ABHRANTA: Locating Bugs that Manifest at Large System Scales

Bowen Zhou, Milind Kukarni, Saurabh Bagchi
Purdue University

Abstract

A key challenge in developing large scale applications (both in system size and in input size) is finding bugs that are latent at the small scales of testing, only manifesting when a program is deployed at large scales. Traditional statistical techniques fail because no error-free run is available at deployment scales for training purposes. Prior work used *scaling models* to detect anomalous behavior at large scales without being trained on correct behavior at that scale. However, that work cannot localize bugs automatically. In this paper, we extend that work with automatic diagnosis technique, based on *feature reconstruction*, and validate our design through case studies with two real bugs from an MPI library and a DHT-based file sharing application.

1 Introduction

A key challenge in developing large-scale software, intended to run on many processors or with very large data sets, is detecting and diagnosing *scale-dependent* bugs. Most bugs manifest at both small and large scales, and as a result, can be identified and caught during the development process, when programmers are typically working with both small-scale systems and small-scale inputs. However, a particularly insidious category of bugs are those that predominantly arise at deployment scales. These bugs appear far less frequently, if at all, at small scales, and hence are often not caught during development, but only when a program is released into the wild and is deployed at large scales. As one example of this class of bugs, we present a case study of a bug in an MPI library which causes a sub-optimal algorithm to be taken when the total amount of data being exchanged between the processes of the parallel application is large. As another example, we present discussion of a bug in a DHT implementation which arises only when the number of participating peers is very large.

In a previous work, VRISHA [20], we derived a scaling model from small-scale runs and used it to detect bugs that cause behavioral deviation in large-scale runs of an application. Once a bug is detected in the application, the next challenge is to further localize the bug. Modern programs may have hundreds of thousands, or millions of lines of code, so simply identifying the program as the culprit will not suffice to fix the bug. Instead, the developer would like to know *where* the bug arose: which module, function, or even line number.

Unfortunately, while VRISHA's detection of bugs is automatic, it can only identify that the scaling trend has been violated; it cannot determine *which* program behavior violated the trend, nor *where* in the program the bug manifested. Hence, diagnosis in VRISHA is a manual process. The behavior of the program at the various small scales of the training set are inspected to predict expected behavior at the problematic large scale, and discrepancies from these manually-extrapolated behaviors can be used to hone in on the bug. This diagnosis procedure is inefficient for real-world applications for two reasons. First, the number of features could easily grow to a scale that is unmanageable by manual analysis. One can conceive of a feature related to each performance counter (such as, cache hit rate), each aspect of control flow behavior (number of times a calling context is seen, number of times a conditional evaluates to true, etc.), and each aspect of data flow behavior (number of times some elements of a matrix are accessed, etc.). Second, some scaling trends may be difficult to detect unless a large number of training runs at different scales are considered, again making manual inference of these trends tedious.

1.1 Our approach: ABHRANTA

This paper presents ABHRANTA[1], an *automatic, scalable approach to detecting and diagnosing bugs in large-scale systems.* ABHRANTA is based on the same high level concepts as VRISHA, but provides one key contribution: *automatic bug diagnosis*.

As described above, VRISHA's diagnosis technique requires careful manual inspection of program behaviors both from the training set and from the deployed run. ABHRANTA, in contrast, provides an *automatic* diagnosis technique, built on a key modification to the scaling model used by VRISHA. We adopt a statistical modeling technique from Feng *et al.* [9] that results in an "invertible" model. Essentially, such models not only detect deviations from a scaling trend for bug detection, but can actually be used to *predict* the expected, bug-free behavior at larger scales, lifting the burden of manual analysis of program behaviors. Therefore, bug localization can be

[1]ABHRANTA is a Sanskrit word meaning "one who cannot be made to err."

automated by contrasting the reconstructed bug-free behavior and the actual buggy behavior at a large scale and identifying the most diverging feature of program behavior as the root cause of bug.

2 Overview of ABHRANTA

This section presents a high level overview of ABHRANTA, an approach to automatically detecting and diagnosing scale-determined bugs in programs.

Figure 1 shows a block-diagram view of ABHRANTA's operation. The key components are: (i) collecting data that characterizes the behavior of a deployed application; (ii) building a statistical model from the training data; (iii) using the statistical model to detect an error caused by the application; and (iv) reconstructing the "expected" correct behavior of a buggy application to diagnose the fault.

2.1 Data collection

ABHRANTA operates by building a model of behavior for a program. To do so, it must collect data about an application's behavior, and sufficient information about an application's configuration to predict its behavior. The approach is broadly similar to that taken by VRISHA [20].

For a given application run, ABHRANTA collects two types of features: *control* features and *observational* features. Control features are a generalization of scale: they include of all input parameters to an application that govern its behavior. Example control features include input size, number of processes and, for MPI applications, process rank. Control features can be gathered for a program execution merely by analyzing the inputs and arguments to the program.

Observational features capture the observed behavior of the program. Examples include the number of times a particular branch is taken, or the number of times a particular function is called. ABHRANTA generates an observational feature for each unique calling context. In the case of instrumented network libraries, ABHRANTA also records the amount of communication performed by each unique calling context. This data collection is accomplished by using Pin [15] to instrument applications. Exactly what the observational features will be (*e.g.*, whether for libc library calls, all library calls, etc.) is driven by the developer, possibly with some idea of where the bug lies. The developer can of course cast a wider net with an attendant increase in cost of data collection.

Observational and control features are collected separately for each unit of execution we wish to build a model for. For example, when analyzing MPI applications, ABHRANTA collects data for each process separately, creating a model for individual processes of the application. In contrast, in our DHT case study (Section 4.2) ABHRANTA is configured to collect control and observational data for each message the application processes. Currently, the execution unit granularity must be specified by the developer; automatically selecting the granularity is beyond the scope of this work.

2.2 Model building

The basic approach to model building in ABHRANTA is similar to in VRISHA: a series of training runs are conducted, at different, small, scales (*i.e.*, with different control features). The control features for the training runs are collected into a matrix **C**, while the observational features are collected into a matrix **O**, with the property that row i of **C** and **O** contains the control and observational features, respectively, for the i-th process or thread in a training run. We then use a statistical technique called *Kernel Canonical Correlation Analysis* [4, 18] to construct two *projection* functions, f and g, that transform **C** and **O**, respectively, into matrices of the same (lower) dimensionality, such that the rows of the transformed matrices are highly correlated. The projection functions f and g comprise the model.

In VRISHA, the f and g projection functions are *non-linear*, allowing the model to capture non-linear relationships between scale and behavior. Unfortunately, the functions are not *invertible*: given a set of control features, it is very difficult to infer a set of observational features consistent with these control features. This facility is necessary for ABHRANTA's bug localization strategy, discussed below. Hence ABHRANTA uses a modified version of KCCA to construct its projection functions. The key difference of ABHRANTA's model is that while f (the projection function for the control features) remains non-linear, g (the projection function for the observational features) is linear. Section 3 explains how this modified model can be used to predict the behavior of large-scale runs.

2.3 Bug detection

ABHRANTA detects bugs by determining if the behavior of a program execution is inconsistent with the scaling trends captured by the behavioral model.

To detect bugs, ABHRANTA uses the same instrumentation used in the training runs to collect control and observational features from a test run. These features are projected into a common subspace using the projection functions f and g computed during the model building phase. If the projected feature sets are well-correlated, then the observed behavior of the program is consistent with the scaling trends captured in the model, and the program is declared bug-free. If the projected features are *not* well-correlated, then the program is declared buggy, and more sophisticated diagnosis procedures (discussed below) are initiated.

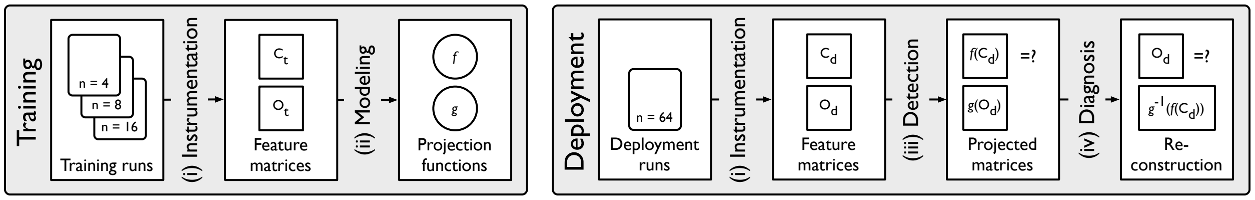

Figure 1: Overview of ABHRANTA architecture

2.4 Bug localization

Once a bug is detected, ABHRANTA then attempts to localize the bug to a particular function or even line of code. Unlike VRISHA, which relied on manual inspection to identify buggy behaviors (as discussed in Section 1), ABHRANTA attempts to *reconstruct* the expected non-buggy behavior of a buggy program, *i.e.*, it predicts what the behavior of the buggy program *would have been* had the bug not occurred.

To perform reconstruction, we take advantage of the fact that even though the observational features for a buggy program, $\mathbf{o}$ are anomalous, the control features for the program, $\mathbf{c}$, are nevertheless correct. A good guess for reconstruction is an $\mathbf{o}'$ such that $g(\mathbf{o}')$ is correlated with $f(\mathbf{c})$ (in other words, $\mathbf{o}'$ is a set of observational features that would appear non-buggy to our model). Section 3 describes how ABHRANTA infers $\mathbf{o}'$.

Given $\mathbf{o}'$ and $\mathbf{o}$, ABHRANTA's diagnosis strategy is straightforward. The two observational feature sets are compared. Those features whose values deviate the most between $\mathbf{o}'$ and $\mathbf{o}$ have been most affected by the bug, and hence are likely candidates for investigation. The features are ranked by the discrepancy between the actual observations and the reconstructed observations. Because each feature is associated with a calling context, investigating a feature will lead a programmer to specific function calls and line numbers that can help pinpoint the source of the bug.

3 Inferring expected program behavior

The key technical challenge for ABHRANTA's diagnosis strategy given a buggy run is to compute $\mathbf{o}'$, a prediction of what the observational features of the run would be were there no bug. An appealing approach to finding $\mathbf{o}'$ would be as follows. Given $\mathbf{c}$, the control vector for the buggy execution, compute $f(\mathbf{c})$ to find its projected image in the common KCCA subspace. Then, because both the control and observational features are intended to be highly correlated in the projected space, we can treat $f(\mathbf{c})$ as equivalent to the projected value of $\mathbf{o}'$, $g(\mathbf{o}')$. We can then compute $\mathbf{o}'$ by inverting g: $\mathbf{o}' = g^{-1}(f(\mathbf{c}))$ Unfortunately, as discussed before, the projection functions used in VRISHA were non-linear and non-invertible.

In ABHRANTA, we sidestep this problem by abandoning non-linear transformations of the observational features with g. Instead, we use a *linear* projection function for g, while leaving f as a non-linear function. Note that while using a simpler g means certain relationships cannot be captured, f remains non-linear, allowing us to still model program behaviors that vary non-linearly with scale. Section 4 confirms that this restricted modeling space does not significantly reduce ABHRANTA's detectability.

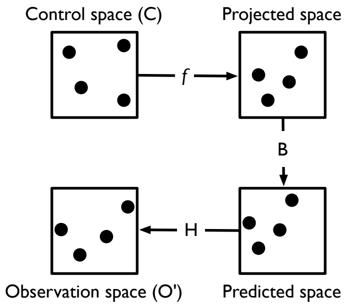

Figure 2: Process to derive reconstructed observations (O') from control features (C). f is a non-linear transformation, while B and H are linear.

ABHRANTA's reconstruction strategy is inspired by the preimage reconstruction method presented by Feng *et al.* [9]. Figure 2 shows the steps to reconstruct a predicted set of observational features $\mathbf{O}'$ given a set of control features $\mathbf{C}$. At a high level, we compute the projected form of $\mathbf{C}$, P_C, using the non-linear projection function for control features f. We then use a linear transformation, B, to predict the *projected* form of $\mathbf{O}'$, $P_{O'}$. We then compute a second linear transformation, H, which inverts the linear mapping provided by the projection function g, allowing us to compute $\mathbf{O}'$ as follows: $\mathbf{O}' = H \cdot B \cdot P_C$. How do we determine B and H?

To compute B, recall that f and g maximize the linear correlation between the control and observational features. Hence for non-buggy runs, we can assume that the projections of the control and observational features will be linearly correlated. We can hence compute B using linear regression (for an N-dimensional predicted space):

$$\min_B \sum_{i=1}^{N} \left\| BP_C^i - P_{O'}^i \right\|^2$$

Given B, we can predict the projected form of $\mathbf{O}'$ for a buggy execution. The next step is to undo that projection. Because ABHRANTA uses a linear kernel for observational features, this can be accomplished by deriving a reverse linear mapping from the projected space back to the original observational feature space. That is, we want to find an H such that $H \cdot P_{O'} = O'$. Because the projection subspace is of lower dimensionality than the original observational space, H is underdetermined. Hence, we find H by solving the following least-squares problem (for an n-dimensional observational feature space):

$$\min_H \sum_{i=1}^{n} \left\| H P_{O'}^i - O'^i \right\|^2$$

4 Evaluation

This section describes our evaluation of ABHRANTA. We present two case studies, demonstrating how ABHRANTA can be used to detect and localize bugs in real-world parallel and distributed systems. Both case studies concern scale-dependent bugs that are only triggered when executed with a large number of nodes. Thus, they are unlikely to manifest in testing, and must be detected at deployed scales. The case studies are conducted on a 16-node cluster running Linux 2.6.18. Each node is equipped with two 2.2GHz AMD Opteron Quad-Core CPUs, 512KB L2 cache and 8GB memory.

4.1 Case Study 1: MPICH2's ALLGATHER

ALLGATHER is a collective communication operation defined by the MPI standard, where each node exchanges data with every other node. The implementation of ALLGATHER in MPICH2 pre-1.2 contains an integer overflow bug [1], which is triggered when the total amount of data communicated causes a 32-bit `int` variable to overflow (and hence is triggered when input sizes are large or there are many participating nodes). The bug results in a sub-optimal communication algorithm being used for ALLGATHER, severely degrading performance.

Detection: In our prior work, we showed that VRISHA could detect the MPICH2 bug using KCCA with Gaussian kernels. To show that the linear kernel used by ABHRANTA does not affect detectability compared to VRISHA, we applied ABHRANTA to a test harness that exposes the ALLGATHER bug at scale, using both VRISHA's original Gaussian kernel and our new linear kernel. The control features were the number of processes in the program, and the rank of each process, while the observational features were the amount of data communicated at each unique calling context (i.e. call stack) in the program. The model is trained on runs with 4–15 processes (all non-buggy), while we attempted to detect the bug at 64 processes.

Experimentally, we validate that the use of the linear kernel does not hurt the detectability of ABHRANTA vis-à-vis VRISHA. We find that in both cases the buggy run has a significantly lower correlation between control and observational features than the test runs. We can quantify the accuracy of our model as the margin between the *lowest* correlation in the test case and the *highest* correlation in the buggy case. Using the linear kernel results in only a 7.2% drop in detection margin.

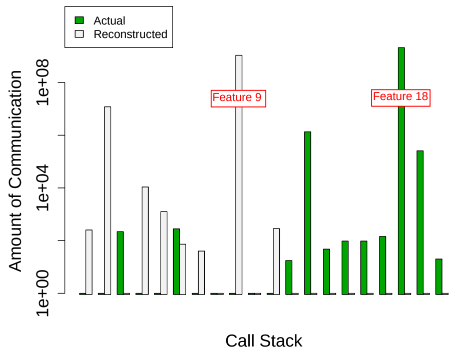

Figure 3: Reconstructed vs. actual buggy behavior for ALLGATHER

Localization: We next evaluate ABHRANTA's ability to localize the ALLGATHER bug by reconstructing the expected behavior of the 64-process execution. Figure 3 shows how the actual observational behavior of the buggy run compares with the reconstructed behavior. ABHRANTA ranks all the observational features in descending order of reconstruction error; this is the suggested order of examination to find the bug. The call stacks of the top two features, Features 9 and 18, differ only at the buggy `if` statement inside ALLGATHER, precisely locating the bug.

4.2 Case Study 2: Transmission's DHT

Transmission is a popular P2P file sharing application on Linux platforms. The bug [2] exists in its implementation of the DHT protocol (before version 0.18). When a new node joins the network, it sends a message to each known peer to find new peers. Each peer responds to these requests with a list of all its known peers. Upon receiving a response, the joining node processes the messages to extract the list of peers. However, due to a bug in the processing code, if the message contains a list of peers longer than 2048 bytes, it will enter an infinite loop.

It may seem that this bug could be easily detected using, *e.g.*, GPROF, which could show that the message processing function is consuming many cycles. However, this information is insufficient to tell whether there is a bug in the function or whether it is behaving normally but is just slow. ABHRANTA is able to definitively indicate that a bug exists in the program.

For this specific bug, given the information provided by GPROF, we can focus on the message processing function which is seen most frequently in the program's execution. We treat each invocation of the message processing function as a single execution instance in our model and use the function arguments and the size of the input message as the control features. For the observational feature, we generalize our instrumentation to track the number of calls, and the associated contexts, to any

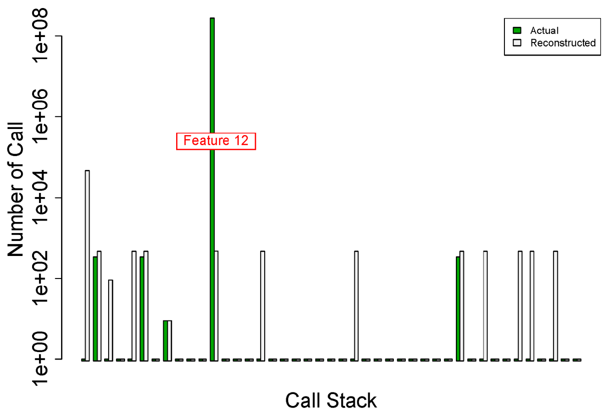

Figure 4: Reconstructed vs. actual buggy behavior for Transmission DHT

shared libraries.

To train ABHRANTA, we used 45 normal invocations of the message processing function, and apply the trained model to 1 buggy instance. First, ABHRANTA detects that the correlation for the buggy run is abnormally low, confirming that the buggy instance is truly abnormal behavior, and not just an especially long-running function. Having established that the long message is buggy, ABHRANTA reconstructs the expected behavior and compares it to the observed behavior, as in Figure 4. The rank ordering of deviant features highlights Feature 12, which corresponds to the call to a libc function `strtol`, only a few lines away from the root cause of the bug in this several-hundred-line function.

5 Related work

There is a substantial amount of work concerning statistical debugging [5, 6, 11–14, 16, 19, 20]. Some of these approaches focus primarily on detection, with diagnosis as a secondary, often *ad hoc* capability [5, 11, 16, 20], while others focus primarily on automatically assisting bug diagnosis [3, 7, 8, 10, 12–14, 19].

The typical approach taken for detection by statistical approaches [5, 6, 11, 16, 20] is to characterize a program's behavior as an aggregate of a number of features. A model is built based on the aggregate behavior of a number of training runs that are known to be buggy or non-buggy. To determine if a particular program execution exhibits a bug, the aggregate characteristics of the test program are checked against the modeled characteristics; deviation is indicative of a bug. The chief drawback to many of these approaches is that they do not account for scale. If the system or input size of the training runs differs from the scale of the deployed runs, the aggregate behavior of even non-buggy runs is likely to deviate from the training set, and false positives will result. Some approaches mitigate this by also detecting bugs in parallel executions if some processes behave differently than others [6, 16]; this approach does not suffice for bugs which arise equally in all processes (such as our MPI case study), or bugs that do not involve multiple processes (such as our DHT study).

Other statistical techniques eschew detection, in favor of attempting to debug programs that are known to have faults [3, 7, 8, 10, 12–14, 19]. These techniques all share a common approach: a large number of executions are collected, each with aggregate behavior profiled and labeled as "buggy" or "non-buggy." Then, a classifier is constructed that attempts to separate buggy runs from non-buggy runs. Those features that serve to distinguish buggy from non-buggy runs are flagged as involved with the bug, so that debugging attention can be focused appropriately. The key issue with all of these techniques is that they (a) rely on *labeled* data—whether or not a program is buggy must be known; and (b) they require a large number of *buggy* runs to train the classifier. In the usage scenario envisioned for ABHRANTA, the training runs are *all* bug-free, but bug detection must be performed *given a single buggy run*.

6 Conclusions and Challenges

We developed ABHRANTA, which leverages novel statistical modeling techniques to automate the detection and diagnosis of scale-dependent bugs where traditional statistical debugging techniques fail to provide satisfactory solutions. With case studies of two real-world bugs, we showed that ABHRANTA is able to automatically and effectively diagnose bugs.

Challenges There are several challenges that still remain to develop an effective system for diagnosing bugs in large-scale systems:

Feature selection To be effective at diagnosing scaling bugs, features must be (a) correlated with scale, and (b) related to the bug's manifestation. The former is necessary for the scaling model to be effective, while the latter is necessary for the bug to be detected. We are looking into approaches based on dynamic information flow to identify scale-related program behaviors to narrow down the set of possible features.

Model over-fitting A common pitfall in statistical modeling is over-fitting the training data, resulting in poor predictive performance for test data: the model may accurately predict behavior at scales close to those of the training set, but will fail as they are applied to ever-larger scales. Our current modeling approach uses very high-degree polynomials, increasing the likelihood of over-fitting. We are exploring the use of techniques such as the Bayesian Information Criterion (BIC) [17] to reduce the likelihood of over-fitting.

Non-deterministic behavior Many program behaviors are non-deterministic, which causes inaccurate trend predictions. Nevertheless, higher-level program behavior often is more predictable. For example, the amount of data sent over the network can be deterministic even if the particular network send methods used (immediate vs. buffered) may differ. We are investigating *aggregation* techniques that combine non-deterministic features to produce higher-level, deterministic features.

References

[1] `https://trac.mcs.anl.gov/projects/mpich2/changeset/5262`.

[2] `https://trac.transmissionbt.com/changeset/11666`.

[3] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In S. Matwin and D. Mladenic, editors, *18th European Conference on Machine Learning*, Warsaw, Poland, September 17–21 2007.

[4] F. R. Bach and M. I. Jordan. Kernel independent component analysis. *J. Mach. Learn. Res.*, 3:1–48, March 2003.

[5] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, , M. Schulz, and D. H. Ahn. Statistical Fault Detection for Parallel Applications with AutomaDeD. In *IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, pages 1–6, 2010.

[6] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, , and M. Schulz. AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks. In *40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 231–240, June-July 2010.

[7] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, pages 34–44, 2009.

[8] L. Dietz, V. Dallmeier, A. Zeller, and T. Scheffer. Localizing bugs in program executions with graphical models. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22 : Proceedings of the 2009 Conference*, pages 468–477, Vancouver, Canada, 2009. NIPS Foundation.

[9] W.-W. Feng, B.-U. Kim, and Y. Yu. Real-time data driven deformation using kernel canonical correlation analysis. In *ACM SIGGRAPH 2008 papers*, pages 91:1–91:9, 2008.

[10] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[11] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons Learned at 208K: Towards Debugging Millions of Cores. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC)*, SC '08, pages 1–9, 2008.

[12] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 141–154, New York, NY, USA, 2003. ACM.

[13] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[14] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32:831–848, October 2006.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[16] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem diagnosis in large-scale computing environments. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[17] G. E. Schwarz. Estimating the dimension of a model. *Annals of Statistics*, 6(2):461–464, 1978.

[18] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.

[19] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 1105–1112, New York, NY, USA, 2006. ACM.

[20] B. Zhou, M. Kulkarni, and S. Bagchi. Vrisha: using scaling properties of parallel programs for bug detection and localization. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 85–96, 2011.