Statistically Debugging Massively-Parallel Applications

Tristan Ravitch and Ben Liblit
Computer Sciences Department, University of Wisconsin–Madison
{travitch,liblit}@cs.wisc.edu

Bronis R. de Supinski
Lawrence Livermore National Lab
bronis@llnl.gov

Abstract—**Statistical debugging identifies program behaviors that are highly correlated with failures. Traditionally, this approach has been applied to desktop software on which it is effective in identifying the causes that underlie several difficult classes of bugs including: memory corruption, non-deterministic bugs, and bugs with multiple temporally-distant triggers.**

The domain of scientific computing offers a new target for this type of debugging. Scientific code is run at massive scales offering massive quantities of statistical feedback data. Data collection can scale well because it requires no communication between compute nodes. Unfortunately, existing statistical debugging techniques impose run-time overhead that is unsuitable for computationally-intensive code despite being modest and acceptable in desktop software. Additionally, the normal communication that occurs between nodes in parallel jobs violates a key assumption of statistical independence in existing statistical models.

We report on our experience bringing statistical debugging to the domain of scientific computing. We present techniques to reduce the run-time overhead of the required instrumentation by up to 25% over prior work, along with challenges related to data collection. We also discuss case studies looking at real bugs in ParaDiS and BOUT++, as well as some manually-seeded bugs. We demonstrate that the loss of statistical independence between runs is not a problem in practice.

I. Introduction

Statistical debugging methods instrument programs to record program behaviors at run time, along with a success or failure label. An offline statistical analysis then identifies the program behaviors that are highly correlated with program failure. These techniques have most often been used after an application, typically desktop oriented, is deployed to many users. Recorded program behaviors are referred to as *feedback data* and are reported across many independent runs of the application.

This approach to debugging has been demonstrated to be able to identify the causes of difficult classes of bugs as varied as input validation, bad comment handling, unchecked return values, inconsistent data structure coordination, buffer overruns (both with and without memory writes), configuration-sensitive hash table mismanagement, memory exhaustion, premature returns, poor error-path handling, race conditions, and dangling pointers [1], [2], [3], [4]. Further, it is not restricted to bugs that crash the program; the analysis requires only success and failure labels, so performance and correctness bugs are equally amenable to diagnosis via statistical debugging.

Scientific computing represents a tantalizing new arena in which to apply statistical debugging techniques, but carries some unique challenges. Unlike desktop software, scientific applications are rarely, if ever, finished. Any given version of a scientific application is run very few times by few users, rather than many times by many users as for widely-deployed desktop applications. Fortunately, scientific applications are often run at a large scale, with many individual processes participating. If we treat each process as a program run in the traditional sense, we can gather large quantities of feedback data for analysis in the few runs available. Unfortunately, because the processes of a run communicate, they are no longer independent, which violates one of the key assumptions underlying most of the statistical models used in feedback analysis.

Performance represents another challenge. The instrumentation used to observe program behavior at run time is generally lightweight and works well for desktop applications. This instrumentation fares worse in the presence of the tight loops that are common to computationally-intensive code. However, if we can overcome this problem then we expect statistical debugging itself will scale well, as it requires no extra communication between compute processes.

In this paper, we address the issues outlined above and demonstrate the applicability of statistical debugging to scientific computing applications. Our main contributions are:

- Instrumentation transformations to eliminate up to 25% of instrumentation overhead;
- An efficient feedback-collection strategy that can capture results from 500,000 processes in under 50MB;
- Support for C++ applications; and
- Case studies of real bugs in ParaDiS and BOUT++, as well as manually-seeded bugs, that demonstrate that statistical models employed by statistical debugging can effectively determine the root causes of programming errors in scientific applications.

In Section III, we describe transformations and techniques to reduce the overhead imposed by instrumentation in computationally-intensive code. We show how to collect feedback data efficiently from completed jobs in Section IV. Section V discusses our implementation of these ideas in a new instrumenting compiler. We evaluate the effectiveness of these techniques in Section VI, and describe our experiences with diagnosing several real and seeded bugs in Section VII. Section VIII addresses related work and Section IX concludes.

II. Statistical Debugging Background

So far we have described statistical debugging as identifying program behaviors that are highly correlated with failure. We formalize the notion of program behavior in terms of *predicates* that are true or false at certain program points. At any given program point, multiple predicates may exist. Consider an **int** variable: at any given program point its sign can be either negative, zero, or positive. An *instrumentation site* is a program point instrumented to observe at least one predicate.

The statistical debugging process begins by instrumenting at the source or binary level to observe and to record predicate values. Next, users run the instrumented program. Each run generates a *feedback report* that captures the true and false counts of each observed predicate, plus a success or failure label for the entire run. Finally, we analyze these reports statistically to identify predicates that correlate highly with failed runs.

A. Instrumenting the Code

Programs have many potential predicates. Prior work focused on efficiently-testable predicates [3], [5]. Examples include:

- The branch taken in a conditional;
- The signs of integral function return values; and
- The classification of floating-point values (NaN, infinity, denormal, or normal).

Although these tests are efficient, recording the values of these predicates every time they are encountered can be costly. Liblit et al. [4] describe a method of *sampling* instrumentation sites based on work by Arnold and Ryder [6]. Sampling treats the occurrences of sites in a program execution as a stream of events and uses a Bernoulli process to select a random subset to record. Randomness ensures that periodicity inherent to a program cannot cause the algorithm always to choose the same sites. The Bernoulli process can be thought of as a flip of a biased coin at each site reached by the program: if the coin comes up as heads, the site is sampled, recording true and false observations of its constituent predicates. This mechanism ensures that any site in the execution is equally likely to be sampled.

Directly implementing a Bernoulli process as a series of coin flips reduces performance even more than observing every predicate at every site. A more efficient formulation considers the inter-arrival times between observations. Equivalently, we treat this inter-arrival time as a *countdown* of sites to ignore before sampling. We model this quantity by a geometric distribution parametrized by the desired average sampling rate.

In terms of implementation, we could decrement the countdown every time that a site executes. When the countdown reaches zero, we observe the associated site's predicates and draw a new random countdown from a geometric distribution. This approach still suffers from too much overhead; to reduce the number of countdown tests against zero, we segment the control-flow graph of a program into acyclic regions. Each acyclic region has a finite and known maximum number of sites along each possible path, known as the *weight* of the region. At the beginning of any acyclic region there are two possibilities: if $\mathit{countdown} > \mathit{weight}$, we know that we will not have to sample in the acyclic region. Otherwise, we may have to take a sample and must check at each decrement. To exploit this observation, we can clone each acyclic region and remove all checks of the countdown against zero in one copy. This copy is called the *fast path*; it still retains its countdown decrements, but these operations typically disappear into instruction-level parallelism. The other copy is the *instrumented path*.

B. Statistical Analysis

We introduce no new statistical methods over previous work; we cover several formulations of the statistical analysis in Section VIII. We review one model [7] that has been demonstrated to work well in a variety of debugging contexts. We choose this model because its relatively simple mathematical foundations scale well to large data sets, such as those we expect to generate from supercomputing applications.

Predicate counts are meaningless alone; we must relate them to failure and success labels. To grasp the intuition of the analysis, consider how predicates can identify deterministic bugs. Any predicate P is a perfect predictor of failure if it is always true in every failing run and never true in a correct run.

Unfortunately, many bugs, such as buffer overflows in C and C++, are not so deterministic. Even when a buffer overflows, execution might not necessarily fail: the overwritten memory may have been unused. Thus, a predicate that exactly describes when the buffer overflow occurs can be true in some successful runs. While the program technically misbehaved in those runs, it terminated without exhibiting any overtly-incorrect behavior.

A statistical model of failure can overcome this type of noisy data by giving a probability that P being true leads to failure. The most basic measure of failure-predictive power is

$$\mathit{Failure}(P) \equiv \Pr(\text{failed run} \mid P \text{ observed to be true})$$

Failure (P) is high when P being observed to be true is highly correlated with program failure. *Failure* is not a sufficient metric for identifying failure causes: it is far too aggressive and many predicates on the path between the root error and the program failure could receive high *Failure* scores.

We can trim incidental predicates that just happen to hold on a failing path by considering whether we ever observe a predicate (regardless of whether it was true or false) in a non-failing run. We first define an intermediate quantity:

$$\mathit{Context}(P) \equiv \Pr(\text{failed run} \mid P \text{ observed})$$

Context is high for predicates rarely observed in correct runs. We now define a more direct measure of failure-predictivity:

$$\mathit{Increase}(P) \equiv \mathit{Failure}(P) - \mathit{Context}(P)$$

$\mathit{Increase}(P)$ is high if program failure is much more likely when P is observed to be true, contrasted with always being likely to execute the code associated with P. Thus, the model tends to find predictors near the root cause of failure instead of ones simply on the path from a bug to an observable failure.

We can discard predicates with an *Increase* score below a threshold. We then must rank the remaining predicates by

```
1 for (i = 0; i < vec_len; i += STRIDE) {
2   if (countdown > WEIGHT) {
3     // Fast path
4   } else {
5     // Instrumented path
6   }
7 }
```

Listing 1. A simple sampled loop

```
1  i = 0;
2  while (i < vec_len) {
3    int loop_start = i;
4    int bound = vec_len;
5    if (countdown <= (bound − i) / STRIDE * WEIGHT)
6      bound = i + (countdown − 1) / STRIDE * WEIGHT;
7    for (; i < bound; i += STRIDE) {
8      // Completely uninstrumented path
9    }
10   countdown −= (i − loop_start) / STRIDE * WEIGHT;
11   if (i < vec_len) {
12     // Instrumented path
13   }
14   i += STRIDE;
15 }
```

Listing 2. Optimized variant of listing 1

their predictive value, which we will call *Importance*. Ranking predicates by the number of failing runs in which they appear true boosts the scores of predicates that are weakly correlated with many failures. Alternatively, using the *Increase* score directly favors predicates that are rarely true but almost always lead to failure. A common technique to combine multi-modal ranking inputs is to use the harmonic mean:

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

where NumF is the number of failed runs and $F(P)$ is the number of failed runs in which P holds. This ranking favors predicates that are both predictive of failure (high specificity) and appear to explain many failures (high sensitivity).

III. Sampling Scientific Workloads

The sampling procedure described in Section II-A works well for typical interactive desktop applications, which spend most of their time waiting for user input. However, scientific workloads are CPU-bound and spend most of their time in loops that perform numeric computations. The logic to choose between the fast or instrumented path is executed once per acyclic path, and therefore, once per loop iteration. This frequency imposes a significant overhead, especially for short loops. We present an optimization to the sampling transformation to reduce sampling overhead substantially for most numeric loops.

A. Sampling Optimizations for Loops

Listing 1 shows a normal loop after the sampling transformation. The path check on line 2 is based on the countdown and the two copies of the loop body: one instrumented and the other with only countdown decrements (the fast path). The WEIGHT constant is, again, the maximum number of instrumentation sites in any path through the loop body.

For small loops, the weight of the loop body is much less than the countdown. Further, we waste many checks that simply lead to choosing the fast path. Our optimization amortizes the cost of the path check over as many iterations as possible. If the loop meets the conditions that we discuss in Section III-A1 then we can precisely bound the number of loop iterations that can execute before we must check countdown again.

We rewrite the loop in three parts. First, it executes without any instrumentation, not even countdown decrements, up to the computed bound. Next, since the optimized loop body has no countdown decrements, we must decrement the countdown by the total number of executed instrumentation sites. Finally, execution enters a fully instrumented version of the loop body in which we take a sample. We wrap these two steps inside of a driving loop to repeat the process as often as necessary to reach the total iteration count.

Listing 2 shows the optimized form of the previous code example. The induction variable changes by an amount equal to the constant STRIDE each iteration. Line 2 shows the driving loop that ensures we execute the correct total iteration count. We compute the bound in lines 4 to 6, and determine the number of consecutive uninstrumented loop body iterations in line 7. We decrement the countdown in line 10 by the number of executed instrumentation sites. The additional check in line 11 ensures that we do not execute the loop body an extra time when the fully uninstrumented path includes the last iteration.

The optimization generalizes to support nested loops if each loop meets the transformation requirements. In practice we can rarely apply the optimization to loops nested more than three deep. More complicated nested looping constructs are rare and typically contain other violations of the conditions in Section III-A1. Further, doubly-nested loops usually maximize the performance benefits of the optimization due to the complexity of the loop bound calculation for more deeply-nested structures and limits imposed by the sampling rate.

1) Conditions on the Loop Body: Loop bodies must satisfy two high-level requirements in order to qualify for the loop optimization: the weight of the loop body must be finite and the iteration count must be symbolically expressible. Further, all paths through the loop body must have the same weight and must not have control-flow–altering constructs such as **break**. In principle, this restriction is unnecessary; we can insert dummy instrumentation sites to balance all paths through a loop. In practice, loops with unbalanced instrumentation counts usually fail to meet the finite-weight requirement and do not benefit from the path balancing. The dummy instrumentation sites are undesirable, particularly in loop bodies, because they consume randomness without the chance to provide useful feedback data. Thus, we do not use them.

In order for the number of loop iterations to be symbolically expressible, the following conditions must hold:

- The loop body must not modify the induction variable;
- The loop body must not modify the iteration count bound;
- The stride must be constant;
- The loop condition must be idempotent; and
- The induction variable must be local.

We require idempotence because the transformation duplicates the evaluation of the loop upper bound. With nested loops, loop upper bounds and initial induction variable values must not depend on variables defined in enclosing loops.

B. *Non-uniform Sampling Rates*

The preceding optimization effectively amortizes path checks in numeric loops over many iterations. Unfortunately, useful sampling rates are about $1/100$, which limits the scope of the amortization. By definition the transformation applies to loops that are computational *leaves* and cannot call functions with side effects. The instrumentation sites in these loops are usually floating-point operations that are less interesting for debugging than other operations. Further, they occur frequently, with event counts reaching hundreds of millions.

We leverage the nature of these loops by dynamically reducing the sampling rate for their duration, in the style of Hauswirth and Chilimbi [8]. This reduction, in turn, magnifies the benefit of the loop-splitting optimization. Each optimized loop runs in a learning mode in which we discover how many iterations it executes during its first execution. Each time the loop completes a set of uninstrumented iterations, we reduce the sampling rate by a factor of 10, with a minimum sample rate controlled by an environment variable. This mechanism exponentially decays the sample count in each loop based on the size of its inputs. The new sampling rate for the loop is memoized and re-used in future loop executions.

C. *Revisiting Numeric Loops*

The loop-splitting transformation optimizes a class of numerically- and computationally-intensive loops. The information that we obtain by instrumenting these loops has little diagnostic value for many bug classes. Consider a vector normalization function. The loop termination condition contributes two predicates since it can be true or false on each iteration. However, these predicates are often redundant: if the loop executes at least once, predicates that precede the loop imply that we could observe the loop condition as true. Similarly, predicates after the loop imply that we could observe the loop condition as false. If the loop never terminates, we will not observe any predicates after the loop, yielding approximately equivalent results. When diagnosing these bug types, we could omit the instrumentation from these loops.

IV. Data Collection

Prior work, such as that of Liblit et al. [9], relies on the instrumented application to report its own feedback data by writing to a file. This approach is a scalability barrier due to I/O pressure. Also, reporting feedback data from a failing process requires handling POSIX signals to catch events like segmentation faults. Performing complex tasks in signal handlers is unwise in the best of times; when the process is failing and in an unsteady state, it is even riskier. To address these problems, we move the reporting infrastructure from the instrumented process to an external *watchdog*, and propagate feedback data to a reporting node using MRNet [10].

A. *Reporting Machinery*

The watchdog process uses the Dyninst framework [11] to monitor instrumented processes for termination, abnormal or otherwise. There is one watchdog per physical compute node. Each of these watchdogs attaches to all of the application processes local to its node via the debug interface (ptrace). To communicate feedback data efficiently from an instrumented process to the watchdog, we employ a shared memory segment visible to both processes. The instrumented process stores its feedback data within the shared memory segment. When the instrumented process terminates, the watchdog simply reads the feedback data out of the shared memory segment. In the event of an abnormal termination, the watchdog also captures a stack trace.

Besides efficiency, the shared memory segment is significantly more robust than in-process reporting. With in-process reporting, heap corruption could easily render the instrumented application unable to produce a report or, worse, could cause it to produce a seemingly-valid report with hidden corruption. Standard library components such as I/O buffers or file descriptors are also vulnerable. By contrast, when using a shared memory segment, only the small area occupied by that segment is exposed to corruption. Relative to the entire address space, this vulnerability surface is much smaller. Even in the face of extreme termination measures such as SIGKILL, a shared memory segment still allows feedback to be captured, whereas in-process reporting does not.

After the watchdog collects all available feedback reports, it sends the data to a reporting node via MRNet, a scalable Multicast/Reduction Network. MRNet provides a tree-structured communication network in which our watchdog processes form leaves, or MRNet *backends*. We propagate data up the tree to a *frontend* that writes the feedback data to disk.

Once feedback data enters the MRNet tree, it passes through *filter functions* at each level of the tree until it reaches the frontend. These filters can transform the data as it propagates; we use them to compress samples losslessly. Each watchdog process sends its feedback data into the communication tree uncompressed and the first layer compresses it with a standard compression algorithm. Further levels in the tree concatenate the data that they receive. This approach allows the compression algorithm to exploit a large window of redundancy across more feedback reports than are available to a single watchdog.

Figure 1 compares simulated data collections for several compression strategies. First, compressing feedback reports only on the watchdogs is fast because the work is distributed

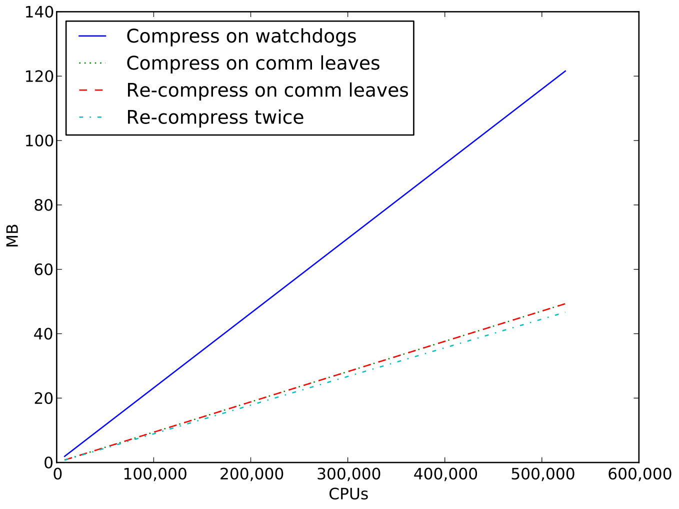

Fig. 1. Feedback data sizes

across more nodes. This approach also imposes more than 50% overhead in the size of the collected data. Alternatively, the watchdog processes compress their feedback reports, which are then decompressed and recompressed by a tree node. This approach yields the same compression ratio as our strategy with a minor time penalty. Another variant repeats this recompression higher in the tree to increase space savings slightly at the cost of two orders of magnitude longer completion time.

B. Data Format

Feedback reports are ordered integer tuples. Each tuple represents an instrumentation site and each tuple entry denotes the number of times that the individual predicate was observed to be true at that site. Many predicates are never observed to be true, or are observed to be true only a few times. Alternatively, we can easily observe predicates in nested loops hundreds of millions of times. This range of data benefits from the implicitly variable-length encoding afforded by plain text; however, textual formats waste space for delimiters and to represent large numbers. Alternatively, we use the standard Abstract Syntax Notation (ASN.1), which is a binary encoding with variable-length integer representations.

A given program run never reaches many instrumentation sites. We use a *sparse* representation to reduce the space overhead of unreached code. Each feedback report has an associated bitmask and represents tuples that contain all zeros by a zero in the bitmask. We represent all tuples that contain data by a one and reconstruct the full sequence of tuples at analysis-time. The sparse ASN.1 format has shown space savings of 35% to nearly 700% in report encodings.

Our simulated data collections in Figure 1 use feedback reports generated by IRS. We use the sparse ASN.1 representation and compress with bzip2. We also randomly perturb the reports to prevent the compression algorithm from achieving trivial best-case behavior. This simulation indicates that data format and communication infrastructure can collect reports from 500,000 processes in less than 50MB.

V. Implementation

We implement a new instrumenting compiler as a source-to-source translator using the ROSE compiler infrastructure [12]. To coordinate our watchdog processes with the applications that we debug, we start them simultaneously using LaunchMON [13]. These watchdogs monitor applications via the Dyninst StackwalkerAPI [11] and report results using MRNet [10]. We leverage the OpenMP, C++, and Fortran support in ROSE to handle a larger set of applications than previous source-based instrumenting compilers. This support is particularly important for scientific applications, many of which use some features from (or components written in) these languages.

Three C++ language features complicate our instrumentation mechanism: (1) reference types, (2) objects with user-defined constructors or destructors (known as non-Plain Old Data, or *non-POD*, types), and (3) **try** blocks. These features cause problems for the same underlying reason: they inhibit jumping between fast and instrumented paths. At the beginning of each acyclic region, execution can either stay on its current path or jump to the other one, depending on the value of the countdown. After this jump, the same variables must be in scope with the same values, which is easily facilitated in C by lifting all variable declarations, with proper renaming, to the top of a function body. We also instrument C++ code that does not use any of the complicating features in this way.

Reference types are complicated because we must initialize them and they cannot refer to another object after initialization. If we lift the declaration of the reference to the top of the function, its initializer may not be in scope. Thus, we rewrite reference-typed variables as pointer-typed variables and make previously implicit dereferences explicit.

Non-POD declarations also pose a scoping problem: if we move or duplicate these declarations then we also move or duplicate the side effects of their constructors or destructors. Also, execution cannot jump past the declaration of non-POD objects. We do not move non-POD declarations; instead we recursively treat the code that they dominate as a new function when we create fast and instrumented paths. Effectively, we split the code around them. In principle, this technique could apply to all variable declarations. However, it reduces performance by making fast code paths shorter. Therefore, we apply this transformation only for non-POD variables. We continue to lift POD variables up to the top-level scope in each function.

C++ **try** blocks introduce a similar difficulty to non-POD declarations: execution cannot jump from the middle of one **try** block into the middle of another. Thus, we treat **try** blocks similarly to non-POD variable declarations: we never clone them, but instead recursively treat **try** block bodies as though they were the entry points of new functions for purposes of fast versus instrumented path creation.

VI. Experimental Evaluation

We evaluate several aspects of the performance overhead of our data collection infrastructure:

1) the overhead imposed on serial codes,

2) the overhead imposed on parallel codes, and
3) the additional communication overhead.

We run all experiments on 4-way dual-core 2.4GHz AMD Opterons with 16GB of RAM with an InfiniBand interconnect. Where applicable, we compile all benchmark applications in several configurations using

- GCC as a performance baseline,
- the instrumenting compiler from Liblit et al. [7], and
- our new ROSE-based instrumenting compiler.

The instrumenting compiler of Liblit et al. serves as a performance degradation baseline. While we use a different instrumentation infrastructure, comparison to this baseline demonstrates the improvements that our techniques provide for scientific applications. We use our instrumenting compiler in three modes: (1) with no optimizations, (2) with just the loop splitting optimization, and (3) with the loop splitting optimization and adaptive sampling rates for numeric loops. We also explore the performance benefits of omitting instrumentation from computational kernels.

A. Loop Optimization Impact

First, we evaluate the run-time overhead of our sampled instrumentation on serial program execution. We compile these benchmarks without MPI support and run them with a single thread. All instrumented runs sample the same set of sites, monitoring the directions of branches and for the signs of function return values. All runs in this section use a sampling rate of $1/100$.

The overheads in this section are presented as run times of instrumented binaries compared to a baseline version compiled with GCC. The runs labeled as cil are instrumented using the instrumenting compiler of Liblit et al. [7]. We present results for four different configurations of our instrumenting compiler:

- rose-baseline is a mode with no optimizations beyond the sampling transformation discussed in Section II-A, roughly equivalent to the transformation applied by the *cil* runs;
- rose-opt utilizes the loop-splitting transformation described in Section III-A up to doubly-nested loops;
- rose-opt-adapt is rose-opt plus adaptive reduction of the sampling rate for simple numerical loops as discussed in Section III-B; and
- rose-no-numeric removes all instrumentation from numeric loops, as in Section III-C.

The graph in Figure 2 shows runs of AMG on the default inputs using solver 5 and problem 2 at problem sizes ranging from 150^3 to 300^3. As expected, the loop-splitting transformation benefits greatly from adaptively modifying the sampling rate for numeric loops. The combination reduces overheads compared to previous work by 5-20%. Further, adaptively modifying the sampling rate for numeric loops achieves similar efficiency to not instrumenting them at all.

Figure 3 shows runs of IRS with 8, 27, 64, 125, and 216 domains. The version using an adaptive sampling rate averages about a 4% improvement over the cil baseline. For non-trivial problem sizes, the absolute overhead imposed compared to GCC is less than 12%. Not instrumenting numeric loops at all yields a modest improvement of 2% on top of the adaptive sampling rate approach.

The runs of ParaDiS, shown in Figure 4, use the default problem (fmm_8cpu.ctrl) modified to run on a single processor and to use 4^3, 8^3, and 16^3 cells with 20 time steps. The adaptively-sampled version cuts the overhead imposed by the cil baseline by over 40%, with approximately 10% overhead compared to GCC.

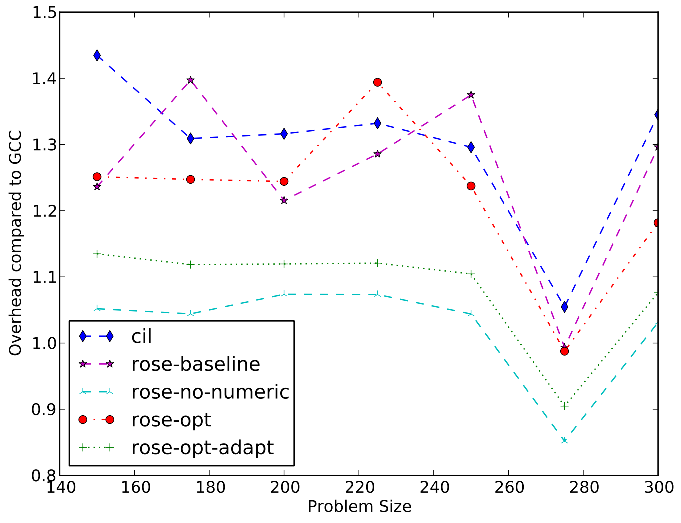

Fig. 2. Serial overhead for AMG

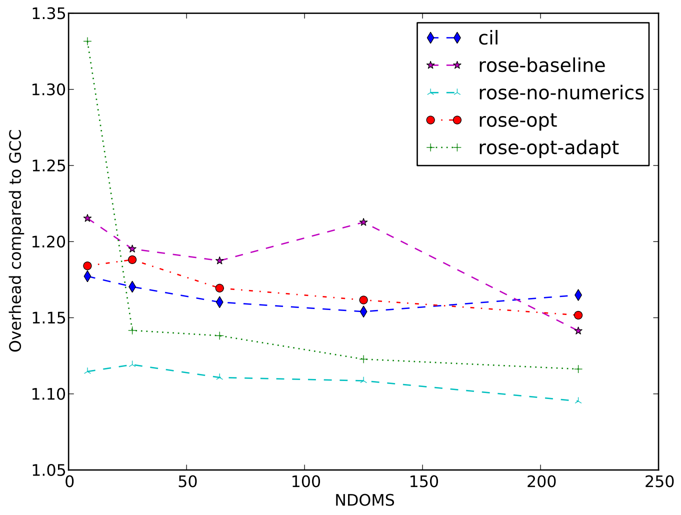

Fig. 3. Serial overhead for IRS

TABLE I
APPLICABILITY OF THE LOOP SPLITTING TRANSFORMATION

Benchmark	Total Loops	Split Loops	Fraction Split
AMG	2,859	1,508	52%
IRS	2,566	1,188	46%
ParaDiS	1,785	702	39%

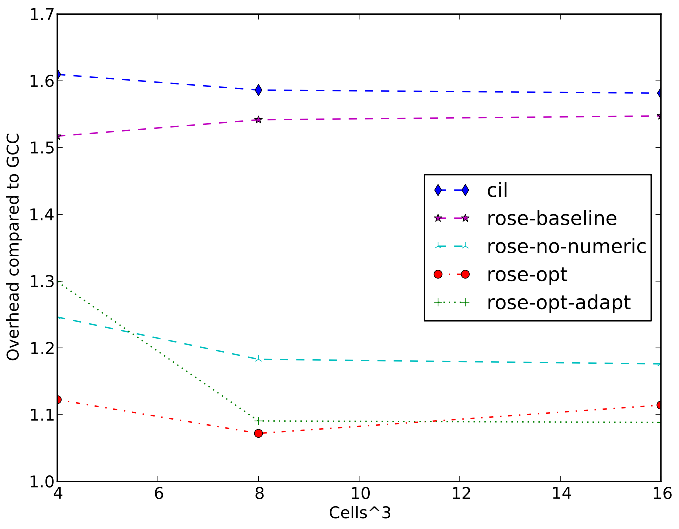

Fig. 4. Serial overhead for ParaDiS

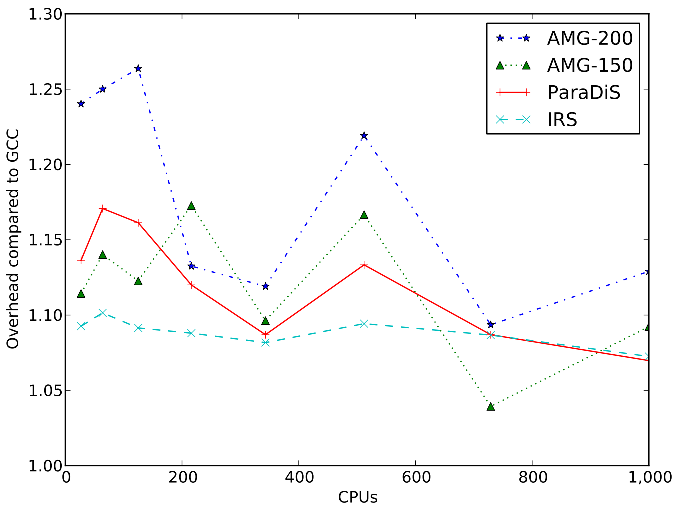

Fig. 5. Overhead versus GCC in parallel executions

Despite the many conditions restricting its applicability, our loop splitting transformation effectively reduces instrumentation overhead. Table I shows the number of times the optimization was applied in each benchmark. Overall, about 45% of loops in these benchmarks are amenable to the transformation.

B. *Parallel Execution Overhead*

We also examine the overhead imposed by our instrumentation on the execution of parallel programs. This analysis includes both computational overhead due to instrumentation and communication overhead contributed by random sampling. Figure 5 compares the execution times of AMG (with per-processor problem sizes of 150^3 and 200^3), IRS, and ParaDiS against baseline versions compiled with GCC. Each binary is instrumented to observe the directions of branches and the signs of function return values. All instrumented binaries are subject to our loop optimization and adaptive sampling technique.

We run each application in each problem-size/CPU configuration four times. The reported numbers are the overhead due to instrumentation: the ratio of the mean run-times of our instrumented binaries versus uninstrumented binaries. We use a sampling rate of $1/100$, as in our previous experiments. Observed overheads are between 10% and 15% for IRS and ParaDiS, closely tracking the serial overheads reported previously. AMG also falls largely within this range, but with spikes at some problem sizes up to 25% overhead. The spike in the AMG results for the 512 CPU configuration is due to some characteristic of the input. Both with and without instrumentation, that configuration takes significantly less time to execute than most other problem sizes. The short running time provides little opportunity for instrumentation costs to be amortized, resulting in a higher relative overhead.

These overheads do not include the time required to report feedback data to the frontend; we separate them to simplify the analysis of the impact of our instrumentation. Our experimental configuration compresses feedback reports only in the leaf nodes of the communication tree. No runs took more than 5 seconds to finish reporting to the frontend, including network communication costs.

C. *Communication Overhead*

Finally, we attempt to separate the overhead that our instrumentation imposes on inter-process communication from computational overhead. Since the instrumentation samples randomly, nodes that choose to take many samples may reach communication points more slowly than those that take few. Thus, although our instrumentation adds no direct inter-process communication, it can randomly perturb the execution time on each node, leading to additional communication delays. To measure this effect, we instrument MPI collective operations in AMG.

In runs with 1,024 processes with problem size 150^3, we record the time each process spends waiting in collective operations at each dynamic call-site using the mpiP profiling library. We repeat this 40 times each for uninstrumented and instrumented binaries. About 15% of (process, dynamic call) pairs exhibit an increase in time spent waiting on collective operations, as determined by a t-test with 95% confidence. The affected (process, dynamic call) pairs span 43% (281 out of 652) of all dynamic calls. The increase in wait time averaged 56% with a standard deviation of 85 percentage points. The median increase was 46%. Taking the sum of the maximum wait time increases for each dynamic call as an over-approximation, communication overhead accounts for at most 13% of observed overhead in this configuration.

We repeated the experiment with the same number of processors, but at problem size 200^3. The same percentage of (process, dynamic call) pairs exhibit a significant increase in collective wait times. By the conservative metric above, communication delays account for at most 30% of the overhead observed in this configuration. Subtracting these estimated communication overheads from the runs at problem sizes 150^3 and 200^3 leaves both configurations at approximately the same overhead.

TABLE II
FAILURE PREDICTORS FOR PARADIS

Predicate	Function
i < home−>newNodeKeyPtr	SortNativeNodes
inode < home−>newNodeKeyPtr	MonopoleCellCharge
tag.domID == home−>myDom	GetNodeFromTag
cycleEnd == 0	DD3dStep
iNbr > nXcells	InitCellNeighbors
remDom == 0	GetNodeFromTag
node != 0	CommPackGhosts

Based on these observations, we claim that communication overhead explains the difference between the serial overheads in Section VI-A and the parallel overheads in Section VI-B. The serial overheads were approximately the same for most problem sizes, while the parallel runs showed additional overhead for larger problem sizes. Intuitively, the larger problem sizes allow each process to run independently for longer without communication. These longer periods between communication points allow the processes to diverge in the number of instrumentation sites randomly sampled. This divergence manifests as increased overhead due, in part, to time spent waiting for collective operations to complete.

VII. Root-Cause Analysis of Application Bugs

A. *ParaDiS 2.0*

We have applied our techniques to ParaDiS [14], a dislocation dynamics simulator. Version 2.0 of this code suffers from a bug that causes it to crash on most of its inputs. We instrument the code to sample predicates on branches taken and function return values. After finding a working input, we apply our analysis to several crashing runs and a few runs on the successful input; the analysis identifies the predicates in Table II as significant failure predictors.

This code divides the problem space, and hence dislocations, into *domains* that are distributed among compute nodes. ParaDiS refers to dislocations as "nodes" internally, particularly in function names; except for direct references to ParaDiS functions with "node" in the name, we use the term to refer only to compute nodes in a cluster. Each domain is divided into cells and is responsible for a set of *native* dislocations; non-native dislocations are represented as *ghosts*. At each step in the computation, each compute node

1) *migrates* ownership of dislocations that cross the boundary of its domain to appropriate neighbors,
2) organizes its remaining native dislocations into cells,
3) sends updates of ghost information to neighboring nodes, and
4) computes the local effects of forces on dislocations.

The bug manifests as a segmentation fault in the OrderNodes function. This function is called by MonopoleCellCharge in the loop controlled by the predicate identified by our analysis. We can explain the bug by working backwards in the call graph from this point of failure. The nearest failure predictor is remDom == 0 evaluating to true in GetNodeFromTag. This returns a NULL pointer, which eventually causes the segmentation fault.

Temporally, the next preceding predictors are in CommPackGhosts and SortNativeNodes, which update ghost dislocations in neighboring domains and divide local dislocations into cells, respectively. Both of these predictors arise because they are executed more frequently in failing runs, thus appearing in fewer successful runs. This suggests a correlation between failure and runs with many dislocations.

Temporally, the next nearest predictor is in InitCellNeighbors. This function is called before the first time step. Note that each step of the computation begins by migrating some dislocations to neighboring nodes. From here, we hypothesize that the crash arises because dislocation ownership is not tracked correctly. This leaves nodes with an inconsistent view of the dislocations owned by their neighbors after the first migration. MonopoleCellCharge causes the crash when it attempts to inspect nonexistent dislocations on a neighbor node. The cases in which the application succeeds are those with few dislocations which happen to not fall near cell boundaries. Inspection of the next ParaDiS release (2.2) shows that the code tracking dislocations eligible for migration and ghosts has been completely rewritten, suggesting that this was indeed a significant contributor to the underlying bug.

The predictor trace offers significant detail not available from backtraces. Backtraces can only show the state of the stack when a problem occurs; predictor traces can link together events from different branches of the run-time call graph. In this case, the backtrace would not include GetNodeFromTag, InitCellNeighbors, SortNativeNodes, or CommPackGhosts, which are essential to our understanding of the actual root cause.

B. *BOUT++*

We have also applied our tools to a bug in version 0.7 of BOUT++ [15], a fluid dynamics simulator. The bug manifests as non-deterministic output; in particular, the right-hand side of an equation is evaluated different numbers of times for identical inputs. Our tools require some program runs to be successful and others to fail, but in this situation the correct output for a given input is not obvious to non-experts in the domain of fluid dynamics.

To fit this bug into our model, we cluster runs into two groups: one in which the outputs are similar up to some common prefix, and the rest. We treat the runs with output sharing a common prefix as successful, and the other runs as failing. The top three failure-predicting predicates returned by the analysis strongly indicate that uninitialized memory is used in the ModifiedGS function. This is consistent with the observed non-deterministic behavior. The inputs to the program are completely deterministic, so the randomized contents of uninitialized memory could explain the non-deterministic behavior.

Specialized tools can be extremely helpful if the nature of the bug is known or suspected. Although our analysis is not specifically focused on finding uninitialized-memory bugs, it actually performs quite well at this task. As a comparison to other debugging tools, our analysis for this bug exhibits diagnostic power approximately equivalent to that of Valgrind's memcheck [16], a much more specialized, narrowly-focused tool.

C. *Manually-Seeded Bugs*

To test the effectiveness of statistical debugging techniques on a wider range of bugs, we performed a blind experiment in which a colleague inserted some common types of bugs into AMG and ParaDiS. We then applied statistical debugging techniques to find these bugs, as in the case of the real bugs above. The types of bugs that could be inserted are limited by two factors:

- we have only a few inputs to these programs, and
- we require some program runs to be successful in addition to the failing runs.

Nevertheless, we report our experiences. The first bug was seeded into AMG. When run on this buggy version of the code, the analysis identified a single strong predicate. This predicate was within a function called directly by the function containing the actual bug. The bug, a pair of swapped dimensions, was easily apparent by tracing the data dependencies backwards.

The next bug was also seeded into AMG. The analysis returned three strong predicates in the second major setup phase of the application. Two of the three predicates were closely related and indicated that failure was highly correlated with a particular intermediate value being large. This intermediate value was extracted from a matrix constructed in the first setup phase: the construction of a Laplacian matrix. The bug was indeed in the code constructing this matrix; however, we did not find it ourselves due to a lack of familiarity with the algorithm being implemented. Someone familiar with the algorithm would have seen that an input matrix was traversed in the wrong order.

The third bug was seeded into ParaDiS, and the analysis again returned three strong predicates. The execution of ParaDiS proceeds in steps, with the same sequence of calls repeated each time step. These predicates pointed to two calls that occur in sequence in a time step. The bug, a set of missing initializers, was actually in a function called right before the temporally first predicate. Again, we did not find this bug directly, but a developer familiar with the algorithm in question would be able to identify it.

Our experience with these manually seeded bugs demonstrates that statistical debugging produces useful predicates. Some predicates, as in the first AMG bug and the ParaDiS bug, identify the region of a program in which errors occur. Others, as in the second AMG bug, identify strong data-dependence chains that point backwards through the program execution to real errors. Combined with domain knowledge of the intended function of a program, these can be powerful debugging aids.

VIII. Related Work

Two main bodies of work relate to ours: statistical debugging and debugging of large-scale parallel programs.

A. *Statistical Debugging*

Our work is closest to that of Liblit et al. [4], [7]. They instrument programs to collect predicates over multiple runs and then identify bug predictors by correlating predicates with program failures. Chilimbi et al. [17] improve the precision of statistical debugging by examining program path profiles instead of simple predicates. These prior efforts operate on desktop software with few threads (usually just one), while we address the challenges of statistical debugging of massively-parallel applications. Recent work by Bond et al. [18] and Jin et al. [19] consider sampled instrumentation strategies for detecting concurrency bugs, but their focus remains on desktop-class software. Jin et al. propose an instrumentation strategy that detects bad thread interleavings without adding new synchronization to the original code; this option could also apply to supercomputing environments.

Arumuga Nainar and Liblit [20] achieve low run-time overheads by instrumenting few predicates in each execution and directing the choice of predicates based on the feedback of prior runs. This approach could achieve low overheads for scientific applications, particularly if instrumentation is applied in the communication tree as binaries are being distributed. Unfortunately, the feedback-directed predicate search must analyze many runs before the next choice can be made; most individual versions of scientific applications do not receive enough runs to make this search practical.

Numerous statistical models have been proposed for identifying failure-predictive behaviors once data has been collected. These models include regularized logistic regression [5], [21], probability density function comparison [22], likelihood ratio testing [23], [7], iterative bipartite graph voting [24], [25], hypothesis testing [26], three-valued logic [1], support vector machines [27], random forests [27], and document topic analysis [28]. We do not propose new statistical models; instead we design efficient mechanisms to gather the raw data that these models require from scientific applications at supercomputing scales. Additionally, these existing models assume that runs are mutually independent while our applications have strong interdependencies among nodes. We find that at least the model used in our evaluation continues to perform well, discovering good bug predictors despite of this lack of independence.

B. *Distributed Tools*

While single-process debuggers are impractical at scale, TotalView and Allinea DDT are distributed debuggers that enable traditional debugging techniques for clusters. Unfortunately, bugs that are non-deterministic or otherwise difficult to resolve with traditional debuggers are even harder to find at scale. Our work investigates an alternative debugging paradigm that can provide deeper insight into the root causes of failures.

The Stack Trace Analysis Tool (STAT) of Ahn et al. [29] groups related processes into equivalence classes based on stack traces and program execution order. These equivalence classes make analyzing large numbers of processes manageable, and the associated stack traces have been used to diagnose large-scale application hangs. This type of tool complements our work and is especially valuable when no successful program runs are available for statistical analysis.

Execution traces are another tool for debugging large-scale applications; Ratn et al. [30] describe a lossless MPI trace collection tool. In a sense, our work records highly lossy program execution traces, throwing away event ordering but recording non-MPI events.

IX. Conclusion

Essentially all non-trivial programs contain bugs. Scientific applications, which are normally under constant development, are especially susceptible. The relative lack of debugging tools that are effective at supercomputing scales exacerbates this unfortunate situation. We have described our experience with bringing statistical debugging from the realm of desktop applications to the computationally-intensive domain of scientific computing. Statistical debugging identifies program predicates that correlate highly with failure. The statistical component of the analysis allows it to diagnose the root causes of traditionally-difficult bug classes, such as ones with non-deterministic or temporally-distant manifestations.

We have introduced optimizations to the run-time instrumentation required by statistical debugging to reduce the overhead by up to 25%. We have also described new data representations and collection mechanisms that efficiently and losslessly aggregate feedback data from 500,000 CPUs into less than 50MB. We have also found that, in practice, statistical debugging can remain effective even when the assumption of independence between runs, or compute nodes, is violated. Taken together, these advances point the way toward scalable, informative diagnosis of bugs in massively-parallel applications.

References

[1] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound Boolean predicates," in *ISSTA*, D. S. Rosenblum and S. G. Elbaum, Eds. ACM, 2007, pp. 5–15.

[2] A. Lal, J. Lim, M. Polishchuk, and B. Liblit, "Path optimization in programs and its application to debugging," in *ESOP*, ser. Lecture Notes in Computer Science, P. Sestoft, Ed., vol. 3924. Springer, 2006, pp. 246–263.

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

[4] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Sampling user executions for bug isolation," in *Proceedings of the Workshop on Remote Analysis and Measurement of Software Systems*, Portland, Oregon, May 9 2003, pp. 5–8.

[5] ——, "Bug isolation via remote program sampling," in *PLDI*. ACM, 2003, pp. 141–154.

[6] M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code," in *PLDI*, 2001, pp. 168–179.

[7] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *PLDI*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 15–26.

[8] M. Hauswirth and T. M. Chilimbi, "Low-overhead memory leak detection using adaptive statistical profiling," in *ASPLOS*, S. Mukherjee and K. S. McKinley, Eds. ACM, 2004, pp. 156–164.

[9] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Public deployment of Cooperative Bug Isolation," in *Proceedings of the Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04)*, A. Orso and A. Porter, Eds., Edinburgh, Scotland, May 24 2004, pp. 57–62.

[10] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *SC*. ACM, 2003, p. 21.

[11] "Dyninst API," Apr. 2011. [Online]. Available: http://www.dyninst.org/

[12] K. Davis and D. J. Quinlan, "ROSE: An optimizing transformation system for C++ array-class libraries," in *ECOOP Workshops*, ser. Lecture Notes in Computer Science, S. Demeyer and J. Bosch, Eds., vol. 1543. Springer, 1998, pp. 452–453.

[13] D. H. Ahn, D. C. Arnold, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Overcoming scalability challenges for tool daemon launching," in *ICPP*. IEEE Computer Society, 2008, pp. 578–585.

[14] V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis, "Scalable line dynamics in ParaDiS," in *SC*. IEEE Computer Society, 2004, p. 19.

[15] B. Dudson, M. Umansky, X. Xu, P. Snyder, and H. Wilson, "Bout++: A framework for parallel plasma fluid simulations," *Computer Physics Communications*, vol. 180, no. 9, pp. 1467 – 1480, 2009.

[16] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 89–100.

[17] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *ICSE*. IEEE, 2009, pp. 34–44.

[18] M. D. Bond, K. E. Coons, and K. S. McKinley, "Pacer: Proportional detection of data races," in *PLDI*, B. G. Zorn and A. Aiken, Eds. ACM, 2010, pp. 255–268.

[19] G. Jin, A. Thakur, B. Liblit, and S. Lu, "Instrumentation and sampling strategies for Cooperative Concurrency Bug Isolation," in *Proceedingsof the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, M. Rinard, Ed., SIGPLAN. Reno/Tahoe, Nevada: ACM, Oct. 2010.

[20] P. Arumuga Nainar and B. Liblit, "Adaptive bug isolation," in *ICSE (1)*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 255–264.

[21] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken, "Statistical debugging of sampled programs," in *NIPS*, S. Thrun, L. K. Saul, and B. Schölkopf, Eds. MIT Press, 2003.

[22] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical model-based bug localization," in *ESEC/SIGSOFT FSE*, M. Wermelinger and H. Gall, Eds. ACM, 2005, pp. 286–295.

[23] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *ASE*, D. F. Redmiles, T. Ellman, and A. Zisman, Eds. ACM, 2005, pp. 273–282.

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

[25] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: Simultaneous identification of multiple bugs," in *ICML*, ser. ACM International Conference Proceeding Series, W. W. Cohen and A. Moore, Eds., vol. 148. ACM, 2006, pp. 1105–1112.

[26] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Trans. Software Eng.*, vol. 32, no. 10, pp. 831–848, 2006.

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

[28] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu, "Statistical debugging using latent topic models," in *ECML*, ser. Lecture Notes in Computer Science, J. N. Kok, J. Koronacki, R. L. de Mántaras, S. Matwin, D. Mladenic, and A. Skowron, Eds., vol. 4701. Springer, 2007, pp. 6–17.

[29] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, "Scalable temporal order analysis for large scale debugging," in *SC*. ACM, 2009.

[30] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz, "Preserving time in large-scale communication traces," in *ICS*, P. Zhou, Ed. ACM, 2008, pp. 46–55.