 2007 Int. Symp. on Software Testing and Analysis (ISSTA '07)*, 5–15, 2007.

[4] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix and W. Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5): 22-29.

[5] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Proc. of 31st Int. Conf. on Software Engineering*, 34–44, 2009.

[6] H. Cleve and A. Zeller. Locating causes of program failures. In *Proc. of the 27th Int. Conf. on Software Engineering*, 342-351, 2005.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Software Engineering*, 27(2): 99–123, 2001.

[8] L. Fortnow. The status of the P vs. NP problem. *Communications of ACM* 52(9): 78-86, 2009.

[9] M. K. Govil and M. C. Fu. Queueing Theory in Manufacturing: A Survey. *Journal of Manufacturing Systems* 18(3): 214-240, 1999.

[10] M. J. Harrold, A. J. Offutt and K. Twewary. An approach to fault modeling and fault seeding using the program dependence graph. *Journal of Systems and Software* 36(3): 273-295, 1997.

[11] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. In *Proc. of the 2004 Fundamental Approaches to Software Engineering Conf.*, 267-280, 2004.

[12] D. Jeffery, N. Gupta and R. Gupta. Fault Localization Using Value Replacement. In *Proc. of the 2008 Int. Symp. on Software Testing and Analysis*, 167–177, 2008.

[13] D. Jeffery, N. Gupta and R. Gupta. Effective and efficient localization of multiple faults using value replacement. In *Proc. of the 2009 IEEE Int. Conf. on Software Maintenance*, 221-230, 2009.

[14] L. Jiang and Z. Su. Profile-guided program simplification for effective testing and analysis. In *Proc. of the 16th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, 48-58, 2008.

[15] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proc. of the 20th IEEE/ACM Int. Conf. on Automated Software Engineering*, 273–282, 2005.

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

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

[18] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In: *Proc. of the ACM SIGPLAN 2003 Conf. on Programming language Design and Implementation* (PLDI '03), 141–154, 2003.

[19] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation*, 15–26, 2005.

[20] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical model-based bug localization. In *Proc. of the 13th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, 286–295, 2005.

[21] ns-2: Java Network Simulator. http://jns.sourceforge.net

[22] B. Parhami. Defect, Fault, Error, …, or Failure. *IEEE Trans. on Reliability*, 46(4): 450–451.

[23] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proc. of the 18th IEEE/ACM Int. Conf. on Automated Software Engineering*, 30–39, 2003.

[24] SimJ Modeling Tools. http://jmt.sourceforge.net.

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

[26] StochSiemens. http://cs.virginia.edu/~rjg7v/stochsiemens/.

[27] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3): 121-189, 1995.

[28] W. Weimer, S. Forrest, C. L. Goues, and T. Nguyen, Automatic program repair with evolutionary computation. *Communications of the ACM* 53(5): 109-116, 2010.

[29] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4): 352-357,1984.

[30] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *Proc. of the ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, 169-180, 2006.

[31] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proc. of the 28th Int. Conf. on Software Engineering*, 272-281, 2006.

[32] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proc. of the 23rd Int. Conf. on Machine Learning*, 1105–1112, 2006.