TECHNIQUES FOR DETECTING SCALABILITY BUGS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Bowen Zhou

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2014

Purdue University

West Lafayette, Indiana

To my parents

ACKNOWLEDGMENTS

During my time at Purdue, I received countless help from different people: advisors, instructors, classmates, labmates, friends, and family members. Although most of my time was spent with a computer, coding or writing papers, it is these people that make the PhD a journey of no regrets in my life. I want to extend my sincere thanks to:

Milind Kulkarni, my advisor and role model in research. His insight, passion and pursuit for perfect solutions are always an inspiration. During these moments when I was frustrated by seemingly intangible problems and about to give up, he was always a reliable source for ideas and encouragement. He is as much a comrade as an advisor, willing to figure out the gory details buried in any technical problems his students may have. Thanks for being a fantastic advisor and friend.

Saurabh Bagchi, my advisor and instructor for countless 699 research credits. I will always remember the kindly manner and the perseverance for knowledge he showed during our first meeting in September 2009. Though most of our meetings were about research, from time to time we chatted about things other than research: politics, culture, life and much more. We even watched a world cup match together after a research meeting. Thanks for being a terrific advisor.

My PhD advisory committee members, Xiangyu Zhang and Ninghui Li, who attended my exams and provided invaluable feedbacks on my work.

Donghoon Shin, my labmate for three years. We don't have overlap in research but we chatted a lot when we were both in the lab. I cannot remember much of our conversations but I do remember the joy of talking with a senior student who is kind and experienced in pretty much everything about graduate school. Good luck in Arizona, Donghoon!

Myungjin Lee, my collaborator for summer of 2009 on a research project. I still remember his extraordinary programming skills.

Jianfu Li, my first roommate at Purdue and friend since the first day I landed on the continent of North America. Thanks for all exciting trips and distractions. Good luck with your PhD!

My Purdue friends: Hongbin Kuang, Weiqiang Chen, Chao Pan, Qiming Huang, Yang Zhao, Chao Xu, Jing Ouyang, Yu Zhang, Junyan Ge, Cheng Liu, Han Wu, Tanzima Zerin, Ignacio Laguna, Youngjoon Jo, Fahad Arshad, Amiya Maji, Nawanol Theera-Ampornpunt, Gaspar Modelo-Howard, Subrata Mitra, Tsungtai Yeh, Sambit Mishra, Sidharth Mudgal Sunil Kumar, Zaiwei Zhang, Jonanthan Too, Wei-Chiu Chuang, Bo Sang, Sunghwan Yoo, Zheng Zhang, Soumyadip Banerjee, Dimitrios Koutsonikolas, Abhinav Pathak.

Last but not least, I am forever indebted to my family for their support throughout my life and work. Without the happiness brought to me by my parents, my wife and my daughter, my life as a PhD student at West Lafayette would be miserable without a doubt.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

ABSTRACT xi

1 INTRODUCTION 1
  1.1 Statistical Debugging 2
  1.2 VRISHA 4
  1.3 ABHRANTA 6
  1.4 WUKONG 7
  1.5 LANCET 9
  1.6 Outline 13

2 VRISHA: USING SCALING PROPERTIES OF PARALLEL PROGRAMS FOR BUG DETECTION AND LOCALIZATION 14
  2.1 Overview 14
  2.2 Background: Kernel Canonical Correlation Analysis 17
    2.2.1 Canonical Correlation Analysis 17
    2.2.2 Kernel Canonical Correlation Analysis 17
    2.2.3 Comparison to Other Techniques 18
  2.3 Feature Selection 20
    2.3.1 Features Used by VRISHA 21
    2.3.2 Discussion 22
  2.4 Design 24
    2.4.1 Communication Profiling 24
    2.4.2 Building the KCCA Model 24
    2.4.3 Using Correlation to Detect Errors 25
    2.4.4 Localization of Bugs 26
    2.4.5 Discussion 28
  2.5 Evaluation 30
    2.5.1 Allgather Integer Overflow in MPICH2 30
    2.5.2 Bug in Handling Large Messages in MPICH2 36
    2.5.3 Performance Measurement 40
    2.5.4 Model Selection and False Positive Rate 40
  2.6 Summary 42

Page

3 ABHRANTA: LOCATING BUGS THAT MANIFEST AT LARGE SYSTEM SCALES . . . 44
3.1 Overview . . . 44
3.1.1 Data Collection . . . 44
3.1.2 Model Building . . . 45
3.1.3 Bug Detection . . . 46
3.1.4 Bug Localization . . . 47
3.2 Inferring Expected Program Behavior . . . 47
3.3 Evaluation . . . 49
3.3.1 Case Study 1: MPICH2's ALLGATHER . . . 49
3.3.2 Case Study 2: Transmission's DHT . . . 52
3.4 Summary . . . 53

4 WUKONG: AUTOMATICALLY DETECTING AND LOCALIZING BUGS THAT MANIFEST AT LARGE SYSTEM SCALES . . . 55
4.1 Overview . . . 55
4.2 Modeling Program Behavior . . . 56
4.2.1 Model Building . . . 56
4.2.2 Base Model Customization . . . 57
4.3 Feature Selection and Pruning . . . 59
4.4 Debugging Programs at Large Scales . . . 62
4.4.1 Bug Detection . . . 63
4.4.2 Bug Localization . . . 64
4.4.3 Sources and Types of Detection and Diagnosis Error . . . 64
4.5 Data Collection . . . 68
4.5.1 Control and Observational Features . . . 68
4.5.2 Optimizing Call Stack Recording . . . 69
4.6 Evaluation . . . 70
4.6.1 Fault Injection Study with AMG2006 . . . 71
4.6.2 Case Study 1: Performance Degradation in MPICH2 . . . 77
4.6.3 Case Study 2: Deadlock in Transmission . . . 79
4.6.4 Overhead . . . 82
4.7 Summary . . . 83

5 LANCET: GENERATING TARGETED SCALING TESTS . . . 85
5.1 Background: Dynamic Symbolic Execution . . . 85
5.1.1 Dynamic Symbolic Execution Basics . . . 85
5.1.2 Path Exploration Heuristics . . . 86
5.1.3 Dynamic Symbolic Execution Overhead . . . 87
5.1.4 WISE . . . 88
5.2 Design . . . 89
5.2.1 Overview of LANCET . . . 89
5.2.2 Explicit Mode . . . 93

Page

5.2.3 Inference Mode 95

5.3 Implementation 100

5.3.1 POSIX Thread 100

5.3.2 Socket 102

5.3.3 Libevent 102

5.3.4 Various Optimizations 103

5.4 Evaluation 104

5.4.1 General Observations with Benchmarks 105

5.4.2 Case Study with Memcached 107

5.5 Summary 109

6 RELATED WORK 110

6.1 Statistical Bug Detection and Diagnosis 110

6.2 Performance Test Generation 112

7 CONCLUSION 114

7.1 Data Dependence in Scaling Behavior Prediction 114

7.2 Environment Modeling in Symbolic Execution 115

LIST OF REFERENCES 117

VITA 123

LIST OF TABLES

Table | Page

2.1 Sensitivity of false positive rate to model parameters in Vrisha 42

4.1 Scalability of WuKong for AMG2006 on test runs with 256, 512 and 1024 nodes. 73

4.2 The accuracy of detection at various levels of detection threshold with a 90% pruning threshold. 75

4.3 The accuracy and precision of detection and localization at various levels of feature pruning with detection threshold parameter $\eta = 1.15$. 76

5.1 Lancet supports most of PThread API for thread management, synchronization and thread-specific store. 101

5.2 Effectiveness of Lancet and KLEE at generating scaling inputs for target programs. KLEE is unable to generate any inputs for `lbm`, as it runs out of memory. 106

5.3 Examples of path conditions generated for `Memcached`. ID, number of iterations and path condition are listed for each generated test. A character or space represents an equality constraint for the byte where it appears. A '*' symbol represents a constraint that enforces a non-space character for the byte where it appears. 108

LIST OF FIGURES

Figure Page

1.1 Example of a real bug. This bug appears in the MPICH2 library implementation and manifests itself at a large scale. 3

2.1 Overview of system architecture 14

2.2 Example for demonstrating localization of a bug. 27

2.3 Communication behavior for the Allgather bug at two training scales (4 and 8 nodes) and production scale system (16 nodes). The bug manifests itself in the 16 node system (and larger scales) 32

2.4 Correlation in the projection space using the KCCA-generated maps for systems of different scale. VRISHA is trained in 4- through 15-node systems (in light color) and tested in the buggy 16-node system (in dark color). The dashed line indicates the bug detection threshold. 32

2.5 Call stacks for the correct case (call stack 9, in the training system) and the erroneous case (call stack 16, in the production system). 33

2.6 Correlation of training and testing nodes for CG with synthetically injected bug. VRISHA is trained on 4- and 8-node systems (in light color) and tested on 16 nodes (in dark color). 35

2.7 Communication behavior for the large message bug at two training scales (512 MB and 1 GB) and production scale system (2 GB). The bug manifests itself in data sizes of 2 GB and larger. 38

2.8 Call stacks from a normal process (left) and at the point of crash due to large-sized data. Error message "socket closed" reported by MPICH2 at MPID_nem_tcp_send_queued helps localize the bug. 39

2.9 Overhead due to profiling in VRISHA for NASPAR Benchmark applications. 41

2.10 Modeling and detection time for CG, LU and MG on 4-, 8- and 16-node systems. 41

3.1 Overview of ABHRANTA architecture 44

3.2 Process to derive reconstructed observations (O') from control features (C). f is a non-linear transformation, while B and H are linear. 48

Figure Page

3.3 Reconstructed vs. actual buggy behavior for ALLGATHER 51

3.4 Reconstructed vs. actual buggy behavior for Transmission DHT 53

4.1 Possible outcomes and errors when using WuKong to detect and diagnose bugs. 65

4.2 MPICH2 bug that manifests at large scale as performance degradation. 77

4.3 The top suspicious features for the buggy run of MPICH2 ALLGATHER given by WuKong. 79

4.4 The deadlock bug appears in Transmission, and manifests when a large number of peers are contained in a single DHT message. 80

4.5 The top suspicious features for the buggy run of Transmission given by WuKong. 82

4.6 Runtime overhead of WuKong on NPB benchmarks. 83

5.1 High level flow of Lancet's inference-mode approach for generating inputs for a given loop. 90

5.2 Running example: request parsing in Memcached. 92

ABSTRACT

Zhou, Bowen Ph.D., Purdue University, August 2014. Techniques for Detecting Scalability Bugs. Major Professor: Xiangyu Zhang.

Developing correct and efficient software for large scale systems is a challenging task. Developers may overlook pathological cases in large scale runs, employ inefficient algorithms that do not scale, or conduct premature performance optimizations that work only for small scale runs. Such program errors and inefficiencies can result in an especially subtle class of bugs that are scale-dependent. While small-scale test cases may not exhibit these bugs, large-scale production runs may suffer failtures or performance issues caused by them. Without an effective method to find such bugs, the developers are forced to search through an enormous amount of logs generated in production systems to fix a scaling problem.

We developed a series of statistical debugging techniques to detect and localize bugs based on a key observation that most programs developed for large scale systems exhibit behavioral features predictable from the scale. Our techniques extrapolate these features to large scale runs, based solely on the training data collected in small scale runs. The predicted behaviors are then compared with the actual behaviors to pinpoint individual program features that contain a bug. We applied these techniques to detect and localize real-world bugs found in a popular MPI library, a P2P file sharing program, and synthetic faults injected into an HPC application.

We also built Lancet, a symbolic execution tool, to generate large scale test inputs for distributed applications. Lancet infers the constraints that a large-scale input should satisfy based on models built on small-scale inputs, allowing programmers to generate large-scale inputs without performing symbolic execution at large scales. With built-in support for multithreading, socket API, and event-driven asynchronous

programming, Lancet is ready to be applied to real world distributed applications. We demonstrated the effectiveness of Lancet by using it to generate large-scale, targeted inputs for various SPEC benchmarks and Memcached.

1 INTRODUCTION

Many software bugs result in subtle failures, such as silent data corruption [1], and degradation in the application performance [2]. These are undesirable because they make the results of applications untrustworthy or reduce the utilization of the computer systems. It is therefore imperative to provide automated mechanisms for detecting errors and localizing the bugs in software. With respect to error detection, the requirement is to detect the hard-to-catch bugs while performing lightweight instrumentation and runtime computation, such that the performance of application is affected as little as possible. With respect to bug localization, the requirement is to localize the bug to as small a portion of the code as possible so that the developer can correct the bug. These two motivations have spurred a significant volume of work in the research community, with a spurt being observable in the last decade [3–8]. Unlike prior work, we focus on bugs that manifest as software is scaled up.

A common development and deployment scenario for parallel and distributed systems is that the developer develops the code and tests it on small(ish)-sized computing clusters. The testing done by the developer at the small scale is rigorous (for well-developed codes) in that different input datasets, architectures, and other testing conditions are tried. Both correctness and performance errors can be detected through the manual testing as long as the error manifests itself in the small scale of the testbed that the developer is using. However, errors that manifest only at larger scale are less likely to be caught, for a number of reasons. The developer may not have access to large-scale systems; he may not have the resources to exhaustively test the application at larger scales; it can be difficult to even determine when bugs arise when dealing with large scale programs. This difference in the behavior of the application between what we will call *the testing environment* and *the production environment* provides the fundamental insight that drives our work.

To see an illustrative example, consider a bug in MPICH2 [9], a popular MPI library from Argonne National Lab. The bug shown in Figure 1.1 is in the implementation of the `allgather` routine, a routine for all-to-all communication. In `allgather`, every node gathers information from every other node using different communication topologies (ring, balanced binary tree, etc.) depending on the total amount of data to be exchanged. The bug is that for a large enough scale an overflow occurs in the temporary variable used to store the total size of data exchanged because of the large number of processes involved (Line 12). As a result, a non-optimal communication topology will be used. This is an instance of a performance bug, rather than a correctness bug, and may not be evident in testing either on a small-scale system or with small amount of data.

Classical program analysis based debugging techniques, such as dynamic program slicing [10, 11], would not help here because they rely on heuristics or user-defined rules to define symptoms of buggy behaviors, such as crash or deadlock, while the subtle performance issue demonstrated in the above example needs more sophisticated rules, e.g. a predictive model, to be identified from normal scaling behaviors.

1.1 Statistical Debugging

The most relevant class of prior works [5–7, 12] for error detection and bug localization use statistical approaches to create models for correct behavior. These approaches typically follow the same strategy: error free runs are used to build models of correct behavior, runtime behavior is modeled using profiling data collected at runtime via instrumentation, and if the runtime behavior deviates significantly from the normal behavior model, an error is flagged. The factor that causes the difference is mapped to a code region to aid in bug localization. If an error-free run is unavailable to produce the model, prior work relies on "majority rules." Under the assumption that most processes behave correctly, outliers are flagged as faulty.

```
1  int MPIR_Allgather (void *sendbuf, int sendcount,
2                      MPI_Datatype sendtype, void *recvbuf,
3                      int recvcount, MPI_Datatype recvtype,
4                      MPID_Comm *comm_ptr)
5  {
6      int         comm_size, rank;
7      int         mpi_errno = MPI_SUCCESS;
8      int         curr_cnt, dst, type_size,
9                  left, right, jnext, comm_size_is_pof2;
10     MPI_Comm    comm;
11     ...
12     if ((recvcount*comm_size*type_size < MPIR_ALLGATHER_LONG_MSG)
13          && (comm_size_is_pof2 == 1)) {
14     /* BUG IN ABOVE CONDITION CHECK DUE TO OVERFLOW */
15     /* Short or medium size message and power-of-two
16      * no. of processes. Use recursive doubling algorithm */
17     }
18     ...
19 }
```

Figure 1.1. Example of a real bug. This bug appears in the MPICH2 library implementation and manifests itself at a large scale.

However, the traditional statistical approach is insufficient to deal with scale-dependent bugs, especially as system and input scales become large. If the statistical model is trained only on small-scale runs, statistical techniques can result in numerous false positives. Program behavior naturally changes as programs scale up (*e.g.*, the number of times a branch in a loop is taken will depend on the number of loop iterations, which can depend on the scale), leading small-scale models to incorrectly label bug-free behaviors at large scales as anomalous. This effect can be particularly insidious in strong-scaling situations, where each process of a program inherently does less work at large scales than at small ones.

While it may seem that incorporating large-scale training runs into the statistical model will fix the aforementioned issue, doing so is not straightforward. If a developer cannot determine whether large-scale behavior is correct, it is impossible to correctly label the data to train the model. Furthermore, many scale-dependent bugs affect *all* the processes, i.e. Single Program Multiple Data (SPMD), and are triggered at *every*

execution at large scales. Thus, it would be impossible to get *any* sample of bug-free behavior at large scales for training purposes.

An additional complication in building models at large scale is the overhead of modeling. Modeling time is a function of training-data size, and as programs scale up, so, too, will the training data. Moreover, most modeling techniques require global reasoning and centralized computation. Hence, the overheads of performing complex statistical modeling on large-scale training data can rapidly become impractical.

1.2 Vrisha

To handle the problem of error detection and bug localization under the conditions identified above, we observe that as parallel applications scale up, some of their properties are either *scale invariant* or *scale determined*. By scale invariant, we mean that the property does not change as the application scales up to larger scales, and by scale determined, we mean that the property changes in a predictable manner, say in a linearly increasing manner. Example scale-invariant properties might include the number of neighbors that a process communicates with, or the relative communication volume with each neighbor. Example scale-determined properties might include the total volume of communication performed by a processor. This observation has been made in previous work [13], though no one has used it for error detection.

We leverage the above observation to build a system called Vrisha [14]. In it, we focus on bugs that manifest at large scales. In particular, we target bugs that affect communication behavior (though Vrisha could be adapted to other types of bugs). Examples of communication bugs include communicating with the wrong process, sending the wrong volume or type of data, or sending data to a legitimate neighbor, but in the wrong context (*e.g.*, an unexpected call site). This is an important class of bugs because bugs of this kind are numerous, subtle, and importantly, for the distributed nature of the computation, can result in error propagation. As a result of

error propagation, multiple processes may be affected, which will make it difficult to detect the failure and to perform recovery actions.

At a high level, VRISHA operates by building a model of the application running on a small scale in the testing environment. This model attempts to determine the relationship between certain input, or *control*, parameters (such as program arguments, and including the scale) and the programs behavior, captured by scale-determined *observational* parameters (such as which call sites are being used to send messages). By considering test runs at various scales, VRISHA can build a model that can predict, for any scale, what the expected behavior of the program will be. VRISHA does not make any *a priori* assumptions about how scale affects the observed behavior. In a production environment, VRISHA can use this predictive model to detect errors: if the observed behavior does not match the predicted behavior, VRISHA will flag an error. Next, VRISHA determines which part of the observed behavior deviated from the predicted behavior, aiding in bug localization.

The contributions of VRISHA are as follows.

1. It is the first to focus on bugs that are increasingly common as applications scale to larger-sized systems or inputs. These bugs manifest at large scales and at these scales, no error-free run is available and the common case execution is also incorrect. This appears to be a real issue since the application will ultimately execute at these large scales and at which exhaustive testing is typically not done.

2. VRISHA is able to deduce correlations between the scale of execution and the communication-related properties of the application. It makes no assumption about the nature of the correlation and it can belong to one of many different classes. Violation of this correlation is indicative of an error.

3. VRISHA handles bugs at the application level as well as at the library level because our monitoring and analysis are done at the operating system socket level, *i.e.*, beneath the library.

4. We show through experimental evaluation that our technique is able to detect errors and localize bugs that have been reported and manually fixed prior to our

work and that cannot be handled by prior techniques. We also show that VRISHA can do so at minimal performance overhead (less than 8%).

1.3 ABHRANTA

Unfortunately, while VRISHA takes a model-based automatic approach to bug detection, 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, without heuristics provided by human users. 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.

We build ABHRANTA [15] to complement VRISHA in bug diagnosis. ABHRANTA provides an automatic, scalable approach to localize bugs systematically using a new statistical model that allows inference on individual features. 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.* [16] 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 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.

1.4 WuKong

Learning from the experience with Vrisha and Abhranta, we start from the first principle and develop WuKong [17], a regression model based approach to detecting and diagnosing bugs that manifest at large system scales. WuKong provides three key contributions over the previous work:

Automatic bug localization

As described above, Vrisha's diagnosis technique requires careful manual inspection of program behaviors both from the training set and from the deployed run. WuKong, in contrast, provides an *automatic* diagnosis technique. WuKong alters Vrisha's modeling technique, using *per-feature regression models*, built across multiple training scales that can accurately *predict* the expected bug-free behavior at large scales.

When presented with a large-scale execution, WuKong uses these models to infer what the value of each feature *would have been* were a run bug-free. If any feature's observed value deviates from the predicted value by a sufficient amount, WuKong detects a bug. With carefully chosen program features that can be linked to particular regions of code (WuKong uses calling contexts rooted at conditional statements, as described in Section 4.2), ranking features by their prediction error can identify which

lines of code result in particularly unexpected behavior. This ranked list therefore provides a roadmap the programmer can use in tracking down the bug.

Feature pruning

Not all program behaviors are well-correlated with scale, and hence cannot be predicted by scaling models. Examples of such behaviors include random conditionals (*e.g.*, `if (x < rand())`) or, more commonly, data-dependent behaviors (where the *values* of the input data, rather than the *size* of that data determine behavior). The existence of such hard-to-model features can dramatically reduce the effectiveness of detection and localization: a feature whose behavior seems aberrant may be truly buggy, or may represent a modeling failure. To address this shortcoming, we introduce a novel cross-validation-based *feature pruning* technique. This mechanism can effectively prune features that are hard to model accurately from the training set, allowing programmers to trade off reduced detectability for improved localization accuracy. We find that our pruning technique can dramatically improve localization with only a minimal impact on detectability.

Scaling

A key drawback to many statistical debugging techniques is the scalability of both the initial modeling phase, and the detection phase. As scales increase, the cost of building statistical models of large-scale behavior becomes prohibitive, especially with global modeling techniques. WuKong possesses an intriguing property: because the training models do not need to be built at large scales, *WuKong's modeling cost is independent of system scale.* Hence, WuKong is uniquely suited to diagnosing bugs in very large scale systems. Furthermore, because WuKong's detection strategy is purely local to each execution entity, detection and diagnosis cost scales only linearly with system size, and is constant on a per-process basis.

In this work, we show that our per-feature scaling models can be used to effectively detect and diagnose bugs at large scales ($>$ 1000 processes) even when trained only at small scales ($\leq$ 128 processes). In particular, we show through a fault-injection study that not only can WuKong accurately detect faulty program behaviors in large-scale

runs, but that it can correctly locate the buggy program location 92.5% of the time. We show that our modeling errors and overheads are independent of scale, leading to a truly scalable solution.

1.5 Lancet

Our bug detection and diagnosis techniques all require a series of test inputs that exhibit the scaling trends of a given application to capture the correlation between scale and program behavior. We have been crafting such test inputs by manually analyzing the source code of applications or adopting the companion test suites provided by authors of open source software packages. The need for a systematic approach to generating performance tests motivates the last piece of work in this thesis. The following paragraphs will discuss the general motivation for a performance test generation tool and give a high-level overview of this work.

Writing correct and performant software running at scale has always been a challenge, especially so given the rising popularity of multithreaded and distributed systems. Scalability turns out to be the Achilles heel of many, otherwise well-designed software systems, which suffer from correctness or performance problems when executed at a large scale. For example, Foursquare, a geo-social network for sharing experiences of places, had an unprecedented 17 hours of downtime because the data stored in one of its two MongoDB shards reached the hosting computer's RAM capacity [18]. This is an example of a scaling problem with the data size. As another example, the Hadoop Distributed File System (HDFS) runs into performance problems when the number of clients becomes too large, relative to the processing power of the namespace server [19]. This is an example of a scaling problem that is triggered by a large number of nodes, and indirectly, by large sizes of data.

Bugs often happen out of the frequent paths of a program, rather in places where less attention has been paid to in the development process. As a remedy, unit tests are often introduced to cover both hot and cold paths and check for errors in the

runtime. An important quality criterion for a unit test suite is code coverage, or how much portion of the entire code base of the system under test (SUT) is touched by a test suite. By the conventional definition of code coverage, a line of code is considered covered if it is executed for at least once by a test. Such definition of code coverage is fraudulent in two means. First, it is purely a control flow concept and therefore completely ignorant to the data flow. For example, a null pointer dereference error may never be triggered by a unit test that exercises the line of code but with a valid pointer. Furthermore, running every part of SUT for once is not sufficient to reveal many performance issues. For example, there was a scalability issue in a LRU cache implementation of MySQL [20], which can only be detected by running SQL SELECT in a loop for a large number of times.

Performance testing, as one kind of system testing that performs end-to-end black-box testing, is designed to find performance issues in software. The idea in its essence is to run SUT through a large input, measure the sustainable performance and observe if any failure is triggered in the runtime. For simple programs, it is obvious that a larger input will invoke a longer execution. For example a string copy takes more time for a longer input string. However, it requires years of practice before one can master the black art of finding "large" inputs for complex real-world software systems. Sheer increasing the amount of data contained in the input does not necessarily lead to a longer execution, depending on how the program processes the input data. Take the classic binary search algorithm for example, its run time grows as a logarithmic function of the input size. To double the run time of binary search, the input must be an order of magnitude larger in size. On the other hand, more data means more logs, which are more difficult to analyze if a bug does occur. Numerous research projects invested in log analysis [21] have proved the task a difficult one.

It is only natural to ponder if it is possible to achieve a long execution or a load spike with a reasonably sized input for SUT, or how to push an execution to the limit with inputs of a certain size. Unfortunately, there exists no systematic method to the best of our knowledge that leads to a performance test that induces a significant

level of workload on SUT, not to mention striking a balance between execution length and input size. QA engineers often need to understand the internals and interfaces of complex software systems that consist of a pile of distributed components and millions of lines of code written by other developers, then endure a lengthy trial-and-error session to find a performance test of good quality. If we take a closer look, the QA workflow consists of the following steps: (a) making the test plan, i.e. starting with simple tests focused on individual code paths, followed by combination of different paths to simulate real user behaviors; (b) understanding the relevant code path and every other part of SUT that interacts with it to get sufficient knowledge for creating the performance tests; (c) developing and running the performance tests, while measuring the resultant performance using profiling tools. Out of these steps, (a) defines the policy, or the goal for the performance tests, for which human effort is indispensable, while (b) and (c) provide the mechanisms that implement and verify the resultant performance tests, which also form the most laborious part of the process for QA engineers.

We introduce Lancet, a performance test generation tool, to address the pain in the search process of performance tests by automating the latter two steps in the QA workflow where computers can be more efficient than human beings. Lancet is built on top of symbolic execution and statistical prediction, providing a tool with: (a) low entry barriers, i.e. little knowledge of SUT internals is required to create good performance tests, (b) systematic algorithms to reason the performance impact of values in the input, (c) integrated statistical models to extrapolate the scaling trend of SUT, (d) support for widely used concurrent and event-driven programming libraries, (e) on-the-fly SUT instrumentation for measurement and verification of generated performance tests. By default, Lancet inspects every loop in a program and tries to find inputs to run each loop for the number of iterations specified by the user. In the case where the user is familiar with the implementation of SUT, he can designate a set of target loops or even a single loop to reduce the scope of code considered by Lancet and speed up the test generation process. The reason

for choosing loops is twofold. First, programs spend a large part of their execution time in loops. Second, we observe, through examination of applications that have had documented scalability problems, as the application is run at larger scales, the negative impact of inefficient or incorrect code inside a loop often gets amplified into severe performance regressions or cascading failures.

When applying LANCET to an application, the user can specify the loop he wants to test and the load level, *i.e.,* the number of iterations, for the loop. LANCET takes these parameters from the user as guidance to steer its symbolic execution engine and finds the inputs that reach the given number of iterations for the given loop. LANCET makes two changes to the state-of-the-art symbolic execution algorithm. First, to reach a specific number of iterations after entering the target loop, LANCET uses a loop-centric search heuristic that favors the paths that stay inside the loop over those that exit. This path prioritization strategy is in contrast to existing path strategies that favor finding new paths. Second, LANCET shortcuts the path exploration by applying constraint inference to derive the path constraints at a large number of iteration from training samples, which consist of path constraints from running the loop a small number of iterations. This way, to reach N iterations of a given loop, LANCET just needs to execute the loop for $1 \ldots M$ iterations where $M \ll N$ and collect the constraints at each iteration. Afterwards, the constraint solver is invoked for the extrapolated path constraints to get the input that would trigger N iterations of the target loop.

We apply LANCET to four disparate benchmarks: a linear algebra code, `mvm`, a quantum computer simulator from SPEC, `libquantum`, a fluid dynamics program, `lbm`, a utility from the GNU Coreutils suite, `wc`, and `memcached`, a distributed in-memory caching system. Each of these programs needs to scale up to satisfy common use cases: `mvm` to tackle large matrices, `lq` to factorize large numbers, `lbm` to simulate flows for more timesteps, `wc` to process large text corpuses, and `memcached` to retrieve more data objects for a request. We show that for these programs, LANCET is able to generate more, and larger-scaling, inputs than a state-of-the-art test gen-

eration tool when given the same amount of computation time, and that in all cases, LANCET can generate even larger inputs through the use of constraint inference. Moreover, because of the regularity of the constraints that LANCET performs inference over, it is able to make its predictions perfectly, allowing programmers to correctly generate inputs of any size without incurring the cost of additional symbolic execution.

1.6 Outline

Chapter 2 provides a detailed description of VRISHA's approach to bug detection, Chapter 3 describes ABHRANTA's new model to automatic bug localization, Chapter 4 discusses WUKONG's regression-based feature prediction and feature pruning techniques, and Chapter 5 describes LANCET, a symbolic execution tool for performance test generation. The thesis is concluded in Chapter 7.

2 VRISHA: USING SCALING PROPERTIES OF PARALLEL PROGRAMS FOR BUG DETECTION AND LOCALIZATION

2.1 Overview

A ten-thousand foot overview of Vrisha's approach to bug detection and diagnosis is given in Figure 2.1. As in many statistical bug finding techniques, Vrisha consists of two phases, the *training phase*, where we use bug-free data to construct a model of expected behavior, and the *testing phase*, where we use the model constructed in the training phase to detect deviations from expected behavior in a production run. We further subdivide the two phases into five steps, which we describe at a high level below. The phases are elucidated in further detail in the following sections.

(a) The first step in Vrisha is to collect bug-free data which will be used to construct the model. Vrisha does this by using instrumented training runs to collect statistics describing the normal execution of a program (see Section 2.4.1). These statistics are collected on a per-process basis. Because we are interested in the scaling behavior of a program, whether by increasing the number of processors or the input size, our training runs are conducted at multiple scales, which are nevertheless smaller than the scales of the production runs. The difficulties of doing testing or getting

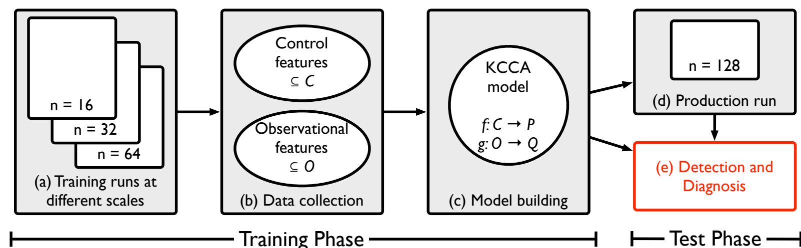

Figure 2.1. Overview of system architecture

error-free runs at large scales that we mentioned in the Introduction also apply to the process of building correct models and hence our training runs are done at small scales. The executions at multiple scales provides enough data to the modeling steps to allow us to capture the scaling properties of the program.

(b) After collecting profiling data from the training runs, Vrisha aggregates that data into two feature sets, the *control* set and the *observational* set. The characteristics of a particular process in a particular training run can be described using a set of *control features*. Conceptually, these control features are the “inputs” that completely determine the observed behavior of a process. Examples of these features include the arguments to the application (or particular process) and the MPI *rank* of the process. Crucially, because we care about scaling behavior, the control features also include information on the scale of the training run (*e.g.*, the number of processes, or the size of input). Each process can thus be described by a feature vector of these control features, called the *control vector*.

The control vector for a process captures the input features that determine its behavior. To describe the processes' actual behavior, Vrisha uses *observational* features that are collected at runtime through lightweight instrumentation that it injects at the socket layer under the MPI library. Example observational features for a process might include its number of neighbors, the volume of data communicated from a single call site, or the distribution of data communicated of different types. The selection of observational features constrains what types of bugs Vrisha can detect: a detectable bug must manifest in abnormal values for one or more observational features. Section 2.3 discusses our choice of features. The feature vector of observations for each process is called its *observation vector*.

(c) The third, and final, step of the training phase is to build a model of observed behavior. Vrisha uses KCCA [22, 23] to build this model. At a high level, KCCA learns two projection functions, $f : \mathcal{C} \to \mathcal{P}$ and $g : \mathcal{O} \to \mathcal{Q}$, where $\mathcal{C}$ is the domain of control vectors, $\mathcal{O}$ is the domain of observation vectors, and $\mathcal{P}$ and $\mathcal{Q}$ are projection domains of equal dimension. The goal of f and g is to project control and observation

vectors for a particular process into the projection domains such that the projected vectors are correlated with each other. These projection functions are learned using the control and observation vectors of bug-free runs collected in step (b).

Intuitively, if an observation vector, $o \in \mathcal{O}$, represents the correct behavior for a control vector, $c \in \mathcal{C}$, projecting the vectors using f and g should produce correlated results; if the observation vector does not adhere to expected behavior, $f(c)$ will be uncorrelated with $g(o)$, signaling an error. Crucially, because the control vectors c include information about the program's scale, KCCA will incorporate that information into f, allowing it to capture scaling trends. Further background on KCCA is provided in Section 2.2. The construction of the projection functions concludes the training phase of Vrisha.

(d) To begin the testing phase, Vrisha adds instrumentation to the at-scale production run of the program, collecting both the control vectors for each process in the program, as well as the associated observation vectors. Note that in this phase we do not know if the observation vectors represent correct behavior.

(e) Finally, Vrisha performs detection and diagnosis. The control vector of each process, c, accurately captures the control features of the process, while the observation vector, o, may or may not correspond to correct behavior. Vrisha uses the projection functions f and g learned in the training phase to calculate $f(c)$ and $f(o)$ for each process. If the correlation between the two vectors is above some threshold τ, then Vrisha will conclude that the process's observed behavior corresponds to its control features. If the projections are uncorrelated, then the observed behavior does not match the behavior predicted by the model and Vrisha will flag the process as faulty. Vrisha then performs further inspection of the faulty observation vector and compares it to the observation vectors in the training runs, after they have been scaled up, to aid in localizing the bug. VRISHA's detection and localization strategies are described in further detail in Sections 2.4.3 and 2.4.4, respectively.

2.2 Background: Kernel Canonical Correlation Analysis

In this section, we describe the statistical techniques we use to model the behavior of parallel programs, *kernel canonical correlation analysis* (KCCA) [22, 23].

2.2.1 Canonical Correlation Analysis

KCCA is an extension of *canonical correlation analysis* (CCA), a statistical technique proposed by Hotelling [24]. The goal of CCA is to identify relationships between two sets of variables, $\mathbf{X}$ and $\mathbf{Y}$, where $\mathbf{X}$ and $\mathbf{Y}$ describe different properties of particular objects. CCA determines two vectors $\mathbf{u}$ and $\mathbf{v}$ to maximize the correlation between $\mathbf{Xu}$ and $\mathbf{Yv}$. In other words, we find two vectors such that when $\mathbf{X}$ and $\mathbf{Y}$ are projected onto those vectors, the results are maximally correlated. This process can be generalized from single vectors to sets of basis vectors.

In our particular problem, the rows of $\mathbf{X}$ and $\mathbf{Y}$ are processes in the system. The columns of $\mathbf{X}$ describe the set of "control" features of process; the set of characteristics that determine the behavior of a process in a run. For example, the features might include the number of processes in the overall run, the rank of the particular process and the size of the input. The columns of $\mathbf{Y}$, on the other hand, capture the observed behavior of the process, such as the number of communicating partners, the volume of communication, etc. Intuitively, a row x_i of $\mathbf{X}$ and a row y_i of $\mathbf{Y}$ are two different ways of describing a single process from a training run, and CCA finds two functions f and g such that, for all i, $f(x_i)$ and $g(y_i)$ are maximally correlated.

2.2.2 Kernel Canonical Correlation Analysis

A fundamental limitation of CCA is that the projection functions that map $\mathbf{X}$ and $\mathbf{Y}$ to a common space must be linear. Unfortunately, this means that CCA cannot capture non-linear relationships between the control features and the observational features. Because we expect that the relationship between the control features and

the observational features might be complex (*e.g.*, if the communication volume is proportional to the square of the input size), using linear projection functions will limit the technique's applicability.

We turn to KCCA, an extension of CCA that allows it to use *kernel functions* to transform the feature sets $\mathbf{X}$ and $\mathbf{Y}$ into higher dimensional spaces before applying CCA. Intuitively, we would like to transform $\mathbf{X}$ and $\mathbf{Y}$ using non-linear functions Φ_X and Φ_Y into $\Phi_X(\mathbf{X})$ and $\Phi_Y(\mathbf{Y})$, and apply CCA to these transformed spaces. By searching for linear relations between non-linear transformations of the original spaces, KCCA allows us to capture non-linear relationships between the two feature spaces.

Rather than explicitly constructing the higher dimensional spaces, KCCA leverages a "kernel trick" [22], allowing us to create two new matrices κ_X and κ_Y from $\mathbf{X}$ and $\mathbf{Y}$, that implicitly incorporate the higher dimensional transformation, as follows:

$$\kappa_X(i,j) = \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j) \tag{2.1}$$

with κ_Y defined similarly. In particular, as in prior work [25, 26] we use a Gaussian, defining κ_X as follows:

$$\kappa_X(i,j) = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}} \tag{2.2}$$

with κ_Y defined analogously. Because we use a Gaussian to construct the κs, when we apply CCA to κ_X and κ_Y, we effectively allow CCA to discover correlations using infinite-degree polynomials. The upshot of KCCA is that we can determine two *non-linear* functions f and g such that, for all i, the correlation between $f(x_i)$ and $g(y_i)$ is maximized. We can thus capture complex relationships between the control features and the observed features.

2.2.3 Comparison to Other Techniques

A natural question is why we choose to use KCCA as opposed to other model-building techniques, such as multivariate regression or principal component analysis

(PCA). Multivariate regression attempts to find a function f that maps the input, independent variables X to dependent variables Y. We could consider the control features to be the independent variables, and the observational features to be the dependent variables. However, regression analysis typically requires that the input variables be independent of each other, which may not be the case. More generally, the problem with any model that simply maps the control variables to the observational variables is that such a mapping must account for all the observational variables. Consider an observational feature such as execution time, which is not truly dependent on the control features (because, *e.g.*, it is also dependent on architectural parameters that are not captured by the control features). If the model attempts to predict execution times, then it may be particularly susceptible to false positives since two non-buggy runs with the same control features may exhibit different execution times. Because KCCA projects both the control features and the observational features into new spaces, it is able to disregard features that may not be related to each other. Section 2.3 explores this advantage of KCCA further.

Another approach to modeling is to use PCA build a predictive model for the observational features. Bug detection can then be performed by seeing if the observational features of the production run correspond to the model. Unfortunately, a simple PCA-based model will not accommodate different non-buggy processes that have different observational behaviors. In particular, such a model cannot take into account scaling effects that might change the observed behavior of a program as the system size or the data size increases. Instead, additional techniques, such as clustering, would need to be applied to account for different possible behaviors. For example, we could build separate PCA models at varying scales and then apply non-linear regression to those models to infer a function that predicts observational feature values at new scales. KCCA, by contrast, incorporates scaling effects into its modeling naturally and avoids having to separately derive a scaling model.

2.3 Feature Selection

A critical question to answer when using statistical methods to find bugs is, what features should we use? To answer this question, we must consider what makes for a good feature. There are, broadly, two categories of characteristics that govern the suitability of a feature for use in Vrisha: those that are necessary for KCCA to produce a scale-determined model, and those that are necessary for our techniques to be useful in finding bugs. Furthermore, because Vrisha uses KCCA to build its models, we must concern ourselves with both control features and observational features.

First, we consider what qualities a feature must possess for it to be suitable for Vrisha's KCCA-based modeling.

- The control features we select must be related to the observational features we collect. If there is no relation, KCCA will not be able to find a meaningful correlation between the control space and the observation space. Moreover, because we care about scaling behavior, the scale (system and input size) must be included among the control features.
- The observational features should be scale-determined: Changing the scale while holding other control features constant should either have no effect on behavior or affect behavior in a deterministic way. Otherwise, Vrisha's model will have no predictive power.

Second, we consider what criteria a feature must satisfy for it to provide useful detectability.

- The observational features must be efficient to collect. Complex observational features will require instrumentation that adds too much overhead to production runs for Vrisha to be useful.
- The observational features must be possible to collect without making any change to the application. This is needed to support existing applications and

indicates that the instrumentation must be placed either between the application and the library, or under the library. VRISHA takes the latter approach.

- The observational features must reflect any bugs that are of interest. VRISHA detects bugs by finding deviations in observed behavior from the norm. If the observational features do not change in the presence of bugs, VRISHA will be unable to detect faults. Notably, this means that the types of bugs VRISHA can detect are constrained by the choice of features.
- The control features must be easily measurable. Our detection technique (described in Section 2.4.3) assumes that the control vector for a potentially-buggy test run is correct. The control features must cover all the attributes of the input that determine the observational behavior, such as scale, input size, etc. If some input attributes affect observational behavior but are not captured by the control features, VRISHA may be unable to build an accurate predictive model.

2.3.1 Features Used by VRISHA

The features used by VRISHA consist of two parts corresponding to the control feature set $\mathcal{C}$ and the observational feature set $\mathcal{O}$. The control features include (a) the process ID, specifically VRISHA uses the rank of process in the default communicator `MPI_COMM_WORLD` because it is unique for each process in the same MPI task; (b) the number of processes running the program, which serves as the scale parameter to capture system scale-determined properties in communication; (c) the argument list used to invoke the application, which serves as the parameter that correlates with input scale-determined properties in the communication behavior of application because it typically contains the size of the input data set.

The observational feature set of the ith process is a vector $\mathbf{D_i}$ of length c, where c is the number of distinct MPI call sites manifested in one execution of the program.

$$\mathbf{D_i} = (d_{i1}, \cdots, d_{ic})$$

The jth component in $\mathbf{D_i}$ is the volume of data sent at the jth call site. The index j of call sites has no relation with the actual order of call sites in the program. In fact, we uniquely identify each call site by the call stack to which it corresponds.

The set of control and observational features we choose has several advantages. First, they are particularly suitable for our purpose of detecting communication-related bugs in parallel programs. Second, it is possible to capture these features with a reasonable overhead so they can be instrumented in production runs. These features are easy to collect through instrumentation at the Socket API level. Further, with these features, we can identify call sites in a buggy process that deviate from normal call sites and further to localize the potential point of error by comparing the call stacks of the buggy process and the normal process.

We could also consider the features from previous solutions. For example, the frequent-chain and chain-distribution features from DMTracker [7] are good candidates to be adapted into the observational variable set in VRISHA's framework. Also, the distribution of time spent in a function used by Mirgorodskiy *et al.* [5] is also a good feature to characterize timing properties of functions in a program and can also be imported into VRISHA to diagnose performance-related bugs as in prior work.

2.3.2 Discussion

While it may seem that the criteria governing feature selection make the process of choosing appropriate control and observational features quite difficult, there are several aspects of VRISHA's design that lessen the burden.

First, VRISHA's use of KCCA make it robust to choosing too many features. If a control feature has no effect on observational behavior (*i.e.*, its value is completely uncorrelated with observed behavior) or an observational feature is not determined by

the control features (*e.g.*, its value is determined by other inputs not captured by the control features), KCCA is able to ignore the features when building its model. In the bug detection experiment detailed in Section 2.5.1, VRISHA is able to use the features described in Section 2.3.1 to detect an integer overflow bug. We experimented with adding additional "noise" features to both the control features and the observational features that were uncorrelated with behavior. We found that VRISHA's ability to detect the bug was unaffected even in the presence of as many noise features as real features.

Furthermore, Section 2.5.1 demonstrates that even if only a few of the observational features are affected by the presence of a bug, VRISHA is still able to detect it. The upshot of these findings is that inadvertently adding additional features to either the control or observational feature sets, whether those features are representative of correct behavior or completely unrelated to correct behavior, does not affect VRISHA's detectability. Hence, the primary constraint on choosing features is that a subset capture the types of bugs the user is interested in and that they be able to be collected efficiently. Users need not worry that the effectiveness of the model will be impacted by being overly thorough in selecting observational or control features.

Second, the primary bottleneck in VRISHA is running the program at scale. As Section 2.5.3 demonstrates, bug detection time is much less than the time it takes to collect the observational features. We thus envision the following usage scenario for VRISHA. Users can collect as many observational features as is practical for a particular program, spanning a wide range of program characteristics. They can then perform bug detection multiple times on various subsets of the observational features to detect bugs of various types (*e.g.*, all features that might reveal control flow bugs, all that might reveal ordering bugs, etc.).

2.4 Design

In this section, we explain the design of the runtime profiling component, the KCCA prediction model, bug detection method and bug localization method in Vrisha.

2.4.1 Communication Profiling

In order to detect bugs in both application and library level, we implement our profiling functionality below the network module of the MPICH2 library and on top of the OS network interface. So the call stack we recorded at the socket level would include functions from both the application and the MPICH2 library. The call stack and volume of data involved in each invocation of the underlying network interface made by MPICH2 is captured and recorded by our profiling module. The profiling component can be implemented inside the communication library or as a dynamic instrumentation tool separately. In our current prototype implementation of Vrisha, profiling and recording is piggy-backed in the existing debugging facility of MPICH2. This design is distinct from FlowChecker where, though the instrumentation is at the same layer as ours, it can only capture bugs in the library. Thus, application-level calls are not profiled at runtime by FlowChecker.

2.4.2 Building the KCCA Model

First, we construct two square kernel matrices from the values of the control and the observational variables respectively. These matrices capture the similarity in the values of one vector with another. Thus, the cell (i, j) will give the numerical similarity score between vector (control or observational) i and vector j. Since all our variables are numerical, we use the Gaussian kernel function [23] to create the kernel matrices, which is defined in Equation 2.2. Then we solve the KCCA problem to find the projections from the two kernel matrices into the projection space that give the

maximal correlation of the control and the observational variables in the training sets. Finally, we can use the solution of KCCA to project both control and observational variables to the same space spanned by the projection vectors from KCCA.

KCCA instantiated with Gaussian kernel depends on four parameters, namely, N_{comps} (the number of components to retain in the projected space), γ (the regularisation parameter for improving numerical stability), σ_x (kernel width parameter for the observational feature set) and σ_y (kernel width parameter for the control feature set). As done in previous work [25,26], we set the kernel width σ_x and σ_y in Gaussian kernel used by KCCA to be a fixed factor times the sample variance of the norms of data points in the training set. Similarly, we used a constant value for N_{comps} and γ throughout all our experiments. We explore the sensitivity of VRISHA to different model parameters in Section 2.5.4.

2.4.3 Using Correlation to Detect Errors

To detect if there is an error we find, for each process, the correlation between its control vector and its observational vector in the projected space spanned by the projection vectors found when building the KCCA model. The lack of correlation is used as a trigger for detection and the quantitative value of correlation serves as the metric of abnormality of each process. Since KCCA provides two projection vectors that maximizes correlation between the control and observational variables, most normal processes would have a relatively high correlation between the two sets of variables. Therefore, we can set a threshold on the deviation of correlation from 1 (which corresponds to perfectly correlated) to decide whether a process is normal or abnormal. Specifically, the threshold in VRISHA is set empirically based on the mean and standard deviation of $(1 - Correlation)$ of the training set. Because we check for correlation on a per process basis, our detection strategy inherently localizes bugs to the process level.

2.4.4 Localization of Bugs

Bugs that do not Cause Application Crash

Our strategy for localization of bugs uses the premise that the communication behavior in the production run should look similar to that in the training runs, after normalizing for the scale. The similarity should be observed at the granularity of the call sites, where the relevant calls are those that use the network socket API under the MPI library. So the localization process proceeds as follows. VRISHA matches up the call sites from the training runs and the production run in terms of their communication behavior and orders them by volume of communication. For example, in Figure 2.2, the matches are (call site ID in training, call site ID in production): (2, 3), (1, 2), (3, 1), (4, 4). The call site ID order does not have any significance, it is merely a map from the call stack to a numeric value. Now for the matching call sites, the call stacks should in the correct case be the same, indicating that the same control path was followed. A divergence indicates the source of the bug. VRISHA flags the points in the call stack in the production run where it diverges from the call stack in the training run, starting from the bottom of the call stack (i.e., the most recent call). The call stack notation is then translated back to the function and the line number in the source code to point the developer to where she needs to look for fixing the bug.

As would be evident to the reader, VRISHA determines an ordered set of code regions for the developer to examine. In some cases, the set may have just one element, namely, where there is only one divergent call site and only one divergence point within the call site. In any case, this is helpful to the developer because it narrows down the scope of where she needs to examine the code.

Retrieving Debugging Information. To facilitate the localization of bugs, we need certain debugging information in executables and shared libraries to map an address A in the call stack to function name and offset. In case such information is stripped off by the compiler, we also need to record the base address B of the object

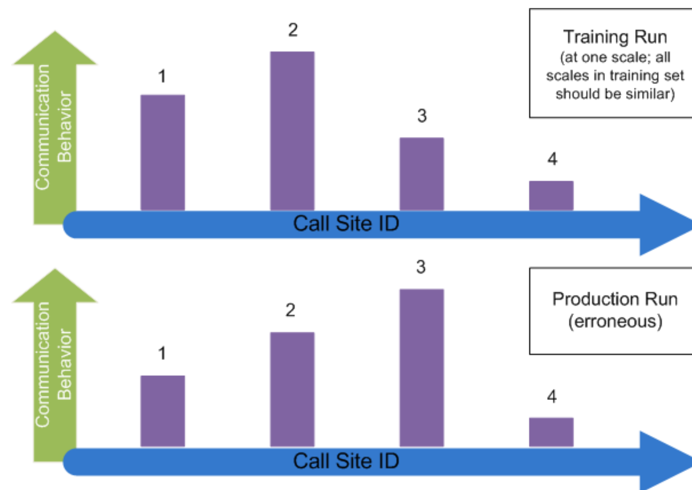

Figure 2.2. Example for demonstrating localization of a bug.

(executable and shared library) when it is loaded into memory so the offset within the object $A-B$ can be calculated and translated into the function name and the line number. This is done in an off-line manner, prior to providing the information to the developer for debugging, and can be done by an existing utility called `addr2line`.

Bugs that Cause Application Crash

It is trivial to detect an error caused by a bug that makes the application crash. However, localization of the root cause of such bugs is not as easy. For this, we use the localization technique for non-crashing bugs as the starting point and modify it. For comparison with the communication behavior of the training runs, we identify the point in execution corresponding to the crash in the production run. We then eliminate all call sites in the training run after that point from further processing. Then we follow the same processing steps as for the non-crashing bugs. One distinction is in the way we order the different call sites. The call site which is closest to the point at which the application crashed is given the highest priority. The intuition is that the propagation distance between the bug and the error manifestation is more likely to be small than large. Hence, we consider the call stack from the crashed application (in the production run) and compare that first to the call stack from the closest point in the training runs and flag the points of divergence, starting from the latest point of divergence.

2.4.5 Discussion

Our proposed design for Vrisha has some limitations, some of which are unsurprising, and some of which are somewhat subtle. The most obvious limitation is that Vrisha's ability to detect bugs is constrained by the choice of features. This limitation is imposed by the observational features and, surprisingly, the control features. If a bug manifests in a manner that does not change the value of an observational feature, Vrisha will be unable to detect it, as the data will not capture the abnor-

mal behavior. Hence, the observational features must be chosen with care to ensure that bugs are caught. Interestingly, the control features must be chosen carefully, as well. Our technique detects bugs when the expected behavior of a process (as determined by its control features) deviates from its observed behavior (as determined by its observational features). If an observational feature (in particular, the observational feature where a bug manifests) is uncorrelated with any of the control features, KCCA will ignore its contribution when constructing the projection functions and hence Vrisha will be unable to detect the bug.

Another limitation, unique to Vrisha's modeling technique, is that KCCA is sensitive to the choice of kernel functions. As an obvious example, if the kernel function were linear, KCCA would only be able to apply linear transformations to the feature sets before finding correlations, and hence would only be able to extract linear relationships. We mitigate this concern by using a Gaussian as our kernel function, which is effectively an infinite-degree polynomial.

Our localization strategy is also limited by the localization heuristics we use. First, we must infer a correspondence between the features of the buggy run and the features of the non-buggy runs. In the particular case of call-stack features, this presents problems as the call stacks are different for buggy vs. non-buggy runs. Our matching heuristic relies on the intuition that while the volume of data communicated at each call site is scale-determined, the *distribution* of that data is *scale invariant* (*i.e.*, is the same regardless of scale). This allows us to match up different call sites that nevertheless account for a similar proportion of the total volume of communication. While this heuristic works well in practice, it will fail if the distribution of communication is not scale-invariant. Another drawback of our localization heuristic is that if several call sites account for similar proportions of communication, we will be unable to localize the error to a single site; instead, we will provide some small number of sites as candidates for the error.

2.5 Evaluation

In this section, we evaluate the performance of Vrisha against real bugs in parallel applications. We use the MPICH2 library [9] and NAS Parallel Benchmark Suite [27] in these experiments. We have augmented the MPICH2 library with communication profiling functionality and reproduced reported bugs of MPICH2 to test our technique. The NAS Parallel Benchmark Suite 3.3 MPI version is used to evaluate the runtime overhead of the profiling component of Vrisha.

The experiments show that Vrisha is capable of detecting and localizing realistic bugs from the MPICH2 library while its runtime profiling component incurs less than 8% overhead in tests with the NAS Parallel Benchmark Suite. We also compare Vrisha with some of the most recent techniques for detecting bugs in parallel programs and illustrate that the unique ability of Vrisha to model the communication behavior of parallel programs as they scale up is the key to detect the evaluated bugs.

All the experiments are conducted on a 15-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.

2.5.1 Allgather Integer Overflow in MPICH2

This bug is an integer overflow bug which causes MPICH2 to choose a performance-suboptimal algorithm for Allgather (Figure 1.1). Allgather is a all-to-all collective communication function defined by the MPI standard, in which each participant node contributes a piece of data and collects contributions from all the other nodes in the system. Three algorithms [28] are employed to implement this function in the MPICH2 library and the choice of algorithm is conditioned on the total amount of data involved in the operation.

The total amount of data is computed as the product of three integer variables and saved in a temporary integer variable. When the product of the three integers overflows the size of an integer variable, a wrong choice of the algorithm to perform

Allgather is made and this results in a performance degradation, which becomes more significant as the system scales up. The bug is more likely to happen on a large-scale system, i.e., with a large number of processors, because one of the multiplier integer is the number of processes calling Allgather. For example, on a typical x86_64 Linux cluster with each process sending 512 KB of data, it will take at least 1024 processes to overflow an integer.

The bug has been fixed in a recent version of MPICH2 [29]. However, we found a similar integer overflow bug in Allgatherv, a variant of Allgather to allow varying size of data contributed by each participant, still extant in the current version of MPICH2 [30].

Detection and Localization

For the ease of reproducing the bug, we use a simple synthetic application that does collective communication using Allgatherv and run this application at increasing scales. The test triggers the bug in the faulty version of MPICH2 if the number of processes is 16 or more. Vrisha is trained with the communication profiles of the program running on 4 to 15 processes where the bug is latent and the communication distribution is not contaminated by the bug. We pictorially represent in Figure 2.3 the communication behavior that is seen in the application for two different sizes of the training system (4 and 8 processes) and one size of the production system where the bug manifests itself (16 processes). The X-axis is the different call sites and the Y-axis is the volume of communication, normalized to the scale of the system. The divergence in the communication behavior shows up with 16 processes where the pattern of communication behavior looks distinctly different. Vrisha successfully detects this bug as the correlation in the projection space for the 16-node system is low, as depicted in Figure 2.4. The Y-axis is the correlation, and a low value there indicates deviation from correct behavior. The detection cutoff is set to avoid false positives in the training set; this is sufficient to detect the error. Note that this bug

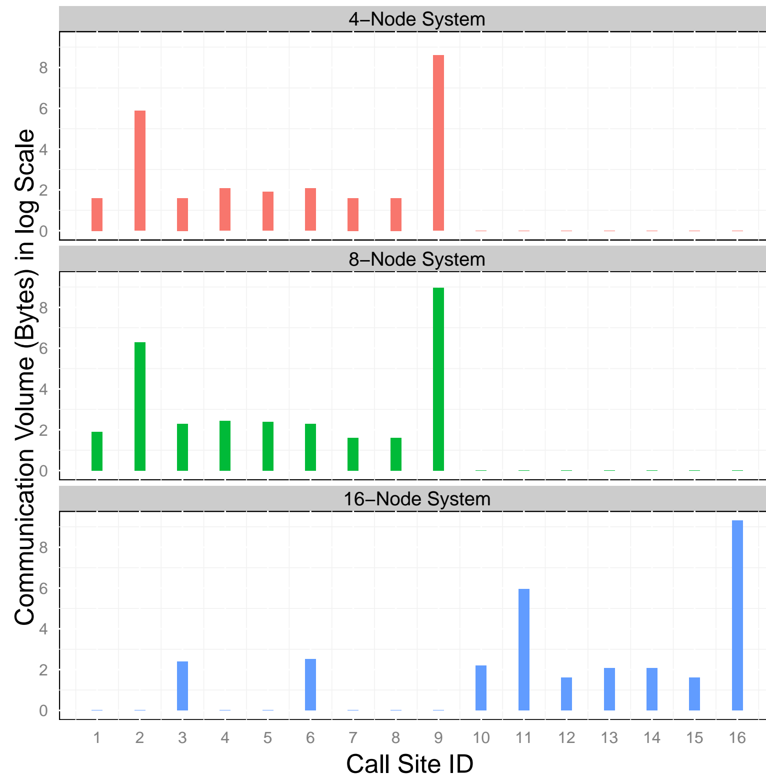

Figure 2.3. Communication behavior for the Allgather bug at two training scales (4 and 8 nodes) and production scale system (16 nodes). The bug manifests itself in the 16 node system (and larger scales)

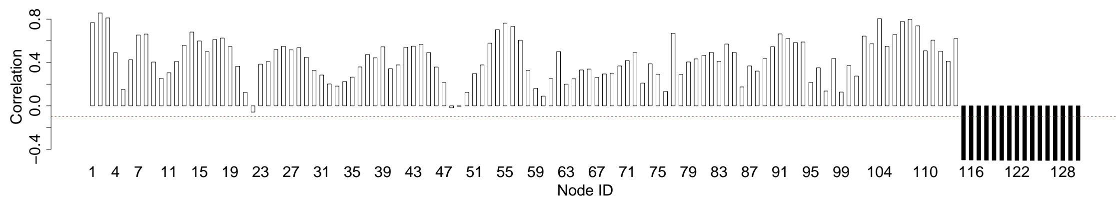

Figure 2.4. Correlation in the projection space using the KCCA-generated maps for systems of different scale. Vrisha is trained in 4- through 15-node systems (in light color) and tested in the buggy 16-node system (in dark color). The dashed line indicates the bug detection threshold.

```
Call Stack 9:                    Call Stack 16:
==============================   ==============================
MPID_nem_tcp_send_queued+0x1cc   MPID_nem_tcp_send_queued+0x1cc
MPID_nem_tcp_connpoll+0x3a3      MPID_nem_tcp_connpoll+0x3a3
MPID_nem_network_poll+0x1e       MPID_nem_network_poll+0x1e
MPIDI_CH3I_Progress+0x2ab        MPIDI_CH3I_Progress+0x2ab
MPIC_Wait+0x89                   MPIC_Wait+0x89
MPIC_Sendrecv+0x246              MPIC_Sendrecv+0x246
MPIR_Allgatherv+0x6a2  <------>  MPIR_Allgatherv+0x17fd
PMPI_Allgatherv+0x1243           PMPI_Allgatherv+0x1243
main+0x14c                       main+0x14c
__libc_start_main+0xf4           __libc_start_main+0xf4
```

Figure 2.5. Call stacks for the correct case (call stack 9, in the training system) and the erroneous case (call stack 16, in the production system).

affects *all* processes at systems of size 16 or higher and therefore, many previous statistical machine learning techniques will not be able to detect this because they rely on majority behavior being correct.

Following our bug localization scheme, VRISHA compares the normal and the faulty distributions in Figure 2.3. The call site 9 from the training run is matched up with call site 16 from the production run and this is given the highest weight since the communication volume is the largest (90% of total communication). We show the two call stacks corresponding to these two call sites in Figure 2.5. The deepest point in the call stack, *i.e.*, the last called function, is shown at the top in our representation. A comparison of the two call stacks reveals that the faulty processes take a detour in the function `MPIR_Allgatherv` by switching to a different path. The offset is mapped to line numbers in the `MPIR_Allgatherv` function and a quick examination shows that the fault lies in a conditional statement that takes the wrong branch due to overflow.

Detection in an application

The above experiment detects bugs in an MPI library, but *at the application level.* The calling-context features used by VRISHA are specified in relation to the application using the MPI library, and hence provide information regarding the specific line in the application where the bug manifested (*viz.* the last two lines of the calling contexts shown in Figure 2.5).

Because the features that Vrisha uses for detection (and diagnosis) are specified at the application level, there is little difference in identifying and finding a bug in a library and finding a bug in an application. VRISHA's bug reports are in terms of call stacks that include application code, and hence localize bugs to regions of the application source. According to our diagnosis heuristic, if the erroneous call stack differs from the bug-free call stack in a call that is part of the application, the bug is likely in the application itself, while if (as in our example) the difference lies in a library function, the bug is likely in the library.

We do note that one difference between finding a library bug using an application harness (as we did in the above experiment) and finding a bug in a full-scale application is that in the latter case VRISHA may need to track many more features (as the application will make many more MPI calls), most of which will not exhibit the bug. When presented with larger, largely correct, feature sets, it can be harder to detect buggy behavior.

To determine whether the existence of numerous non-buggy features impacts VRISHA's ability to detect bugs such as the Allgather bug, we synthetically injected the bug into an otherwise non-buggy run of CG from the NAS parallel benchmarks [31][1]. We performed this injection by collecting call-stack features for CG at various scales (51 features) and appending the Allgather features (19 features, only one of which exhibits the bug) from our test harness at the corresponding scale, thus simulating an

[1]We study an injected bug rather than a real-world bug due to the dearth of well-documented bugs in MPI applications. Most documented non-crashing bugs (both correctness and performance) appear to occur in libraries. This is likely because scientific applications typically do not maintain well-documented bug reports or provide change logs indicating which bugs are fixed in a new version.

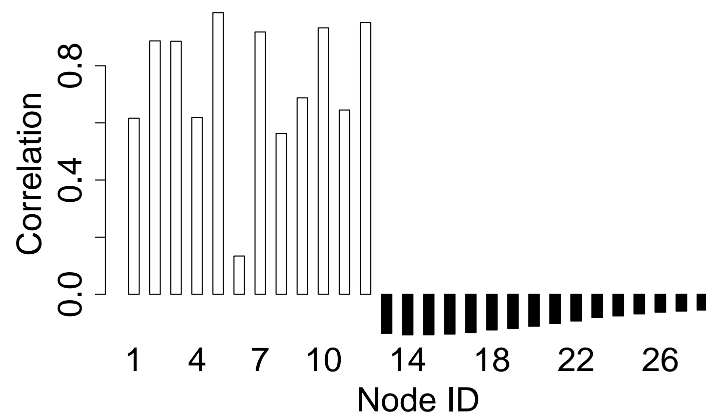

Figure 2.6. Correlation of training and testing nodes for CG with synthetically injected bug. VRISHA is trained on 4- and 8-node systems (in light color) and tested on 16 nodes (in dark color).

application with a large number of non-buggy call-sites and relatively few locations that exhibit the bug. Figure 2.6 shows the results of running VRISHA trained on 4- and 8-node runs (where the Allgather features did not display the bug) and tested on a 16-node execution (where the Allgather features exhibited the bug). We see that the correlations of the testing nodes are notably lower than those of the training nodes (and in particular, are the only nodes with negative correlation). Hence, the testing nodes would be flagged as buggy by VRISHA's detection heuristic. Thus even in a realistic scenario where most features do not show the effects of the bug, VRISHA is able to detect the bug.

Comparison with Previous Techniques

This bug cannot be detected by previous techniques [5–7,32] which capture anomalies by comparing the behavior of different processes in the same sized system. This is due to the fact that there is no statistically significant difference among the behaviors of processes in the 16-node system. As the bug degrades the performance of Allgather but no deadlock is produced, those techniques targeted at temporal progress [3] will not work either. Finally, since there is no break in the message flow of Allgather as all messages are delivered eventually but with a suboptimal algorithm, FlowChecker [8] will not be able to detect this bug. Therefore, VRISHA is a good complement to these existing techniques for detecting subtle scale-dependent bugs in parallel programs.

2.5.2 Bug in Handling Large Messages in MPICH2

This bug [33] was first found by users of PETSc [34], a popular scientific toolkit built upon MPI. It can be triggered when the size of a single message sent between two physical nodes (not two cores in the same machine) exceeds 2 gigabytes. The MPICH2 library crashes after complaining about dropped network connections.

It turns out that there is a hard limit on the size of message can be sent in a single iovec struct from the Linux TCP stack. Any message that violates this limit

would cause socket I/O to fail as if the connection were dropped. The most tricky part is that it would manifest in the MPI level as a MPICH2 bug to the application programmers.

Detection and Localization

Since we have no access to the original PETSc applications that triggered this bug, we compromise by using the regression test of the bug as our data source to evaluate VRISHA against this bug. The regression test, called `large_message`, is a simple MPI program which consists of one sender and two receivers and the sender sends a message a little bit larger than 2GB to each of the two receivers. We adapt `large_message` to accept an argument which specifies the size of message to send instead of the hard-coded size in the original test so we can train VRISHA with different scales of input. Here, "scale" refers to the size of data, rather than the meaning that we have been using so far—number of processes in the system. This example points out the ability of VRISHA to deduce behavior that depends on the size of data and to perform error detection and bug localization based on that. We first run the regression test program with 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, and 1GB to get the training data set and then test with the 2GB case. The distributions of communication over call sites of a representative process in each case of 512MB, 1GB, and 2GB are shown in Figure 2.7.

Since the bug manifests as a crash in the MPICH2 library, there is nothing left to be done with the detection part. We are going to focus on explaining how we localize the bug with the guidance from VRISHA. First of all, as discussed in Section 2.4.4, we need the stack trace at the time of the crash. This is shown on the right part of Figure 2.8. In fact, the MPICH2 library exits with error message "socket closed" at function `MPID_nem_tcp_send_queued`. Comparing with all the five normal call stacks shown in Figure 2.7 (i.e., obtained from training runs), we find call stack 5 is almost a perfect match for the crash stack trace from MPICH2 except for two static

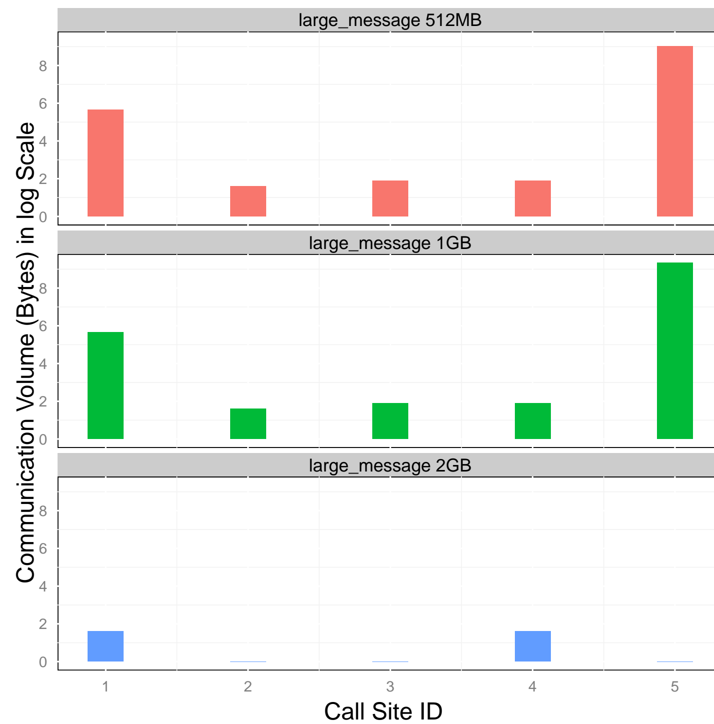

Figure 2.7. Communication behavior for the large message bug at two training scales (512 MB and 1 GB) and production scale system (2 GB). The bug manifests itself in data sizes of 2 GB and larger.

```
Call Stack 5                        Crash Stack from MPICH2
==============================      =============================
MPID_nem_tcp_send_queued+0x132      MPID_nem_tcp_send_queued
(Unknown static function)           state_commrdy_handler
MPID_nem_tcp_connpoll+0x3d8         MPID_nem_tcp_connpoll
MPID_nem_network_poll+0x1e          MPID_nem_network_poll
(Unknown static function)           MPID_nem_mpich2_blocking_recv
MPIDI_CH3I_Progress+0x1d8           MPIDI_CH3I_Progress
MPI_Send+0x8ff                      MPI_Send
main+0x121)
__libc_start_main+0xf4)
```

Figure 2.8. Call stacks from a normal process (left) and at the point of crash due to large-sized data. Error message "socket closed" reported by MPICH2 at `MPID_nem_tcp_send_queued` helps localize the bug.

functions whose names are optimized out by the compiler. The first divergent point in the crash trace is at `MPID_nem_tcp_send_queued`, which is where the bug lies.

To this point, we have localized the bug to a single function. The next step depends on the properties of each specific bug. In practice, most applications implement some error handler mechanism that provide useful error messages before exiting. In the case of this bug, one only needs to search for the error message "socket closed" inside the function `MPID_nem_tcp_send_queued` and would find that it is the failure of `writev` (a socket API for sending data over the underlying network) that misleads MPICH2 to think the connection is closed. In this case, Vrisha only has to search within a single function corresponding to the single point of divergence. In more challenging cases, Vrisha may have to search for the error message in multiple functions. In the absence of a distinct error message, Vrisha may only be able to provide a set of functions which the developer then will need to examine to completely pinpoint the bug.

Comparison with Previous Techniques

Most previous techniques based on statistical rules will not be helpful in localizing this bug because they lack the ability to derive scale-parametrized rules to provide role model to compare with the crash trace. All the processes at the large data sizes suffer from the failure and therefore contrary to the starting premise of much prior work, majority behavior itself is erroneous. However, FlowChecker is capable of localizing this bug since the message passing intention is not fulfilled in `MPID_nem_tcp_send_queued`.

2.5.3 Performance Measurement

In this section, we assess the various overheads of Vrisha. The primary overhead is in Vrisha's runtime profiling. We looked at the profiling overheads introduced by Vrisha in five applications from the NAS Parallel Benchmark Suite [31], `CG`, `EP`, `IS`, `LU` and `MG`. Each application is executed 10 times on 16 processes with the class A inputs, and the average running time is calculated to determine Vrisha's profiling overhead. Figure 2.9 shows that the average overhead incurred by profiling is less than 8%.

The other costs of Vrisha are its modeling time (how long it takes to build the KCCA model) and its detection time (how long it takes to process data from the production run to perform bug detection). Figures 2.10(a) and (b) show the modeling time and detection time, respectively, for different problem sizes, averaged over 100 runs. We see that Vrisha takes a fraction of a second to both model behavior and detect bugs. Detection takes less time as it uses pre-computed project vectors.

2.5.4 Model Selection and False Positive Rate

This section evaluates the impact of model selection of the KCCA method on the false positive rate of Vrisha. Because of the lack of a publicly-available comprehen-

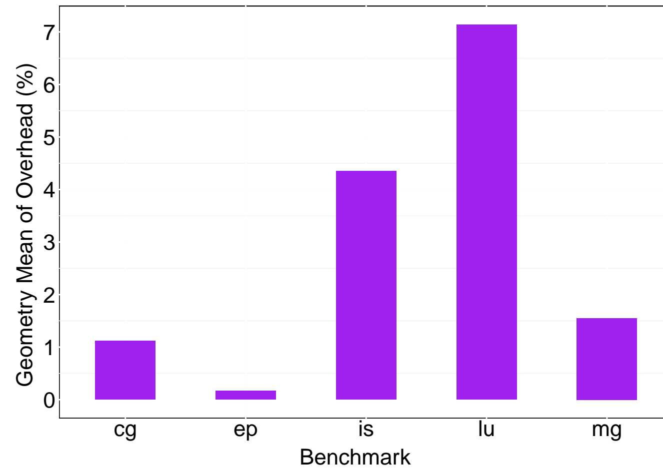

Figure 2.9. Overhead due to profiling in VRISHA for NASPAR Benchmark applications.

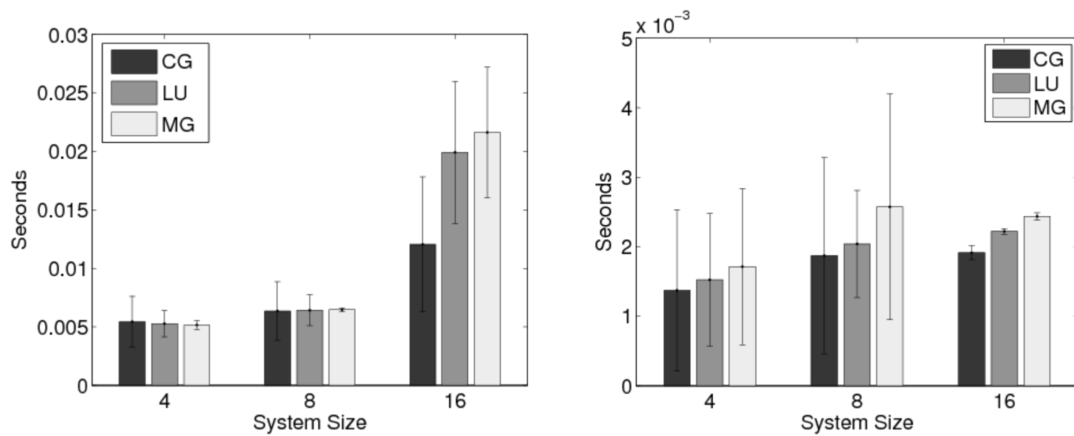

Figure 2.10. Modeling and detection time for CG, LU and MG on 4-, 8- and 16-node systems.

Table 2.1
Sensitivity of false positive rate to model parameters in VRISHA

Parameter	Range	False Positive
N_{comps}	$1, \cdots, 10$	2.85%, 3.16%
γ	$2^{-20}, \cdots, 2^{0}$	2.32%, 3.25%
σ_x	$2^{-20}, \cdots, 2^{20}$	1.79%, 8.19%
σ_y	$2^{-20}, \cdots, 2^{20}$	2.18%, 4.01%

sive database of bugs in parallel programs, we have no way to conduct a study of false negative rate, therefore we follow the practice of previous researchers of focusing on the fault positive rate of our model by considering error-free applications.

The following parameters in the KCCA model, N_{comps}, γ, σ_x, σ_y are measured using five-fold cross validation on the training data from Section 2.5.1. We varied each parameter over a range, given in Table 2.1, while holding the other parameters constant. The table gives the range of false positives found over each parameter's range. As we see, N_{comps}, γ and σ_y do not significantly affect the false positive rate while σ_x has more impact taking the false positive to 8.2% in the worst case. The impact of σ_x on the performance of VRISHA can also be interpreted as such that even for the worst case of σ_x VRISHA could still detect bug with an accuracy of around 91.8% in most cases. Overall, the KCCA model used in VRISHA is not very sensitive to parameter selection which makes it more accessible to users without solid background in machine learning.

2.6 Summary

In this chapter, we introduced VRISHA, a framework for detecting bugs in large-scale systems using statistical techniques. While prior work based on statistical techniques relied on the availability of error-free training runs at the same scale as production runs, it is infeasible to use full-scale systems for development purposes.

Unfortunately, this means that prior bug-detection techniques are ill-suited to dealing with bugs that only manifest at large scales. Vrisha was designed to tackle precisely these challenging bugs. By exploiting *scale-determined* properties, Vrisha uses kernel canonical correlation analysis to build models of behavior at large scale by generalizing from small-scale behavioral patterns. Vrisha incorporates heuristics that can use these extrapolated models to detect and localize bugs in MPI programs. We studied two bugs in the popular MPICH2 communication library that only manifest as systems or inputs scale. We showed that Vrisha could automatically build sufficiently accurate models of large-scale behavior such that its heuristics could detect and localize these bugs, without ever having access to bug-free runs at the testing scale. Furthermore, Vrisha is able to find bugs with low instrumentation overhead and low false positive rates.

To this point, we have evaluated Vrisha with test cases at slightly larger scales than the training inputs, validating our approach, but leaving open the question of how much larger the scale of the test system can be compared to the training runs. This study is a prime target for future work. Further, we will consider other kinds of bugs beyond communication-related bugs, investigate more fully the scaling behavior with respect to data sizes, and evaluate the scalability of the detection and the localization procedures.

3 Abhranta: LOCATING BUGS THAT MANIFEST AT LARGE SYSTEM SCALES

3.1 Overview

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

Figure 3.1 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.

3.1.1 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 [14].

For a given application run, Abhranta collects two types of features: *control* features and *observational* features. Control features are a generalization of scale: they

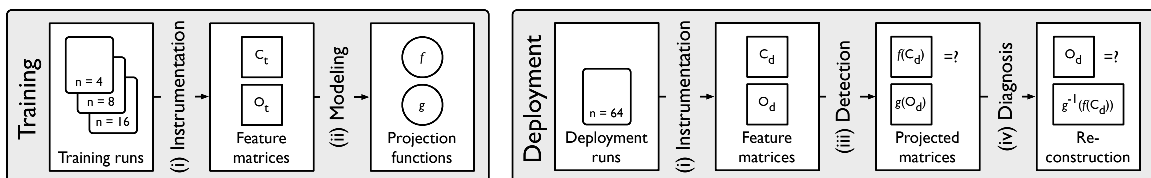

Figure 3.1. Overview of Abhranta architecture

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 [35] 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 3.3.2) 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.

3.1.2 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 $\mathbf{C}$, while the observational features are collected into a matrix $\mathbf{O}$, with the property

that row i of $\mathbf{C}$ and $\mathbf{O}$ 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* [22, 23] to construct two *projection* functions, f and g, that transform $\mathbf{C}$ and $\mathbf{O}$, 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 3.2 explains how this modified model can be used to predict the behavior of large-scale runs.

3.1.3 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.

3.1.4 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, 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 3.2 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.

3.2 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

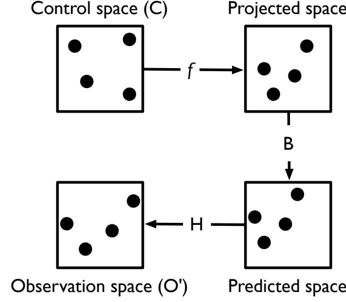

Figure 3.2. Process to derive reconstructed observations (O') from control features (C). f is a non-linear transformation, while B and H are linear.

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 3.3 confirms that this restricted modeling space does not significantly reduce ABHRANTA's detectability.

ABHRANTA's reconstruction strategy is inspired by the preimage reconstruction method presented by Feng *et al.* [16]. Figure 3.2 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):

$$\min_B \sum_{i=1}^{N} \left\| BP^i_C - P^i_{O'} \right\|^2$$

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):

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

3.3 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.

3.3.1 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 [29], 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.

Localization: We next evaluate ABHRANTA's ability to localize the ALLGATHER bug by reconstructing the expected behavior of the 64-process execution. Figure 3.3 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.

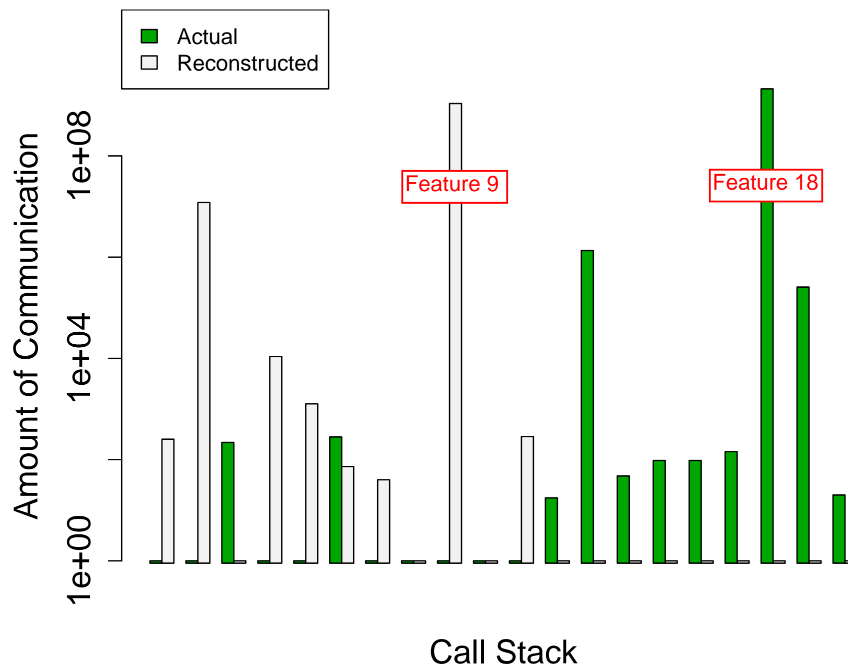

Figure 3.3. Reconstructed vs. actual buggy behavior for ALLGATHER

3.3.2 Case Study 2: Transmission's DHT

Transmission is a popular P2P file sharing application on Linux platforms. The bug [36] 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 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 3.4. The rank ordering of deviant features highlights Feature 12, which corresponds to the call to a

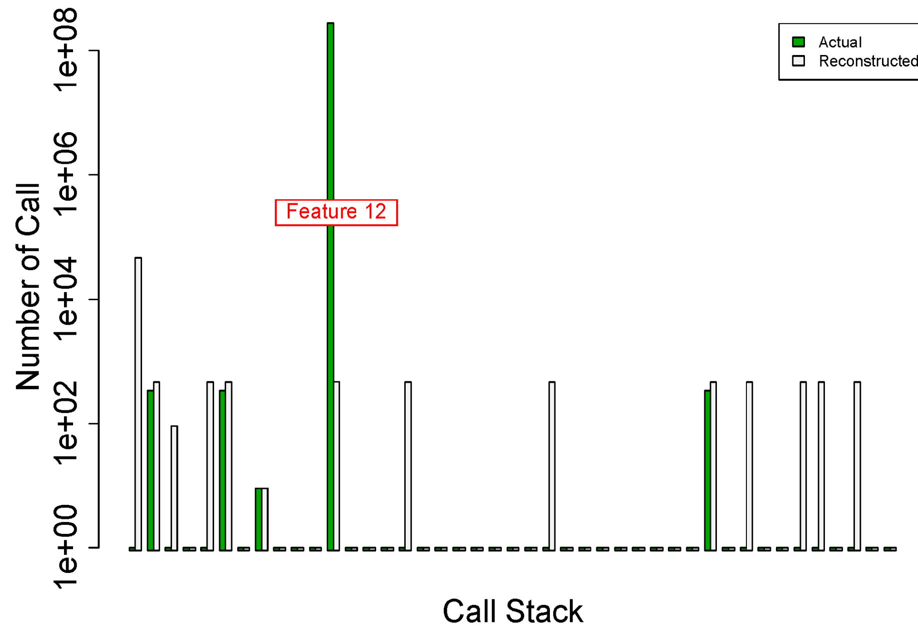

Figure 3.4. Reconstructed vs. actual buggy behavior for Transmission DHT

libc function `strtol`, only a few lines away from the root cause of the bug in this several-hundred-line function.

3.4 Summary

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) [37] 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.

4 WuKong: AUTOMATICALLY DETECTING AND LOCALIZING BUGS THAT MANIFEST AT LARGE SYSTEM SCALES

4.1 Overview

At a high level, WuKong's modeling, detection and localization approach consists of the following components.

(a) Model Building During training, control and observational features are collected at a *series* of small scales. These features are used to construct per-feature regression models that capture non-linear relationships between system scale (the control features) and program behavior (the observational features). Sections 4.2.1 and 4.2.2 describe WuKong's modeling strategy in more detail.

(b) Feature Pruning Features whose behavior is inherently unpredictable (*e.g.*, non-deterministic, discontinuous or overly-complex) cannot be accurately modeled by WuKong's regression models. Because model failures can complicate detection and localization (poorly modeled features may deviate significantly from predictions, triggering false positives), WuKong uses a novel, cross-validation-based *feature pruning* strategy to improve the accuracy of detection and localization. Section 4.3 details this approach.

(c) Bug Diagnosis WuKong can detect and diagnose bugs in large-scale production runs by using its models to predict what behavior *should have been* at that large scale. Intuitively, a feature whose predicted value is significantly different from its actual value is more likely to be involved in the bug than a feature whose predicted value is close to its actual value. A test run is flagged as buggy if any one of its features has a significant deviation between its observed and the predicted values. To

locate a bug, WuKong simply ranks features by the relative difference between the predicted value and the actual value, and presents the ordered list to the programmer. Section 4.4 elaborates further.

4.2 Modeling Program Behavior

This section describes WuKong's modeling technique. The key component is the construction of per-feature models that capture the relationship between the control features and the value of a particular observational feature. These models can be used to predict the expected observational features for production runs at a scale larger than any seen during training. As a result, the correct behavior (observational feature values) of large scale runs can be reconstructed based on the prediction of the model, and this information can be used for detection and localization.

4.2.1 Model Building

WuKong models application behavior with a collection of base models, each of which characterizes a single observational feature. The base model is an instance of multiple regression where multiple predictors are considered. Specifically, the base model for each observational feature considers all control features as predictor variables, and the value of the observational feature as the response variable.

Suppose Y is the observational feature in question, and X_i for $i = 1 \ldots N$ are the N control features. We note that a base model of the form:

$$Y = \beta_0 + \sum_{i=1}^{N} \beta_i \cdot X_i \tag{4.1}$$

is not sufficient to capture complex relationships between control features and program behavior. It does not account for higher-order relationships between behavior and scale (consider the many algorithms that are $O(n^2)$), and it does not capture interaction between control features (consider a program location inside a doubly-nested loop where the inner loop runs X_i times and the outer loop runs X_j times).

To account for this, we apply a logarithmic transform on both the control features and the observational feature, yielding the following base model:

$$\log(Y) = \beta_0 + \sum_{i=1}^{N} \beta_i \log(X_i) \tag{4.2}$$

The refined model transforms multiplicative relationships between the variables into additive relationships in the model, allowing us to capture the necessary higher order and interactive effects.

The multiple regression problem is solved by the ordinary least squares method. The solution is given by a vector of coefficients $\beta_0 ... \beta_N$:

$$\operatorname*{arg\,min}_{\beta_0,\ldots,\beta_N} || \log(Y) - \sum_{i=1}^{N} \beta_i \log(X_i) - \beta_0||^2 \tag{4.3}$$

The resulting model achieves the best fit for the training data, *i.e.,* it minimizes the mean squared prediction error of Y.

WuKong limits the regression model to linear terms as our empirical results suggest linear terms are enough to capture the scaling trend of most observational features. Although more complex terms, (*e.g.*, high order polynomials, cross products, etc.) might result in better fit for the training data, they also have a higher risk of overfitting and generalize poorly for the test data.

Since each feature gets its own base model, we do not face the same problem as in Vrisha [14], where a single model must be "reverse-engineered" to find values for individual observational features. Instead, WuKong can accurately predict each feature in isolation. Moreover, the linear base models leads to more stable extrapolation at large scales, thanks to the lack of over-fitting.

4.2.2 Base Model Customization

One model does not fit all observational features. Observational features usually scale at differing speeds and the scaling trends of different features may be vastly different. Furthermore, some observational features may depend only on a subset

of all control features. Therefore, throwing all control features into the base model for every observational feature may result in over-fitting the data, and lower the prediction accuracy for such features. To handle this problem, we need to customize the base model for each individual observational feature based on the training data. Through the customization process, we want to determine the particular formula used for modeling each individual feature, *i.e.*, which control features should be included as predictor variables in the model. Essentially, we want the simplest possible model that fits the training data; if making the model more complex only yields a marginal improvement in accuracy, we should prefer the simpler model.

WuKong's model customization is based on the Akaike Information Criterion (AIC) [38], a measure of relative goodness of fit in a statistical model given by:

$$AIC = -2\ln(L) + 2k \tag{4.4}$$

where L is the likelihood of the statistical model, which measures the goodness of fit, and k is the number of parameters in the model, which measures the model complexity. Unlike the more common approach to measuring model accuracy, the coefficient of determination R^2, AIC penalizes more complex models (intuitively, a more complex model must provide a much better fit to be preferred to a simpler model). This avoids over-fitting and ensures that WuKong produces appropriately simple models for each observational feature.

In a program with N control features, there are 2^N possible models that match the form of Equation 4.2. If N is small, it is feasible to conduct an exhaustive search through every model configuration to find the appropriate model for each observational feature. However, if N is large, the configuration space might be prohibitively large, making an exhaustive search impractical. In such a scenario, WuKong uses a greedy, hill-descending algorithm [39]. We begin with a model that includes all control features. At each step, WuKong considers all models one "move" away from the current model: all models with one fewer control feature than the current model and all models with one more control feature than the current model. Of the candidate models, WuKong picks the one with the lower AIC and makes it the current

model. The process continues until no "move" reduces the AIC compared to the current model. For any single observational feature, the result of model customization is a model that includes a subset of control features that are most relevant to that particular observational feature.

4.3 Feature Selection and Pruning

As described in Section 4.5, WuKong uses as its observational features all conditionals in a program, augmented with the dynamic calling context in which that conditional executed. Each time a particular conditional is evaluated, WuKong increments the value of the appropriate feature.

The logarithmic model in Section 4.2.1 allows us to readily compute the relative prediction error for a given feature, which we require to identify faulty features (see Section 4.4). The model built for each observational feature, i, is used to make prediction Y_i' for what the value of that feature *should have been* if the program were not buggy. WuKong then compares Y_i' to the observed behavior, Y_i and calculates the *relative prediction error* of each observational feature, using the approach of Barnes *et al.* [40]:

$$E_i = |e^{\log(Y_i') - \log(Y_i)} - 1| \tag{4.5}$$

Note that a constant prediction of 0 for any feature will result in relative reconstruction error of 1.0; hence, relative errors greater than 1.0 are a clear indication of a poorly-predicted feature.

Unfortunately, not all observational features can be effectively predicted by the regression models of WuKong, leading to errors in both detection and diagnosis. There are two main reasons why an observational feature can be problematic for WuKong. One is that the feature value is non-deterministic: a conditional whose outcome is dependent on a random number, for example. Because such features do not have deterministic values, it is impossible to model them effectively. Recollect

that WuKong relies on the assumption that any observational feature is determined by the control features.

A second situation in which a feature cannot be modeled well is if its value is dependent on characteristics not captured by the control features. These could be confounding factors that affect program behavior such as OS-level interference or network congestion. Another confounding factor is *data-dependent* behavior. WuKong uses as its control features scale information about the program, such as number of processes/threads or input data size. If a program's behavior is determined by the *contents* of the input, instead, WuKong does not capture the appropriate information to predict a program's behavior.

WuKong's reconstruction techniques can be thrown off by unpredictable program behavior: the behavioral model will be trained with behavior that is not correlated with the control features, and hence spurious trends will be identified. Note that even a small number of such problematic features can both introduce false positives and seriously affect the accuracy of localization. If WuKong makes a prediction based on spurious trends, even non-buggy behavior may disagree with the (mis)prediction, leading to erroneously detected errors. Second, even if an error is *correctly* detected, because reconstruction will be based on bogus information, it is likely that the reconstruction errors for such problematic features will be fairly high, pushing the true source of errors farther down the list. The developer will be left investigating the sources of these problematic features, which will not be related to any bug.

We note, however, that if we had a means of removing bad features, we could dramatically improve localization performance. Because a bad feature's appearance at the top of WuKong's roadmap occurs far out of proportion to its likelihood of actually being the buggy feature, simply filtering it from the feature set will negatively impact a small number of localization attempts (those where the filtered feature is the source of the bug) while significantly improving all other localization attempts (by removing spurious features from the roadmap). Therefore, WuKong employs a

feature filtration strategy to identify hard-to-model features and remove them from the feature list.

To eliminate bad features, WuKong employs cross validation [39]. Cross validation uses a portion of the training data to test models built using the remainder of the training data. The underlying assumption is that the training data does not have any error. More specifically, WuKong employs k-fold cross-validation. It splits the original training data by row (*i.e.* by training run) into k equal folds, treats each one of the k folds in turn as the test data and the remaining $k-1$ folds as the training data, then trains and evaluates a model using each of the k sets of data. For each cross-validation step, we compute the relative reconstruction error of each feature X_i for each of the (current) test runs.

If a particular feature cannot be modeled well during cross validation, WuKong assumes that the feature is unpredictable and will filter it out from the roadmaps generated during the localization phase. WuKong's pruning algorithm operates as follows.

WuKong has a *pruning threshold* parameter, x, that governs how aggressively WuKong will be when deciding that a feature is unpredictable. Given a pruning threshold x, a feature is only kept if it is well-predicted in at least x% of the training runs during cross-validation. In other words, WuKong will *remove* a feature if more than $(100-x)$% of the runs are poorly predicted (*i.e.*, have a relative reconstruction error less than 1.0). For example, if the pruning threshold is 25%, then WuKong prunes any feature for which more than 75% of its (relative) errors are more than than 1.0. The higher x is, the more aggressive the pruning is. If x is 0, then no pruning happens (no runs need be well predicted). If x is 100, then pruning is extremely aggressive (there can be no prediction errors for the feature during cross-validation). Typically, x is set lower than 100, to account for the possibility of outliers in the training data.

Some discontinuous features are hard to eliminate with cross-validation because only a few runs during training have problematic values. Hence, in addition to

cross-validation-based feature pruning, WuKong also employs a heuristic to detect potentially-discontinuous observational features based on the following two criteria [41]:

- Discrete value percentage: defined as the number of unique values as a percentage of the number of observations; Rule-of-thumb: $< 20\%$ could indicate a problem.
- Frequency ratio: defined as the frequency of the most common value divided by the frequency of the second most common value; Rule-of-thumb: > 19 could indicate a problem.

If both criteria are violated, the feature has too-few unique values and hence is considered potentially discontinuous. These features are pruned from the feature set, and are not used during detection or diagnosis.

It is important to note that the feature pruning performed by WuKong is a complement to the model customization described in the prior section. Model customization prunes the control features used to model a particular observational feature. In contrast, feature pruning filters the observational features that cannot be effectively modeled by *any* combination of the control features.

4.4 Debugging Programs at Large Scales

Once the models are built and refined, as described in the previous section, WuKong uses those models to debug programs at large scales. This proceeds in two steps, detection and diagnosis, but the basic operation is the same. When a program is run at large scale, WuKong uses its models to *predict* what each observational feature should be, given the control features of the large-scale run[1]. In other words, WuKong uses its models to predict the *expected* behavior of the program at

[1]Note that WuKong makes the crucial assumption that the control features for production runs are correct; this is reasonable since control features tend to be characteristics of program inputs and arguments.

large scale. These predictions are then used to detect and diagnose bugs, as described below.

4.4.1 Bug Detection

WuKong detects bugs by determining if the behavior of a program execution is inconsistent with the scaling trends captured by the behavioral model. If any feature's observed value differs significantly from its predicted value, WuKong declares a bug. The question then, is what constitutes "significantly"? WuKong sets detection thresholds for flagging bugs as follows.

For each observational features, WuKong tracks the reconstruction errors for that feature across all the runs used in cross validation during training (recall that this cross validation is performed for feature pruning). For each feature, WuKong determines the maximum relative error (Equation 4.5) observed during cross validation, and uses this to determine the detection threshold. If M_i is the maximum relative reconstruction error observed for feature i during training, WuKong computes E_i, the relative reconstruction error *for the test run*, and flags an error if

$$E_i > \eta M_i \tag{4.6}$$

where η is a tunable *detection threshold* parameter. Note that η is a global parameter, but the detection threshold for a given feature is based on that feature's maximum observed reconstruction error, and hence each feature has its own detection threshold. What should η be? A lower detection threshold makes flagging errors more likely (in fact, a detection sensitivity less than 1 means that even some known non-buggy training runs would be flagged as buggy), while a higher detection threshold makes flagging errors less likely ($\eta \geq 1$ means that no training run would have been flagged as buggy).

We note that in the context of bug detection, false positives are particularly damaging: each false positive wastes the programmer's time searching for a non-existent

bug. In contrast, false negatives, while problematic (a technique that detects no bugs is not particularly helpful!), are less harmful: at worst, the programmer is no worse off than without the technique, not knowing whether a bug exists or not. As a result of this fundamental asymmetry, we bias η towards false negatives to prevent false positives: η should always be set to a greater-than-one constant. We use $\eta = 1.15$ in our experiments; Section 4.6.1 shows how changing η affects false positive and negative rates.

4.4.2 Bug Localization

When a bug is detected, to provide a "roadmap" for developers to follow when tracking down the bug, WuKong ranks all observational features by relative error; the features that deviate most from the predicted behavior will have the highest relative error and will be presented as the most likely sources for the bug. WuKong produces the entire ranked list of erroneous features, allowing programmers to investigate all possible sources of the bug, prioritized by error. Note that while deviant features are likely to be involved with the bug, the most deviant features may not actually be the source of the bug. Buggy behavior can propagate through the program and lead to other features' going awry, often by much larger amounts than the initial bug (a "butterfly effect"). Nevertheless, as we show in Section 4.6's fault injection study, the majority of the time the statement that is the root cause of the bug appears at the top of the roadmap.

4.4.3 Sources and Types of Detection and Diagnosis Error

WuKong has two primary configuration parameters that affect the error rates of both its detection scheme and its diagnosis strategy: the feature pruning parameter x, and the detection threshold parameter η. This section describes how these parameters interact intuitively, while the sensitivity studies in Section 4.6 explore these effects empirically. Figure 4.1 shows the possible outcomes and error types that can occur

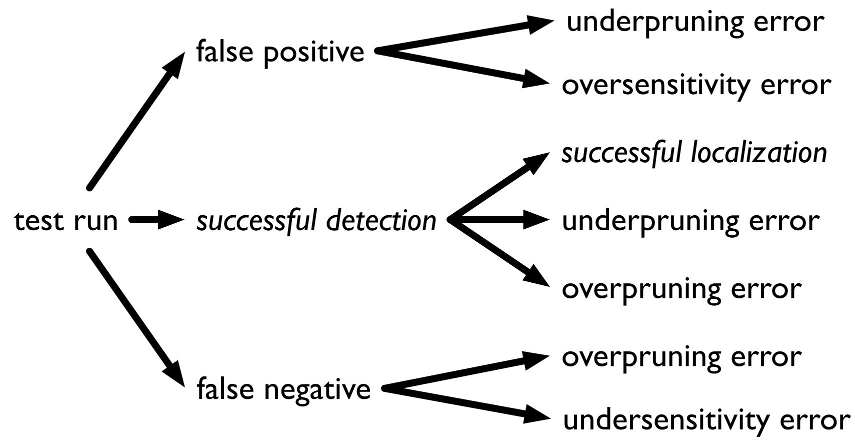

Figure 4.1. Possible outcomes and errors when using WuKong to detect and diagnose bugs.

when WuKong is applied to a test run; we discuss these error sources in more detail below.

False positives The most insidious error, from a developer productivity standpoint, is a false positive (an erroneous detection of an error in a bug-free run): if WuKong throws up a false positive, the developer can spend hours searching for a bug that does not exist. False positives can arise from two sources: *feature underpruning* and *detection oversensitivity*. Feature underpruning occurs when the pruning threshold x is set too low. By keeping too many features, including those that cannot be modeled effectively, WuKong may detect an error when a poorly-modeled feature leads to a bad prediction, even if the observed feature value is correct. Detection oversensitivity happens when the detection threshold η is too low, which increases the model's sensitivity to slight variations and deviations from the predicted value, increasing the likelihood of a false positive.

If a test run results in a false positive, it is hard to pinpoint the source of the error, as both oversensitivity and underpruning lead to correct features' being mispredicted by WuKong. Nevertheless, if the erroneous feature was *never* mispredicted during training (*i.e.*, it would not have been pruned even if the pruning threshold were 100%), then oversensitivity is likely at fault.

False negatives False negatives occur when a buggy run is incorrectly determined to be correct by WuKong, and can occur for two reasons (unsurprisingly, these are the opposite of the issues that result in false positives): *feature overpruning* and *detection undersensitivity*. If *too many* features are pruned, then WuKong tracks fewer features, and hence observes less program behavior. Because WuKong can only detect a bug when it observes program behavior changing, tracking fewer features makes it more likely that a bug will be missed. If the detection threshold is raised, then the magnitude of reconstruction error necessary to detect a bug is correspondingly higher, making WuKong less sensitive to behavior perturbations, and hence less likely to detect a bug.

For false negatives, overpruning is the culprit if the error manifested in a pruned feature, while undersensitivity is the issue if the error manifested in a tracked feature, but WuKong did not flag the error.

Diagnosis errors Even after WuKong correctly detects a bug in a program, it may not be able to successfully localize the bug (here, successful localization means that the bug appears within the top k features suggested by WuKong). The success of localization is primarily driven by x, the feature pruning threshold. Interestingly, there are two types of localization errors, one of which is caused by overpruning, and the other by underpruning. If x is too low, and features are underpruned, then many poorly-modeled features will be included in WuKong's model. These poorly modeled features can have high reconstruction errors, polluting the ranked list of features, and pushing the true error farther down the list. Conversely, if x is too high and the feature set is overpruned, the erroneous feature may not appear *anywhere* in the list. It may seem weird that the erroneous feature could be pruned from the feature set even while WuKong detects the bug. This is due to the butterfly effect discussed earlier; even though the buggy feature is not tracked, features that are *affected* by the bug may be tracked, and trigger detection.

For detection errors, it is easy to determine whether overpruning is the source of an error. If the buggy feature is not in the feature set at all, x is too high. Underpruning is harder to detect. It is a potential problem if the buggy feature appears in the feature set but is not highly placed in the ranked list of problematic features. However, the same outcome occurs if the bug cascades to a number of other features, all of which are perturbed significantly as a result, and hence appear high in the list. Due to this error propagation, it is non-trivial to decide whether more aggressive pruning would have improved localization accuracy.

4.5 Data Collection

This section presents the data collection approach used by WuKong to capture program behaviors at different scales. Recall that the goal of WuKong is to diagnose bugs in program runs at large scales, even if it has never observed correct behavior at that large scale. Therefore, WuKong needs to observe program behaviors at a series of training scales to derive the scaling trend.

The fundamental approach of WuKong is to build a statistical model of program behavior that incorporates scale. Essentially, we would like a model that infers the relationship between scale attributes (*e.g.,* number of processes, or input size) and behavior attributes (*e.g.,* trip count of loops, value distribution of variables). We will discuss what information is collected, how WuKong does the data collection, and a few optimizations to reduce the run-time overhead.

4.5.1 Control and Observational Features

WuKong 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.

WuKong collects values of two types of features: *control* features and *observational* features. Control features generalize scale: they include all input properties and configuration 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 syscall is made, or the number of times a libc function is called.

WuKong uses context-sensitive branch profiles as its observational features. Every time a branch instruction is executed, WuKong's instrumentation computes the current calling context, *i.e.,* the call stack, plus the address of the branch instruc-

tion, and uses the result as an index to access and update the corresponding tuple of two counters: one recording the number of times this branch is *taken*, and the other recording the number of times this branch is *not taken*. The benefits of choosing such observational features are twofold: (1) by choosing observational features that can be associated with unambiguous program points, WuKong can provide a roadmap to the developer to hone in on the source of the bug; (2) with this selection of observational features, WuKong is geared to observe perturbations in both the *taken* → *not taken* and *not taken* → *taken* directions thereby, in principle, detecting and locating all bugs that perturb control-flow behavior.

Observational and control features are collected separately for each unit of execution we wish to model. For example, when analyzing MPI applications, WuKong collects data and builds a model for each process separately. Currently, the execution unit granularity must be specified by the programmer; automatically selecting the granularity is beyond the scope of this work.

4.5.2 Optimizing Call Stack Recording

WuKong's run-time overhead comes solely from collecting the observational features, since the control features can be extracted before running the program. This section presents performance optimizations we employ to reduce the run-time overhead for a given set of observational features. Section 4.3 will describe our approach to pruning the observational feature set, whose main goal is to increase the accuracy of detection and diagnosis, but which has the additional benefit of reducing the overhead of data collection.

WuKong's instrumentation operates at the binary code level, where determining the boundary of a function can be difficult, as compilers may apply complex optimizations, *e.g.,* using "jmp" to call a function or return from one, popping out multiple stack frames with a single instruction, issuing "call" to get the current PC, *etc.*. As a result, simply shadowing the "call" and "ret" instructions cannot capture the call

stack reliably. Instead, WuKong walks down the call stack from the saved frame pointer in the top stack frame, chasing the chain of frame pointers, and recording the return address of each frame until it reaches the bottom of the call stack. This makes sure that WuKong records an accurate copy of the current call stack irrespective of compiler optimizations.

Based on the principle of locality, we design a caching mechanism to reduce the overhead incurred by stack walking in WuKong. First, whenever WuKong finishes a stack walk, it caches the recorded call stack. Before starting the next stack walk, it compares the value of the frame pointer on top of the cached call stack and the current frame pointer register and uses the cached call stack if there is a match. This optimization takes advantage of the temporal locality that consecutive branches are likely to be a part of the same function and therefore share the same call stack. Note that it is possible in theory to have inaccurate cache hit where consecutive branch instructions with the same frame pointer come from different calling contexts. We expect such a case to be rare in practice, and it did not arise in any of our empirical studies.

4.6 Evaluation

This section describes our evaluation of WuKong. We implemented WuKong using PIN [35] to perform dynamic binary instrumentation. To collect the features as described in Section 4.5, we use PIN to instrument every branch in the program to determine which features should be incremented and update the necessary counters. WuKong's detection and diagnosis analyses are performed offline using the data collected after running a PIN-instrumented program at production scales.

We start by conducting large scale fault injection experiments on AMG2006, a benchmark application from the Sequoia benchmark suite [42]. Through these experiments, we show that (a) our log-transformed linear regression model can accurately predict scale-dependent behavior in the observational features for runs at an unseen

large scale; (b) the automatic feature pruning techniques based on cross validation allow us to diagnose injected faults more effectively; (c) as the scale of the test system increases, the modeling time for WuKong remains fixed without hurting accuracy; and (d) the overhead for instrumentation does not increase with the scales of test systems.

We also present two case studies of real bugs, demonstrating how WuKong can be used to localize scale-dependent bugs in real-world software systems. These bugs can only be triggered when executed at a large scale. Thus, they are unlikely to manifest in testing, and must be detected at deployed scales. One of the case studies is also used in Vrisha [14]. We demonstrate here how WuKong can be used to automatically identify which features are involved in the bug and can help pinpoint the source of the fault. The two applications come from different domains, one from high performance computing in an MPI-C program, and the other from distributed peer-to-peer computing in a C program. Since WuKong works at the binary level for the program features, it is applicable to these starkly different domains.

The fault injection experiments were conducted on a Cray XT5 cluster, as part of the XSEDE computing environment, with 112,896 cores in 9,408 compute nodes. The case studies were conducted on a local cluster with 128 cores in 16 nodes running Linux 2.6.18. The statistical analysis was done on a dual-core computer running Windows 7.

4.6.1 Fault Injection Study with AMG2006

AMG2006 is a parallel algebraic multigrid solver for linear systems, written in 104K lines of C code. The application is configured to solve the default 3D Laplace type problem with the GMRES algorithm and the low-complexity AMG preconditioner in the following experiments. The research questions we were looking to answer with the AMG2006 synthetic fault injection study are:

- Is WuKong's model able to extrapolate the correct program behavior at large scales from training runs at small scales?
- Can WuKong effectively detect and locate bugs by comparing the predicted behavior and the actual behavior at large scales?
- Does feature pruning improve the accuracy and instrumentation overhead of WuKong?

We began by building a model for each observational feature of AMG2006, using as training runs program executions ranging from 8 to 128 nodes. The control features were the X, Y, Z dimension parameters of the 3D process topology, and the observational features were chosen using the approach described in Section 4.3, resulting in 3 control features and 4604 observational features. When we apply feature pruning with a threshold of 90%, we are left with 4036 observational features for which WuKong builds scaling models.

Scalability of Behavior Prediction To answer the first research question, we evaluated WuKong on 31 non-buggy test runs of distinct configurations, *i.e.,* each with a unique control feature vector, using 256, 512 and 1024 nodes to see if WuKong can recognize these normal large-scale runs as non-buggy in the detection phase. Based on the detection threshold $\eta = 1.15$ and a feature pruning threshold of 90%, WuKong correctly identified all of the 31 test runs as normal, thus having *zero false positives.* In contrast, the prior state-of-the-art in detection of scale-dependent bugs, Vrisha [14], flags six of the 31 runs as buggy, for a 19.4% false positive rate. Recall that false positives are highly undesirable in this context because each false positive leads the developer to chase after a non-existent bug.

Table 4.1 gives the mean reconstruction error, the time for analysis, and the runtime overhead, due to collecting the observational feature values, at each scale. We see that the average reconstruction error for the features in the test runs is always less than 10% and does not increase with scale *despite using the same model for all*

Table 4.1
Scalability of WuKong for AMG2006 on test runs with 256, 512 and 1024 nodes.

Scale of Run	Mean Error	Analysis Time (s)	Runtime Overhead
256	6.55%	0.089	5.3%
512	8.33%	0.143	5.4%
1024	7.77%	0.172	3.2%

scales. Hence, WuKong's regression models are effective at predicting the large scale behavior of the benchmark despite having only seen small scale behavior.

Furthermore, WuKong's run-time overhead does not increase with scale. Indeed, because there is a fixed component to the overhead of Pin-based instrumentation and larger-scale runs take longer, the average run-time overhead of feature collection *decreases* a little as scale increases. On the other hand, the analysis overhead (evaluating the detection and reconstruction models for the test runs) is always less than 1/5th of a second. Hence, with diminishing instrumentation costs and negligible analysis costs, WuKong provides clear scalability advantages over approaches that require more complex analyses at large scales.

Effectiveness in Fault Diagnosis To determine the effectiveness of WuKong's bug detection and localization capabilities, we injected faults into 100 instances of the 1024-node run of AMG2006. Each time a random conditional branch instruction is picked to "flip" throughout the entire execution. The faults are designed to emulate what would happen if a bug changed the control flow behavior at the 1024-node scale but not at the smaller training scales, as manifested in common bug types, such as integer overflow errors, buffer overflows, *etc.*. This kind of injection has been a staple of the dependability community due to its ability to map to realistic software bugs (*e.g.,* see the argument in [43]).

Using the same pruning and detection thresholds as in the scalability study, we evaluated WuKong's ability to (a) detect the faults, and (b) precisely localize the faults. Of the 100 injected runs, 57 resulted in non-crashing bugs, and 93.0% of those were detected by WuKong. For the crashing bugs, the detection method is obvious and therefore, we leave these out of our study. We also tested with alternative values for the detection threshold η as shown by Table 4.2. This shows, expectedly, that as η increases, *i.e.,*WuKong is less trigger-happy in declaring a run to be erroneous, the false positive rate decreases, until it quickly reaches the desirable value of zero. Promisingly, the false negative rate stays quite steady and low until a high value of η is reached. Furthermore, when $\eta = 1.15$, the average of maximal reconstruction error among all features for the non-buggy runs at scale 256, 512 and 1024 are 96.2%, 91.6% and 91.3%, respectively. Comparing with the average reconstruction error of these runs as shown in Table 4.1, this discrepancies in reconstruction error emphasize the importance of having a different model for each feature to allow different scaling behaviors and improve the overall prediction accuracy.

We next studied the accuracy of WuKong's localization roadmap. For the runs where WuKong successfully detects a bug, we used the approach of Section 4.4.2 to produce a rank-ordered list of features to inspect. We found that 71.7% of the time the faulty feature was the *very first feature* identified by WuKong. This compares to a null-hypothesis (randomly selected features) outcome of the correct feature being the top feature a mere 0.35% of the time. With the top 10 most suspicious features given by WuKong, we can further increase the localization rate to 92.5%. Thus, we find that WuKong is effective and precise in locating the majority of the randomly injected faults in AMG2006.

Sensitivity to Feature Pruning We examined the sensitivity of WuKong to the feature pruning threshold. With a detection threshold $\eta = 1.15$, we used three different pruning thresholds: 0%, 90%, and 99%. Table 4.3 shows how many features were filtered during pruning, the false positive rate of detection, the false negative rate

Table 4.2
The accuracy of detection at various levels of detection threshold with a 90% pruning threshold.

η	False Positive	False Negative
1.05	9.7%	5.3%
1.10	6.5%	7.0%
1.15	0%	7.0%
1.20	0%	7.0%
1.25	0%	12.3%

Table 4.3
The accuracy and precision of detection and localization at various levels of feature pruning with detection threshold parameter $\eta = 1.15$.

Threshold	Features Pruned	Detection		Localization		
		False Positive	False Negative	Located Top 1	Located Top 5	Located Top 10
0%	0%	6.5%	5.3%	64.8%	68.5%	85.2%
90%	12.3%	0%	7.0%	71.7%	77.4%	92.5%
99%	22.5%	0%	12.3%	56.0%	62.0%	78.0%

of detection, the percentage of detected faulty runs where the faulty feature appears among the top 1, top 5 and top 10 of ranked features. Note that if the buggy feature is pruned for a faulty run, localization will always fail.

We see that performing a small amount of feature pruning can dramatically improve the quality of WuKong's detection and localization accuracy: at a threshold of 90%, false positives are completely eliminated from the detection result, compared with a 6.5% false positive rate when no feature pruning is done; in the meantime, over 92.5% of the faulty features appear in the top 10 features suggested by WuKong, a jump from 85.2% in the case of no pruning. We note that being too aggressive with pruning can harm localization: with a threshold of 99% (where all but the most accurately modeled features are pruned), only 78.0% of the cases are successfully located, as too many features are filtered out, resulting in many situations where a bug arises in a feature that is not modeled by WuKong.

Effect of Fault Propagation Occasionally WuKong may detect an error that it cannot localize because the buggy feature has been pruned from the feature set. Because faults can propagate through the program, affecting many other features, WuKong may still detect the error in one of these dependent features despite not

```
1     if (recvcount*comm_size*type_size < MPIR_ALLGATHER_LONG_MSG &&
2         comm_size_is_pof2 == 1) {
3     /*** BUG IN ABOVE CONDITION CHECK DUE TO OVERFLOW ***/
4     /* ALGORITHM 1 */
5     ...
6     } else if (...) {
7     /* ALGORITHM 2 */
8     ...
9     } else {
10    /* ALGORITHM 3 */
11    ...
12    }
```

Figure 4.2. MPICH2 bug that manifests at large scale as performance degradation.

tracking the buggy feature. In 4% of the buggy runs in our fault injection study, with a 90% pruning threshold, the bug is detected but cannot be localized because the faulty feature is pruned (see Section 4.4.3 for a discussion of this seeming contradiction).

In such scenarios, we further investigate whether WuKong's diagnosis could still help developers zoom in to the root cause of a bug. In our study, there were two faults detected by WuKong with root causes in features that were pruned. The two faults targeted the same branch instruction, though with different contexts. In these cases, the top-most feature located by WuKong resides in the same case-block of a `switch` statement as the fault. Moreover, the closest feature to the fault in the top-10 roadmap is a mere 19 lines from the true fault. Given the sheer amount of code in AMG2006, it is clear that WuKong can still help the developer hone in on the relevant code area for bug hunting, even if the precise feature cannot be identified.

4.6.2 Case Study 1: Performance Degradation in MPICH2

To evaluate the use of WuKong in localizing bugs in real-world scenarios, we consider a case study from Vrisha [14], based on a bug in MPICH2's implementation of 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 (before v1.2) contains an integer overflow bug [29], which is triggered when the total amount of data communicated goes beyond 2GB and 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.

We built a test application to expose the ALLGATHER bug when more than 64 processes are employed. The control features were the number of processes in the execution, and the rank of each process, while the observational features were 4126 unique calling contexts chosen as described in Section 4.3. After feature pruning with our default pruning threshold of 90%, WuKong is left with 3902 features. The model is trained on runs with 4–16 processes (all non-buggy), while we attempted to predict the normal behavior for 64-process runs. When the buggy 64-process version was run, WuKong was able to successfully detect the bug. The next question was whether WuKong could aid in the localization of the bug.

First, we used WuKong to reconstruct the expected behavior of the 64-process run and compared it with the observed buggy run. We find that while most features' observed values closely match the predictions, some features are substantially different from the predicted values. As displayed in Figure 4.3, even though the bug involves a single conditional, numerous features are impacted by the fault. However, when we examined the top features suggested by WuKong, we found that all features shared a common call stack prefix which was located inside the branch that would have been taken had the bug not been triggered. Thus, by following the roadmap laid out by WuKong, we could clearly pinpoint the ill-formed "if" statement, the root cause of the bug as shown in Figure 4.2. Here we indirectly located the bug based on the most suspicious features provided by WuKong because the bug did not happen right on one of the observational features we were tracking. We plan to explore direct methods

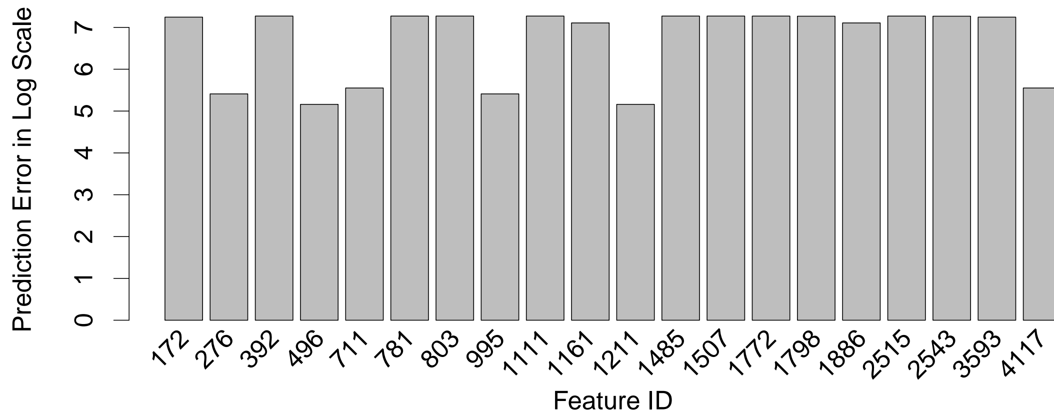

Figure 4.3. The top suspicious features for the buggy run of MPICH2 ALLGATHER given by WuKong.

to locate bugs beyond the set of observational features, such as program slicing, in future work.

Because WuKong's regression models were built using training data collected with a buggy program, an obvious question to ask is whether WuKong is actually predicting what the correct, non-buggy behavior should be, or whether it is merely getting lucky. To test this, we applied a patch fixing the ALLGATHER bug and performed a test run on 64 processes using the now-non-buggy application. We then compared the observed (non-buggy) behavior to the behavior predicted by WuKong's model. We find that the average prediction error is 7.75% across all features. In other words, WuKong is able to predict the *corrected* large-scale behavior; WuKong correctly predicted how the program would behave if the bug were fixed!

4.6.3 Case Study 2: Deadlock in Transmission

Transmission is a popular P2P file sharing application on Linux platforms. As illustrated in Figure 4.4, the bug [36] exists in its implementation of the DHT protocol. Transmission leverages the DHT protocol to find peers sharing a specific file and to form a P2P network with found peers. When Transmission is started, the application

```
1   while (1) {
2     l = strtol((char*)buf + i, &q, 10);
3     if(q && *q == ':' && l > 0) {
4       if(j + l > MAX_VALUES_SIZE)
5         continue;
6       /*** BUG: i INCREMENT IS SKIPPED ***/
7       i = q + 1 + l - (char*)buf;
8       ...
9     } else {
10      break;
11    }
12  }
```

Figure 4.4. The deadlock bug appears in Transmission, and manifests when a large number of peers are contained in a single DHT message.

sends messages to each bootstrapping peer to ask for new peers. Each peer responds to these requests with a list of its known peers. Upon receiving a response, the joining node processes the message to extract the peer list. Due to a bug in the DHT processing code, if the message contains more than 341 peers, longer than the fixed 2048-byte message buffer, it will enter an infinite loop and cause the program to hang. Hence, this bug would more likely manifest when the program is joining a large P2P network where the number of peers contained in a single DHT message can overflow the message buffer.

This bug could be easily detected using full-system profiling tools such as OPROFILE that 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 the function is behaving normally but is just slow. WUKONG is able to definitively indicate that a bug exists in the program.

For this specific bug, given the information provided by OPROFILE, 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 features, we use the same branch profile as in the previous experiments, and the associated contexts, to any shared libraries. This gives 83 features and no feature is pruned with our default 90% pruning threshold.

To train WUKONG, we use 16 normal runs of the message processing function, and apply the trained model to 1 buggy instance. WUKONG correctly determines that the buggy instance is truly abnormal behavior, and not just an especially long-running function. Having established that the long message processing is buggy, WUKONG reconstructs the expected behavior and compares it to the observed behavior to locate the bug, as in Figure 4.5. The rank ordering of deviant features highlights Features 53 and 66, which correspond to the line `if (q && *q == ':' && l > 0)` at

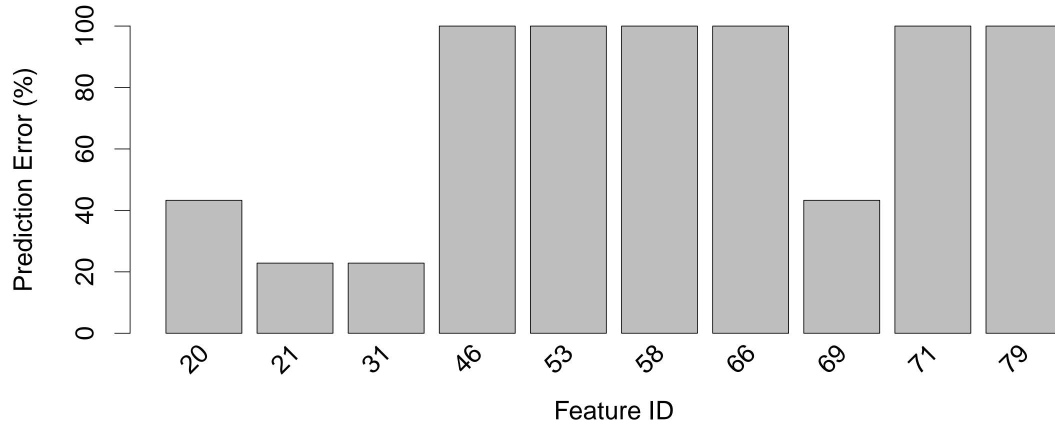

Figure 4.5. The top suspicious features for the buggy run of Transmission given by WuKong.

the beginning of Figure 4.4, exhibiting an excessive number of occurrences as a direct consequence of the bug. This feature is a mere 3 lines above the source of the bug.

4.6.4 Overhead

To further evaluate the overhead of WuKong, we used 5 programs from the NAS Parallel Benchmarks, namely CG, FT, IS, LU, MG and SP[2]. All benchmarks are compiled in a 64-process configuration and each is repeated 10 times to get an average running time. Figure 4.6 shows the average run-time overheads caused by WuKong for each of these benchmarks. The geometric mean of WuKong's overhead is 11.4%. We note that the overhead with the larger application, AMG2006, is smaller (Table 4.1).

It is possible to reduce the cost of call stack walking—the dominant component of our run-time overhead—by using a recently demonstrated technique called Breadcrumbs [44] and its predecessor called Probabilistic Calling Context (PCC) [45], both of which allow for efficient recording of dynamic calling contexts. Breadcrumbs builds

[2]The overhead numbers for our two case studies are not meaningful—for MPICH2, we created a synthetic test harness; and for Transmission, we relied on a prior use of profiling to identify a single function to instrument.

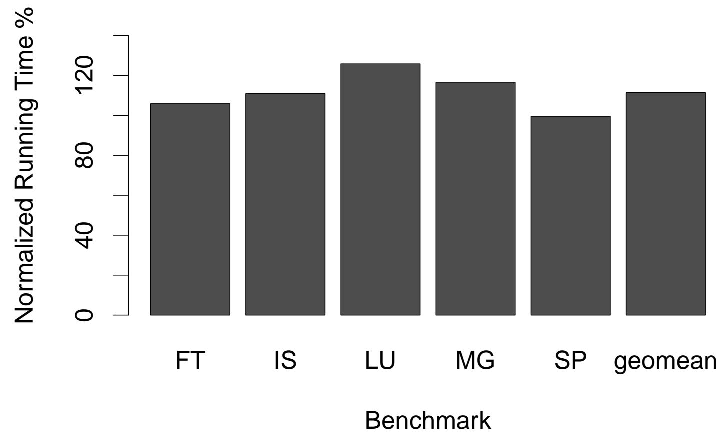

Figure 4.6. Runtime overhead of WuKong on NPB benchmarks.

on PCC, which computes a compact (one word) encoding of each calling context that client analysis can use in place of the exact calling context. Breadcrumbs allows one to reconstruct a calling context from its encoding using only a static call graph and a small amount of dynamic information collected at cold (infrequently executed) call-sites. We defer to future work the implementation of these techniques in WuKong to capture calling contexts at even lower overheads.

4.7 Summary

With the increasing scale at which programs are being deployed, both in terms of input size and system size, techniques to automatically detect and diagnose bugs in large-scale programs are becoming increasingly important. This is especially true for bugs that are scale-dependent, and only manifest at (large) deployment scales, but not at (small) development scales. Traditional statistical techniques cannot tackle these bugs, either because they rely on data collected at the same scale as the buggy process or because they require manual intervention to diagnose bugs.

To address these problems, we developed WuKong, which leverages novel statistical modeling and feature selection techniques to automatically diagnose bugs in large scale systems, even when trained only on data from small-scale runs. This ap-

proach is well-suited to modern development practices, where developers may only have access to small scales, and bugs may manifest only rarely at large scales. With a large-scale fault injection study and two case studies of real scale-dependent bugs, we showed that WuKong is able to automatically, scalably, and effectively diagnose bugs.

5 Lancet: GENERATING TARGETED SCALING TESTS

5.1 Background: Dynamic Symbolic Execution

Many state-of-the-art white-box test-generation tools rely on *dynamic symbolic execution* [46–48]. This section provides a brief overview of how these tools work, and discusses some of the shortcomings of current techniques that Lancet aims to ameliorate.

5.1.1 Dynamic Symbolic Execution Basics

Dynamic symbolic execution couples the traditional strengths of symbolic analysis—tracking constraints on variables to determine the feasibility of various program behaviors—with the efficiency of regular concrete execution. The inputs to a program are treated as symbolic variables. As the program executes, statements that involve symbolic variables are executed symbolically, adding constraints on symbolic variables. When a branch statement is reached (*e.g.* `if (x < y) goto L`), *one* of the paths is taken, and the appropriate constraint is added to the *path condition*, or set of constraints (*e.g.*, if the branch above is taken, the constraint $(x < y)$ would be added to the path condition). As new constraints are added to the path condition an SMT solver (Satisfiability Modulo Theory) is invoked to ensure that the path condition is still satisfiable; unsatisfiable constraints imply that the particular path that execution has taken is infeasible—no program execution could follow that path. Once a path through the program is found, the SMT solver is used to produce a *concrete* set of values for all the symbolic variables. These concrete values constitute an input. Note that unlike traditional symbolic execution, dynamic symbolic analysis can always fall back to concrete values for variables; if execution encounters a state-

ment that cannot be analyzed using the underlying SMT solver, calling an external function for example, the variables involved can be concretized and the statement can be executed normally. What it gives up is completeness of the code coverage as a result of the concretization.

5.1.2 Path Exploration Heuristics

One of the key decisions in dynamic symbolic execution tool is how to choose the path through a program an execution should explore. In other words, how to choose directions for the branches encountered in execution. For example, a tool such as KLEE [46] executes multiple paths concurrently, in an attempt to generate a series of inputs to exercise different paths of the program. When encountering the branch statement described above, KLEE can fork the execution to two clones that follow the two branches respectively, choose one execution clone to explore first and save the other for later. The resultant paths would include the constraint $(x < y)$ in the clone that follows the true branch, and $\neg(x < y)$ in the one that follows the false branch.

There are numerous heuristics that can be used to choose which path to explore first at each branch in dynamic symbolic execution; the choice of the correct heuristic depends on the goals of the particular tool being developed. We abstract away the goal of a dynamic symbolic execution tool by describing it in terms of a *meta-constraint*. A meta-constraint is a higher level constraint that the path condition describing a particular execution attempts to satisfy. For example, in the case of generating high-coverage tests, the meta-constraint for a tool may be to produce a path that exercises program statements not seen by previously-generated inputs, in which case a path-exploration heuristic might prioritize flipping branches to generate a never-before-seen set of constraints. In the case of generating stress tests, as in LANCET, the meta-constraint would be to generate a path condition that exercises a particular loop body a certain, user-defined number of times, in which case the path-exploration heuristic might prioritize taking branches that cause the loop to execute again, till the user-

defined limit is reached (Section 5.2.2). Note that just as a path condition may not be tight—there can be many possible concrete inputs that follow a particular path through the execution—a meta-constraint need not be tight: multiple path conditions may all satisfy a given meta-constraint.

5.1.3 Dynamic Symbolic Execution Overhead

The primary drawback of dynamic symbolic execution is its dependence on an underlying SMT solver to manage the path condition that an execution generates. At every branch, the SMT solver must be invoked to determine whether a particular choice for a branch is feasible or infeasible. Though the constraint solver is also invoked to concretize input variables when necessary, or to generate the final concrete input for a particular run, the majority of queries to the solver arise from these feasibility checks [48]. Though there are many techniques to reduce the expense of queries to the constraint solver, such as reducing the query size by removing irrelevant constraints and reusing the results of prior queries whenever possible [46], the fundamental expense of invoking the constraint solver at each branch in an execution remains. This overhead can lead to an order-of-magnitude slowdown in execution [46].

A second overhead of dynamic symbolic execution is the path-explosion problem. The path exploration heuristics of a particular tool may prioritize certain choices for branches in an attempt to satisfy the tool's meta-constraint. However, the exploration heuristics may be wrong: whenever a branch is encountered, if the heuristic makes the wrong choice, the meta-constraint may not be satisfiable, and the tool must return to the branch to make a different choice.

These two overheads make generating large-scale, long-running inputs using dynamic symbolic execution difficult. First, as the path being explored gets longer, more branches are encountered, resulting in more invocations to the SMT solver. Second, long-running inputs execute more branches, creating more opportunities for the path-

exploration heuristic to "guess wrong," leading to unsatisfiable meta-constraints and ultimately leading to path explosion.

5.1.4 WISE

WISE is perhaps the most closely-related dynamic symbolic execution technique to LANCET. WISE attempts to generate worst-case inputs for programs (*e.g.*, a worst-case input for a binary-search-tree construction program would be a sorted list of integers, generating an unbalanced tree) [49]. In meta-constraint terms, WISE attempts to generate a path condition that produces worst-case inputs of a reasonably large size.

While it is possible to exhaustively search through all possible path conditions for a certain input size to find the worst-case inputs, it is clear that this approach will lead to path explosion when applied to larger inputs. To avoid the path explosion problem, WISE uses the following strategy. It exhaustively searches the input space for small input sizes to find worst-case behaviors. Then, by performing pattern matching on the worst-case path conditions generated for different input sizes, WISE learns what choices worst-case path conditions typically make for branches in the program (*e.g.*, always following the left child to add a new element in a BST). This pattern is then integrated into a path-exploration heuristic.

At large scales, when WISE encounters a branch, it looks through the patterns that it identified at small scales to choose the direction for the branch. In essence, WISE has a notion of what a worst-case path condition "looks like," and chooses directions for branches to make the path condition for a larger input match that worst-case pattern. By using this new path-exploration heuristic, WISE is able to find worst-case inputs for larger scales without exploring every possible path. In many cases, WISE need only explore a single path through the program to find a worst-case input!

WISE addresses the overheads of dynamic symbolic execution by tackling the second drawback described above: it controls path explosion by learning the pattern of path constraints from smaller runs where path explosion is less of an issue. Nevertheless, WISE still sits on top of dynamic symbolic execution, and must query the SMT solver at every branch; it does not tackle the first source of overhead. As a result, even though WISE has much better scalability than naïve symbolic execution, it still cannot generate particularly large inputs. Generating an input that runs a loop one million times still requires performing symbolic execution on a run that visits the loop test condition one million times.

Section 5.2.3 explains how Lancet tackles this problem. In particular, Lancet can generate large-scale inputs for a program *without ever performing symbolic execution at large scales.*

5.2 Design

Lancet is a dynamic symbolic execution tool that aims to generate *scaling inputs* for programs: inputs that cause programs to run for a long time. In contrast to black-box stress-generation tests, Lancet targets particular loops for scaling. It attempts to generate inputs that will cause a chosen loop to execute a specified (large) number of iterations. In this way, particular regions of code can be targeted to see how they behave under heavy load. This section first describes Lancet's behavior at a high level, and then explains Lancet's various components in more detail. For ease of exposition, this section assumes that the program under test takes a single symbolic string as input.

5.2.1 Overview of Lancet

Lancet has two modes of operation: *explicit* mode, which uses traditional dynamic symbolic execution techniques to generate inputs that run a target loop for a specified, small number of iterations, and *inference* mode, which uses statistical

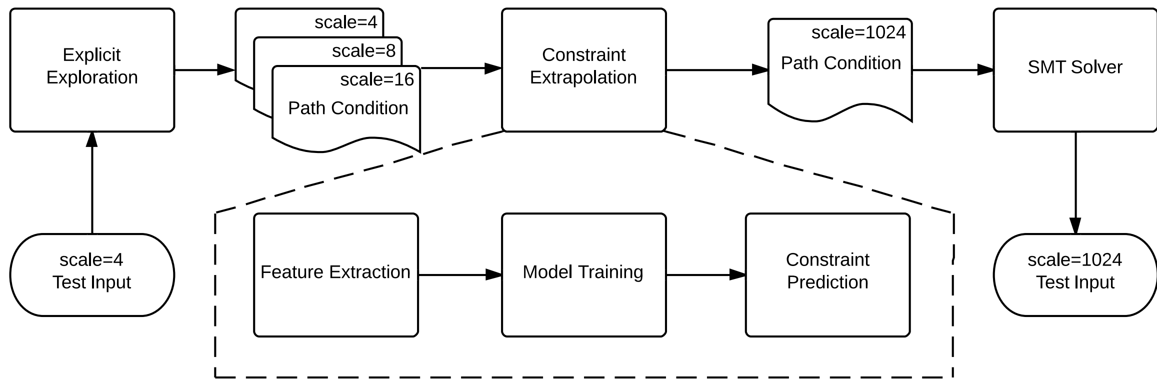

Figure 5.1. High level flow of LANCET's inference-mode approach for generating inputs for a given loop.

techniques to generate inputs that run a target loop for a large number of iterations. These modes are described in more detail in Sections 5.2.2 and 5.2.3, respectively. At a high level, they behave as described next.

LANCET's explicit mode begins by using programmer annotations to identify the target loops (Section 5.2.2). For each target loop, LANCET then generates a path condition that satisfy a *loop-iteration meta-constraint* that the target loop executes exactly N times. For small N, this is done by a custom *loop-centric* path-exploration heuristic (Section 5.2.2).

While the explicit mode suffice to generate path constraints that run a loop a small number of times, it is impractical for large N: *e.g.,* running a loop a thousand times will require two orders of magnitude more invocations to LANCET's constraint solver than running a loop ten times, and will be unacceptably slow. LANCET's novel approach to this problem is to determine the path constraints that satisfy the loop-iteration meta-constraint for a large N *without performing symbolic execution.*

LANCET's inference mode operates as shown in Figure 5.1. First, LANCET uses its explicit mode to generate *multiple* path conditions, for a pair of consecutive numbers of iterations M and $M+1$, using a symbolic input string of L bytes.[1] It then

[1]Multiple path conditions may result in the same number of iterations.

looks into pairs of path conditions for different numbers of iterations and identifies the *incremental set*, the set of path constraints that *only* exist in the $M+1$-iteration path conditions. LANCET then extrapolates the path condition of N iterations by appending $N-M$ copies of the incremental set to the M-iteration path condition, each projected to appropriate offsets beyond the initial L bytes of the input string using a linear regression model (Section 5.2.3). Finally, the predicted N-iteration path condition is solved with a constraint solver to generate a large-scale input for the program (Section 5.2.3), which is then verified in real execution of the program. Depending on the result of verification, LANCET may restart the explicit mode to generate more training data and refine the large-scale input based on new path conditions discovered in the explicit mode.

Crucially, once the initial training phase of LANCET is complete, inputs that target *any scale* can be generated at the same overhead (though potentially different levels of accuracy), making LANCET a truly scalable approach to generating large-scale, stress-test inputs.

Running example To aid in the discussion of the components and operation of LANCET, we will use a running example of request parsing from Memcached [50]. A few salient points: (i) the code parses the request string that contains a *get* command followed by a list of keys separated by one or more spaces; (ii) the first function `tokenize_command` splits the command string (a symbolic string of configurable size received from the Socket layer of LANCET, see Section 5.3.2) into a list of tokens, stores them as a list of tokens terminated by a length 0 token, then returns the number of tokens retrieved; (iii) the second function `parse_get_command` parses the list of tokens in the target loop (line 29) and executes the *get* command for each key contained in `tokens`. (iv) the number of iterations executed by the target loop is determined by the loop in function `tokenize_command`. (v) in the following references to this example, we suppose the length of the `command` string is 8 bytes for the ease of exposition.

```
1  size_t tokenize_command(char *command, token_t *tokens, size_t max_tokens) {
2      char *s, *e;
3      size_t ntokens = 0;
4      size_t len = strlen(command);
5      unsigned int i = 0;
6      s = e = command;
7      for (i = 0; i < len; i++) {
8          if (*e == ' ') {
9              if (s != e) {
10                 /* add a new token into tokens */
11                 ntokens++;
12                 if (ntokens == max_tokens - 1) { e++; s = e; break; }
13             }
14             s = e + 1;
15         }
16         e++;
17     }
18     if (s != e) {
19         /* add the last token into tokens */
20         ntokens++;
21     }
22     /* add a terminal token of length 0 into tokens */
23     ntokens++;
24     return ntokens;
25 }
26
27 void process_get_command(token_t *tokens, size_t ntokens) {
28     token_t *key_token = &tokens[KEY_TOKEN]; /* KEY_TOKEN is the offset to the first key */
29     [[loop_target]] while(key_token->length != 0) {
30         /* retrieve the key from cache */
31         key_token++;
32     }
33 }
```

Figure 5.2. Running example: request parsing in Memcached.

5.2.2 Explicit Mode

The goal of LANCET is to find performance tests that impose a certain load level precisely on a certain part of code in the given program. Specifically, LANCET's test generation is designed for loops and therefore the load level is determined by the trip count of a loop.

For a given trip count N and a target loop l, LANCET's explicit mode uses symbolic execution to generate an execution path that satisfies the meta-constraint that the path executes loop l exactly N times. The symbolic execution engine treats the input as a bitvector of symbolic variables, computes symbolic expressions for input-dependent variables, and accumulates the constraints at every branch to form the set of constraints, *i.e.,* the path condition, that must hold when the path is followed in an execution. LANCET obtains a test input for the program that will run l for N iterations by calling an external SMT solver to find concrete values that satisfy the path condition.

Targeting a loop

LANCET provides a simple yet powerful interface for user to specify which loops she wants to target using source code annotation. To mark a loop as a target for test generation, the attribute `[[loop_target]]` needs to be inserted right before the loop statement. In the running example, the loop at line 29 is targeted for test generation.

Loop-centric search heuristic

The powerful multi-path analysis enabled by symbolic execution comes with a price: the path explosion problem. In order to get meaningful results within a reasonable time frame, any symbolic execution tool must steer through the exponentially growing number of paths and prioritize the exploration of the more interesting ones.

For example, as demonstrated by KLEE [46], path searching heuristics like random path selection and coverage-optimized search are effective for generating high-coverage tests for complex programs (like Gnu Coreutils). However, these heuristics, though good for discovering unexplored code, are ill-suited for the purpose of generating performance tests, because rather than exercising every line of code once, as a functional test suite might, a performance test should instead repeatedly execute critical pieces of code to simulate high loads.

Lancet employs a loop-centric heuristic to guide the search for paths that extend the target loop for a large number of iterations. Following many existing symbolic execution tools, Lancet encapsulates runtime execution information such as program counter, path condition, memory content in a symbolic process. The loop-centric search operates in two modes, the *explorer* mode and the *roller* mode.

In explorer mode, Lancet starts the execution with a single symbolic process from program entry, forking a new process at each branch that has a satisfiable path condition (this is the default execution mode for KLEE). If the loop header of the target loop, l is hit by any of these symbolic processes, that process enters roller mode and the other explorer processes are paused. Roller mode prioritizes symbolic processes that stay inside the target loop (*e.g.*, taking loop back edges to avoid exiting the loop) so that it can reach a high number of iterations more quickly.

Roller mode maintains a FIFO queue for all symbolic processes whose current program counters are inside the target loop and schedules the next process from the head of the queue whenever the queue is not empty. Each symbolic process tracks how many times it has executed the target loop. Lancet counts the number of times the loop has run *in the current calling context* (*i.e.*, the loop trip count is reset if the function is exited). This policy means that in nested loops, inner loops cumulatively count iterations across all iterations of any outer loop. If a symbolic process has executed exactly N iterations, roller mode attempts to exit the loop, yielding a path constraint for an input that will run the loop exactly N times. The explorer mode

is agnostic to the search strategy and any effective code discovery strategy could be leveraged by LANCET for identifying a path from the input to the target loop.

Example: In explorer mode, LANCET will spawn symbolic processes that try every possible path through the program in Figure 5.2. Because `process_get_command` is called only for *get* requests where the `command` string starts with 'get␣', every process that reaches the target loop at line 29 would include the following constraints:

$$command[0] = 'g' \wedge command[1] = 'e' \wedge command[2] = 't' \wedge command[3] = '␣'$$

A process that executes the target loop for one iteration will end up with the following additional constraints:

$$command[4] \neq '␣' \wedge command[5] \neq '␣' \wedge command[6] \neq '␣' \wedge command[7] \neq '␣'$$

Another process that executes the target loop for two iterations will accumulate constraints as follows:

$$command[4] \neq '␣' \wedge command[5] = '␣' \wedge command[6] \neq '␣' \wedge command[7] \neq '␣'$$

A direct comparison between the constraints of 1-iteration and 2-iteration processes would reveal that, omitting the case of consecutive spaces, the number of times the condition at line 8 is true is determined by the number of tokens the string contains, thereby the number of iterations the target loop executes. This observation will lead to our key insight for the inference mode.

5.2.3 Inference Mode

A strawman approach to performance test generation would use LANCET's explicit mode exclusively to generate large-scale inputs, targeting loop l to run N times for some large N. This approach could generate tests that accurately trigger the target loop for N times if given indefinite amount of time. However, nontrivial loops that contain complex control flow structure may cause the path explosion problem

after a large number of iterations even if LANCET only considers the code enclosed by these loops. Secondly, the symbolic execution engine needs to consult with the constraint solver at every branch instruction to determine if the current path condition is satisfiable. In a state-of-the-art symbolic execution tool, more than half of the time is spent by the constraint solver [46]. It is simply impractical to run a symbolic execution engine for more than a handful of iterations of the target loop.

Since our goal is not to verify every possible execution path, but merely to generate a large-scale input, it is unnecessary and wasteful to execute every iteration of the target loop through the symbolic execution engine. LANCET's inference mode takes a more efficient approach that skips symbolic execution of these intermediate iterations and simply generates the path condition for the Nth iteration. In further detail, the training of LANCET's interference is done for various small scale inputs that execute the target loop up to M times, $M << N$, and then skips executing the loop between M and N times.

Recall the running example where the number of iterations of the target loop is determined by the number of times the true branch is taken at line 8. This observation leads to our key insight that for many loops, there is a statistical correlation between the desired trip count for a loop and the number of constraints generated by a set of critical branches, and this correlation can be used for inference of the path condition for the Nth iteration. In its essence, a path condition is just a document that contains a set of constraints represented by strings. However, it is difficult and inaccurate to generate them directly from the inference model using general text mining techniques if we treat a set of constraints as an unstructured document. LANCET first extracts features from the path conditions based on the structural properties of path condition, and trains a regression model to capture the correlation between the trip count of the target loop and each feature of the path conditions using the data from small-scale training runs. The structural features of the N-iteration path condition are then predicted using the regression models and the N-iteration path condition is generated based on the predicted features. Finally, LANCET solves the N-iteration

path condition to obtain a concrete input using a SMT solver, and verifies the input in real execution. In case the input verification fails, LANCET switches back to the explicit mode to generate more training data before running the inference mode again. We will present each of these steps of the inference mode in the following sections.

Extracting features from path conditions

LANCET transforms path conditions into constraint templates and numerical vectors, which are then used to train the statistical models LANCET builds to capture the relationship between the trip count of a loop and the resultant path condition. As a preprocessing step, LANCET first puts the constraints of each path condition into groups introduced by the same branch instruction, then sorts each group by the lowest offset of symbolic byte each constraint accesses. Each ordered group of constraints constitute a *feature* in LANCET's inference mode. For a series of path conditions, $\{P_i \mid i \leq M+1\}$, where P_i represents the path condition ensued by i iterations, and each path condition is processed into a set of features $\{P_i^j\}$, where P_i^j represents the jth feature of P_i, LANCET finds the *incremental set* D_{i+1}^j, the residual part of P_{i+1}^j after removing the longest common prefix between P_{i+1}^j and P_i^j. In the running example, the incremental set between the 1-iteration and the 2-iteration path conditions contains the following constraints:

$$command[5] =' \textvisiblespace ' \wedge command[6] \neq' \textvisiblespace ' \wedge command[7] \neq' \textvisiblespace '$$

Intuitively, the incremental set starts at the first byte where two path features differ and continues till the end in the feature of the more number of iterations.

LANCET extracts from a incremental set the following information: (a) the set of constraint templates; (b) the offsets of symbolic bytes referenced by each constraint; (c) the values of the concrete numbers in each constraint. The constraint templates can be obtained from a incremental set by replacing offsets of symbolic bytes and concrete numbers in each constraint with abstract terms numbered by their appearances. The sequence of offsets of symbolic bytes and concrete numbers are also recorded in

the meantime. For example, the above incremental set from the running example can be abstracted into constraint templates:

$$command[x_1] = x_2 \wedge command[x_3] \neq x_4 \wedge command[x_5] \neq x_6$$

The corresponding sequence of symbolic variable offsets is 5, 6, 7, and the sequence of concrete numbers 32, 32, 32 (32 is the ASCII code for space).

It is possible to reach the same number of iterations with different path conditions. For example, another path that finishes the target loop for one iteration in the running example may require this condition:

$$command[4] =' \text{␣}' \wedge command[5] \neq' \text{␣}' \wedge command[6] \neq' \text{␣}' \wedge command[7] \neq' \text{␣}'$$

Furthermore, the incremental set between this path condition and the previous 2-iteration one is:

$$command[4] \neq' \text{␣}' \wedge command[5] =' \text{␣}' \wedge command[6] \neq' \text{␣}' \wedge command[7] \neq' \text{␣}'$$

which is longer than the aforementioned incremental set. In light of this case where multiple paths are possible for the same trip count of a loop, LANCET uses the minimal incremental set, which contains the shortest list of constraints.

Inference over path conditions

LANCET infers the next $N-M$ incremental sets based on the constraint templates and the sequences of terms, i.e. symbolic variable offsets or concrete numbers, extracted from the current incremental set for M iterations. It first extrapolates these sequences into the next $N-M$ iterations and then fill $N-M$ copies of the templates with predicted values. Since both kinds of sequences contain numbers, LANCET uses the same algorithm to predict them. For the common case when there are multiple constraints in the current incremental set, LANCET employs a regression model to predict the sequence. For the case when there is a single constraint in the current

incremental set, LANCET applies two extrapolation heuristics: (i) repeating the single number in the sequence; (ii) extending the sequence with a series of consecutive numbers.

In our running example, there are 3 constraints in the current incremental set, therefore LANCET will use a regression model to capture the relationship between the trip count and the sequences of terms, and predicts the following incremental set for the 3rd iteration of the target loop:

$$command[8] =' \text{␣}' \wedge command[9] \neq' \text{␣}' \wedge command[10] \neq' \text{␣}'$$

Note although the size of the symbolic input string is constant due to the lack of support for string length operation in the underlying SMT solver [51], LANCET is able to infer constraints beyond the fixed range of input in its inference mode and generate path condition that references input of arbitrary length.

Generating a large-scale input

Once LANCET predicts a path condition to execute the loop N times, it calls the SMT solver [51] to solve the predicted path condition to generate an appropriate large-scale input. For the generated test inputs, it also verifies the actual number of iterations achieved for the target loop in the runtime. LANCET compiles the program under test into a native x86-64 executable with lightweight instrumentation inserted around the target loop in compile time, to record the number of times the loop header is observed during an execution. If the actual trip count does not reach the set goal, LANCET returns to its explicit mode to explore the loop for more iterations and covers more code paths within the loop body, then feeds the new data into the inference mode to get an improved prediction of the path condition for N iterations. This process repeats until the actual trip count are within a small range of N, which is a configurable option of LANCET.

5.3 Implementation

This section presents the implementation of LANCET. The symbolic execution engine used in the explicit mode of LANCET is built on top of KLEE, with added support for pthread-based multithreaded programming, libevent-based asynchronous event processing, socket-based network communication and various performance enhancements for the symbolic execution engine. LANCET uses the build system of Cloud9 [52], based on Clang [53], allowing it to compile makefile-based C programs into LLVM bitcode [54] required by the symbolic execution engine. We now discuss the changes made to baseline KLEE.

5.3.1 POSIX Thread

LANCET supports most of PThread API for thread management, synchronization and thread-specific store (Table 5.1). A thread is treated the same as a process except that it shares address space and path condition with the parent process that has created it. And since threads are treated as processes, they are scheduled sequentially to execute unless a thread is blocked in a synchronization calls, such as calling `pthread_mutex_lock` on a lock that has been acquired by another thread. When a thread encounters a branch with a symbolic condition, it will forked all threads belonging to the same process, essentially making a copy for the entire process. Both mutex and condition variable are implemented as a wait queue. When the current owner releases the synchronization resource, LANCET will pop a thread from the wait queue to take control of the resource and mark the thread runnable. LANCET applies the *wait morphing* technique to reduce wasted wake-up calls in condition variables. When the condition variable has an associated mutex, `pthread_cond_signal/broadcast` does not wake up the threads, but rather move them from waiting on the condition variable, to waiting on the mutex.

Table 5.1
LANCET supports most of PThread API for thread management, synchronization and thread-specific store.

Category	**API**
Thread Management	pthread_create
	pthread_join
	pthread_exit
	pthread_self
Synchronization	pthread_mutex_init
	pthread_mutex_lock
	pthread_mutex_trylock
	pthread_mutex_unlock
	pthread_cond_init
	pthread_cond_signal
	pthread_cond_broadcast
	pthread_cond_wait
Thread Specific Store	pthread_key_create
	pthread_key_delete
	pthread_setspecific
	pthread_getspecific

5.3.2 Socket

Socket API is a network programming interface, provided by Linux and other operating systems, that allows applications to control and use network communication. Server applications, like Apache, MySQL and Memcached, consume data received from the network via sockets, therefore cannot be tested in solitude. Previous work [52] that generates tests for Memcached using symbolic execution combines a client program into the server code to address this problem. However, this approach requires deep knowledge of the server program under test to write the client code and cannot be easily generalized. Inspired by Unix's design that treats sockets the same as regular files, LANCET takes a similar approach that implements symbolic sockets as regular symbolic files of a fixed size configurable in the command line. Essentially, instead of simulating sporadic network traffic in real world, LANCET sends a single symbolic packet to the server program under test. Leveraging the existing symbolic file system of KLEE, our implementation creates a fixed number of symbolic files during system initialization, and allocates a free symbolic file to associate with a socket when the socket is created.

5.3.3 Libevent

Libevent is a software library that provides asynchronous event notification and provides a mechanism to execute a callback function when a specific event occurs on a file descriptor. LANCET implements Libevent as part of its runtime, centering around a core structure, *event base*, the hub for event registration and dispatch. It simulates event base as a concurrent queue that supports multiple producers and consumers. The queue implementation is based on PThread condition variable and mutex for thread synchronization. To register a event, LANCET inserts a new item into the corresponding event queue.

A Libevent-based application usually contains a thread that calls the function `event_base_loop` to execute callback functions for events in a loop. In LANCET's

implementation of Libevent, `event_base_base` runs a loop forever to go through an event queue and call registered callback functions for activated events until all events are deleted from the queue. `event_base_loop` also employs PThread condition variable to synchronize with event registration and dispatch.

5.3.4 Various Optimizations

Simplified libraries Certain functions in the C standard library consume a significant amount of time in symbolic execution. One case is `atoi()`, which is frequently used in C programs to transform a NULL-terminated string into an integer. The standard implementation for `atoi()` supports different bases for the integer and tolerates illegal characters in the input string, all of which add complexity to the code and slow down symbolic execution with a huge number of execution paths. Since LANCET is not concerned with looking for corner cases that trigger bugs—unlike KLEE—we opt to simplify some of these common functions in the runtime. For example, in a simplified `atoi()`, only characters between '0' and '9' is allowed except for the trailing NULL in the string. Note that this simplification applies only to KLEE's handling of `atoi()`; the program under test need not be changed. This simplification also helps keep constraints from different runs in a consistent form and eases the identification of constraint templates.

Scheduling changes The explorer mode of LANCET leverages a code discovery strategy that biases for new code coverage. We found this searching strategy often results in breadth-first traversal of execution paths, causing excessive memory utilization during the search. This phenomenon is because programs usually allocate memory at the beginning of execution and release memory towards the end. Breadth-first search strategies thus perform allocations for every symbolic process, consuming significant amounts of memory. KLEE has a copy-on-write memory sharing mechanism in place to mitigate this problem. However, it cannot skip the duplication of memory allocation when memory initialization is used after allocation. For example,

in libquantum, calloc() is used to acquire memory and initialize the content by zeroing the allocated memory immediately.

To address this problem, LANCET employs an auxiliary search strategy in explorer mode to delay the execution of symbolic processes that are about to make an external function call (*e.g.,*malloc() or calloc()) This strategy keeps all processes with imminent external calls in a separate queue and prioritizes the execution of processes that stay inside the application's own code. When the only processes left are those in the queue, LANCET dequeues a waiting process and begins execution. Thus, LANCET makes sure that earlier symbolic processes have a chance to release resources before another process allocates additional resources.

5.4 Evaluation

We have used LANCET to generate inputs for several applications, both in explicit mode and inference mode. The benchmarks we use to evaluate LANCET's explicit mode are mvm, a simple program for matrix-vector multiplication, lq (libquantum), a simulation of quantum factoring, lbm, a Lattice-Boltzmann computational simulation, and wc, the Unix word-count utility. We take Memcached, a distributed in-memory object caching system as a case study for the inference mode. The first three benchmarks are *numeric*: the inputs to the benchmarks are simple numerical values, which LANCET naturally handles. The fourth benchmark, wc, is *structured*: it takes a list of words separated by spaces and line breaks as the input. Memcached, on the other hand, serves a variety of different commands defined by its client-server communication protocol. Both lq and lbm are drawn from the SPEC CPU2006 benchmark suite. In all benchmarks, we targeted the main (outer) loop of the program for scaling, except with mvm, where we targeted both the inner (row) loop and outer (column) loop (labeled as mvm(i) and mvm(o) in the following discussion). For the case study with Memcached, we targeted the loop that processes a "get"

command, which is used to retrieve data objects associated with the list of keys given in the command.

5.4.1 General Observations with Benchmarks

Lancet's Explicit Mode versus KLEE

In our first experiment, we compared the effectiveness of Lancet's explicit mode for generating scaling inputs to baseline KLEE. This comparison primarily highlights the modifications made to the symbolic execution engine in Lancet: the loop-centric search heuristic and the scheduling changes.

For each program, we ran Lancet in explicit mode for 1 hour (with the exception of `lbm`, which we ran for 24 hours, as the target loop is deep in the code), and determined (i) how many tests Lancet was able to generate (only counting tests that execute the loop at least once), and (ii) the largest-scale input Lancet was able to generate (*i.e.,* the largest number of iterations the target loop executed in any synthesized input). We then performed the same experiment using baseline, unmodified KLEE. The results are given in Table 5.2.

We see that for the more complex programs, `lq` and `lbm`, Lancet is able to generate more test inputs that stress the loop than KLEE, and that those inputs run the loop for more iterations. In `lbm`, KLEE cannot even generate test inputs, as it runs out of memory. Lancet's optimized process scheduling (Section 5.3) avoids this pitfall. In `lq`, although KLEE successfully generates 4 inputs in one hour, none of them reach the target loop, while Lancet generates 168 different inputs. This advantage arises because of Lancet's "roller" path exploration heuristic. While searching for the loop itself, Lancet behaves similarly to KLEE. However, once a path reaching the loop is found, Lancet builds upon that path as much as possible to run the loop for additional iterations.

Note that the `mvm(i)` numbers are misleading. Lancet was configured to stop generating inputs when it reached 100 iterations. Although KLEE may appear to

Table 5.2
Effectiveness of LANCET and KLEE at generating scaling inputs for target programs. KLEE is unable to generate any inputs for `lbm`, as it runs out of memory.

Bench	LANCET		KLEE	
	# of tests	Max scale	# of tests	Max scale
`mvm(i)`	50	100	285	307
`mvm(o)`	81	81	355	4
`lq`	168	27	4	N/A
`lbm`	14	5	0	N/A
`wc`	67	89	502	375

have generated more, and larger, inputs, LANCET generated the `mvm(i)` inputs in ∼1 second, and clearly could have outpaced KLEE. We will perform a more equitable comparison in the final version.

Note that because KLEE does not perform symbolic allocation (allocations are concretized), its normal execution is faster than LANCET; LANCET's advantage arises purely from its optimized path exploration heuristics and process scheduling. In programs where path explosion is attenuated, KLEE can be faster. We see this effect in `wc`, where KLEE generates larger inputs than LANCET. Note, however, that LANCET's inference mode will still allow it to generate large inputs faster, as it need only generate a handful of small inputs to start predicting path conditions for larger inputs.

Inferring Large-scale Inputs

Using the path conditions generated in the explicit mode for each benchmark, we used LANCET's inference mode to predict input values that would run the loop for larger scales. In particular, we attempt to use LANCET to generate an input that will run a loop exactly N times, where N is larger than the scales seen during training.

In all cases, LANCET *successfully generates the large-scale input.* This is despite the very different scaling trends that different programs have. For example, `lbm` scales linearly with the input, while `lq` scales *logarithmically* with its input (the loop runs for $\log_2(N)$ iterations), and LANCET is able to correctly capture both trends.

Interestingly, for all benchmarks, LANCET predictions are perfect: *the generated input runs the program for exactly the specified number of iterations.* This is because the behaviors of these applications are deterministic, hence the *constraints* that govern scaling tend to be highly predictable, even if any individual set of inputs may not be.

5.4.2 Case Study with Memcached

`Memcached` runs as a caching server that communicates with clients through TCP connections. Clients send commands to the server to store and fetch data objects. The server reads commands and sends back responses through a socket file descriptor for each client. In LANCET's symbolic socket layer, a socket is implemented as a symbolic file so that a server application can be tested by it self and all its interaction with clients through the socket can be recorded in the content of the symbolic file. For the case study of `Memcached`, we added code to establish a connection with a single client which sends a single symbolic packet of configurable size to the server. Later on, the command processing functions are exercised with the symbolic packet and eventually a response will be written back to the symbolic file.

We targeted at the processing function for a common command for `Memcached`, the "get" command, used for retrieving one or more data objects currently stored in the cache server. The command takes the form of a space-delimited string that contains "get" or "bget" followed by a list of keys, IDs to address specific data objects. The command string is first split into an array of tokens. The tokens are then analyzed by the processing function for the "get" command. We targeted at the main loop in the processing function, `process_get_command`, as shown in Figure 5.2. In the explicit mode, we set a limit of 10 iterations so LANCET can explore different paths

Table 5.3
Examples of path conditions generated for `Memcached`. ID, number of iterations and path condition are listed for each generated test. A character or space represents an equality constraint for the byte where it appears. A '*' symbol represents a constraint that enforces a non-space character for the byte where it appears.

Test ID	**Iterations**	**Path Condition**
test000001	1	`get *`
test000006	2	`get * *`
test000007	3	`get ** * ****`
test000008	4	`get ** * * *`
test000023	5	`get *** * * *** **** ***`
test000036	6	`get *** * * **** * ** *`

thoroughly at small scales. After running in the explicit mode for around 10 hours, LANCET generated 590 tests that exercise the target loop between 1 and 10 iterations.

Table 5.3 shows some of the path conditions LANCET generated for `Memcached`. We applied the inference mode on these generated tests and successfully deduced the path condition for exactly 100 iterations by extrapolating the difference between test000001 and test000006. Note there are multiple ways to reach 100 iterations by inferring over the generated tests. For example, we were able to do the same based on test000007 and test000008. However, there also exist path conditions that lead to inaccurate inference if employed as the base cases for path condition extrapolation. For example, the incremental set of test000006 and test000007 is `'* * ****'` starting at the 6th byte of the input. The inference based on these two tests ends up with a path condition that exercises the target loop for 293 iterations. The reason for this inaccuracy is that LANCET only runs the explicit mode for a certain amount of time and could not cover every path for a given number of iterations. As a result, the incremental set between test000006 and test000007 is not the *minimal* incremental set between 2-iteration and 3-iteration path conditions.

5.5 Summary

Lancet is the first system that can generate accurate, targeted, large-scale stress tests for programs. Though it builds upon dynamic symbolic execution, it sidesteps many of the fundamental scaling problems of such techniques through a novel use of statistical inference, allowing Lancet to generate large-scale test inputs without having to run large-scale dynamic symbolic execution. Through a series of case studies, we have demonstrated that Lancet is general, efficient and effective.

6 RELATED WORK

6.1 Statistical Bug Detection and Diagnosis

There is a substantial amount of work concerning statistical debugging [4–6, 12, 14, 55–61]. Some of these approaches focus primarily on detection, with diagnosis as a secondary, often *ad hoc* capability [4–6, 14], while others focus primarily on automatically assisting bug diagnosis [12, 55–61].

The typical approach taken for detection by statistical approaches [4–6, 14] 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 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 from others [5]; this approach does not suffice for bugs which arise equally in all processes (such as our MPI case study).

Other statistical techniques eschew detection, in favor of attempting to debug programs that are known to have faults [12, 55–61]. 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 our techniques, the training runs are all known to be bug-free, but bug detection must be performed *given a single buggy run*. We are not attempting to debug widely distributed faulty programs that can generate a large number of sample points, but instead are attempting to localize bugs given a single instance of the bug. Hence, classification-based techniques are not appropriate for our setting. Some of the most relevant works in this domain is sampled in the rest of this section.

The first work in this domain that illuminates our work is that by Mirgorodskiy *et al.* [5], which applies to similarly behaving processes in an application. Behavioral data is collected for each process in the system, and an error is flagged if a process's behavior deviates from correct behavior (given known ground-truth data), or if its behavior is sufficiently different from other processes in the system.

The second relevant work in this domain is AutomaDeD [6]. This work provides a model to characterize the behavior of parallel applications. It models the the control flow and timing behavior of application tasks as Semi-Markov Models (SMMs) and detects faults that affect these behaviors. AutomaDeD detects errors by clustering tasks with similar SMMs together, and identifying tasks that do not fit into expected clusters. AutomaDeD then inspects the faulty SMMs to localize the bugs.

DMTracker [7] uses data movement related invariants, tracking the frequency of data movement and the chain of processes through which data moves. The premise of DMTracker is that these invariants are consistent across normal processes. Bugs are detected when a process displays behavior that does not conform to these invariants, and can be localized by identifying where in a chain of data movements the invariant was likely to be violated.

6.2 Performance Test Generation

The typical method to generate load tests is to induce load by increasing the input size (e.g., a larger query or a larger number of queries) or the rate at which input is provided (e.g., more query requests per unit of time) [62]. However, such techniques are not very discriminating with respect to the kind of load that is introduced, *e.g.,* the kind of query can make a big difference to the response time.

Previous work in workload synthesis [63–65] applies genetic algorithms to synthesize benchmarks for stress testing. The key idea is to form an abstract space parameterized on a finite number of workload characteristics such as instruction mix, instruction level parallelism, or working set size, and employ genetic algorithms to search for the optimal point with regard to a target metric such as execution time, energy consumption, or voltage fluctuation. Nevertheless, the application is treated as a black box that accepts the generated tests as input and emits the performance metrics as output. In the absence of correlation between tests and code, it is arduous to analyze the testing results and fix the issues found from running these tests.

Zhang *et al.* design a tool that generates tests that target certain types of load [66]. In their scheme, the user is asked for the metric that characterizes load, such as memory consumption. Based on this information, the tool searches for paths, using symbolic execution, that stress that load metric. For example, if the user is interested in memory consumption, then paths containing memory allocation actions are favored. In the background section, we have already presented discussion of another technique in this category, WISE [49], which generates worst-case inputs, *i.e.,* inputs that cause the program to run the longest. FOREPOST [67] uses runtime monitoring for a short duration of testing. The data that is collected is fed through a machine learning algorithm and automated test scripts provide insights into properties of test input data that lead to increased computational loads.

There are some solutions that apply dynamic techniques to detect problems with a loop, such as, the loop will never terminate, or that the loop has a performance

issue. For example, [68] describes Looper, which dynamically analyzes a program to detect non-termination. For this, it uses symbolic execution to generate a non-terminating path and then uses the SMT solver, to create concrete variables that will cause such non-termination, if such a situation exists. Some techniques like [69] seek to cure such loop problems.

7 CONCLUSION

This thesis focuses on addressing an important class of software defects, *scale-dependent bugs*, which have a deteriorated impact on the correctness and performance of applications as the system and input scale up. We built VRISHA, ABHRANTA, and WUKONG to detect and diagnose scaling bugs for large scale systems, and developed LANCET to generate test inputs that exhibit scaling trends of a given application to facilitate the above bug detection and localization approaches that need test inputs to work. Statistical inference and compiler techniques, such as dynamic instrumentation and symbolic execution, have been applied throughout these works. We evaluated our techniques with real bugs and actual production code, such as MPICH2, Transmission, SPEC2006 and Gnu Coreutils. All our works are released as open source software for the benefit of the research community.

The works in this thesis constitute only the first step towards building effective defense and offense mechanism for scaling bugs, the most notorious bugs that break the scalability of software in production systems. There are still a lot of challenges and open problems that need to be address in future works. We have discussed some of the limitations and shortcomings of our techniques in their respective chapters. Here, we will summarize the most critical restrictions and provide guidance over the directions for future exploration.

7.1 Data Dependence in Scaling Behavior Prediction

Data dependence describes the situation where the behavior of a program depends on certain values contained in the input. For example, the execution of a sparse matrix multiplication program may very well depend on the number of non-zero elements contained in the input matrices, not just the size of the matrices. Not like size, or

other *control* parameters, which we have used as the determining factors for application behaviors in our prediction models, values in the input data are information of much lower levels and difficult to interpret and model. Still taking the sparse matrix multiplication program as an example, if we just take every byte in the input as a feature and try to see how the value of each byte influences the behavior of the entire application, we will reach nowhere since it is the total number of non-zero values, an aggregated feature over the entire matrix, that determines the behavior of the application. What we need here is a way to extract or abstract high-level features, e.g. the number of non-zeros in the above example, which have more interpretable relation with the application's behaviors, from the ensemble of data values in the input.

However, a naive high-level feature extraction technique would base itself on the specific semantic of the internal algorithms of the given application and the structure of the input data, which would prevent it from being generalized to other applications. A potential general solution to this feature extract problem is to leverage dynamic code instrumentation of the input processing module of a given application and record values of certain critical variables as the desired high-level features, then use them to predict the behavior of the rest of the execution. Now the users only need to identify the code modules that process the input and the critical variables that store high-level information useful for predicting the behavior of application.

7.2 Environment Modeling in Symbolic Execution

In order to generate tests for real-world programs that rely on the system environment, e.g. libraries, system calls, file system, network, etc., symbolic execution engines need to include support for the executing external code symbolically. A common method to achieve this is to build a *model*, a simplified implementation of the external function that understands the semantics of the operation well enough to generate the required constraints. In practice, this is done manually for common libraries and system calls [46], which leads to our second future research direction:

automating the environment model building process. This problem is closely related to program synthesis with respect to a given specification. A simpler problem is to derive the symbolic version of a library from the original concrete version by redirecting all external dependencies into the counterpart modules provided by the symbolic execution engine.

LIST OF REFERENCES

LIST OF REFERENCES

[1] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[2] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[3] Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz. Scalable temporal order analysis for large scale debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 44:1–44:11, New York, NY, USA, 2009. ACM.

[4] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit. Lessons learned at 208k: Towards debugging millions of cores. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 26:1–26:9, Piscataway, NJ, USA, 2008. IEEE Press.

[5] Alexander V. Mirgorodskiy, Naoya Maruyama, and Barton P. Miller. Problem diagnosis in large-scale computing environments. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[6] Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R. de Supinski, Dong H. Ahn, , and Martin Schulz. AutomaDeD: Automata-based debugging for dissimilar parallel tasks. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks,*, DSN '10, pages 231–240, 2010.

[7] Qi Gao, Feng Qin, and Dhabaleswar K. Panda. DMTracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 15:1–15:12, New York, NY, USA, 2007. ACM.

[8] Zhezhe Chen, Qi Gao, Wenbin Zhang, and Feng Qin. FlowChecker: Detecting bugs in mpi libraries via message flow checking. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[9] http://www.mcs.anl.gov/research/projects/mpich2/.

[10] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[11] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 319–329, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael 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] Xing Wu and Frank Mueller. ScalaExtrap: Trace-based communication extrapolation for spmd programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 113–122, New York, NY, USA, 2011. ACM.

[14] Bowen Zhou, Milind Kulkarni, and Saurabh 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*, HPDC '11, pages 85–96, New York, NY, USA, 2011. ACM.

[15] Bowen Zhou, Milind Kulkarni, and Saurabh Bagchi. ABHRANTA: Localizing bugs that manifest at large system scales. In *Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability*, HotDep'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

[16] Wei-Wen Feng, Byung-Uck Kim, and Yizhou Yu. Real-time data driven deformation using kernel canonical correlation analysis. In *Proceedings of ACM SIGGRAPH 2008*, SIGGRAPH '08, pages 91:1–91:9, New York, NY, USA, 2008. ACM.

[17] Bowen Zhou, Jonathan Too, Milind Kulkarni, and Saurabh Bagchi. WUKONG: Automatically detecting and localizing bugs that manifest at large system scales. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 131–142, New York, NY, USA, 2013. ACM.

[18] Eliot Horowitz. Foursquare outage post mortem. https://groups.google.com/forum/#!topic/mongodb-user/UoqU8ofp134.

[19] Konstantin V Shvachko. HDFS scalability: The limits to growth. *USENIX login*, 35(2):6–16, 2010.

[20] http://bugs.mysql.com/bug.php?id=49177.

[21] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61, February 2012.

[22] Francis R. Bach and Michael I. Jordan. Kernel independent component analysis. *Journal of Maching Learning Research*, 3:1–48, March 2003.

[23] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.

[24] Harold Hotelling. Relations between two sets of variates. *Biometrika*, 28(3/4):321–377, 1936.

[25] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[26] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 592–603, Washington, DC, USA, 2009. IEEE Computer Society.

[27] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. NAS parallel benchmark results. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, pages 386–393, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[28] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19(1):49, 2005.

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

[30] `https://trac.mcs.anl.gov/projects/mpich2/browser/mpich2/trunk/src/mpi/coll/allgatherv.c`.

[31] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. RNR-91-002, NASA Ames Research Center, August 1991.

[32] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 4–4, Berkeley, CA, USA, 2010. USENIX Association.

[33] `http://trac.mcs.anl.gov/projects/mpich2/ticket/1005`.

[34] Satish Balay, Jed Brown, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc web page, 2009. `http://www.mcs.anl.gov/petsc`.

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

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

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

[38] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716 – 723, dec 1974.

[39] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning (2nd edition)*. Springer-Verlag, 2008.

[40] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 368–377, New York, NY, USA, 2008. ACM.

[41] Max Kuhn and Kjell Johnson. An introduction to multivariate modeling techniques. `http://zoo.cs.yale.edu/classes/cs445/slides/Pfizer_Yale_Version.ppt`.

[42] `https://asc.llnl.gov/sequoia/benchmarks/`.

[43] Kuan-Yu Tseng, D. Chen, Z. Kalbarczyk, and R.K. Iyer. Characterization of the error resiliency of power grid substation devices. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '12, pages 1–8, June 2012.

[44] Michael D. Bond, Graham Z. Baker, and Samuel Z. Guyer. Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 13–24, New York, NY, USA, 2010. ACM.

[45] Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 97–112, New York, NY, USA, 2007. ACM.

[46] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[47] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[48] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.

[49] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 463–473, Washington, DC, USA, 2009. IEEE Computer Society.

[50] `http://memcached.org/`.

[51] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[52] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 183–198, New York, NY, USA, 2011. ACM.

[53] `http://clang.llvm.org/`.

[54] `http://llvm.org/docs/BitCodeFormat.html`.

[55] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael 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.

[56] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex 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.

[57] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32:831–848, October 2006.

[58] Laura Dietz, Valentin Dallmeier, Andreas Zeller, and Tobias Scheffer. Localizing bugs in program executions with graphical models. In *Proceedings of Advances in Neural Information Processing Systems 22*, NIPS 22, pages 468–476, 2009.

[59] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44, Washington, DC, USA, 2009. IEEE Computer Society.

[60] David Andrzejewski, Anne Mulhern, Ben Liblit, and Xiaojin Zhu. Statistical debugging using latent topic models. In *Proceedings of the 18th European Conference on Machine Learning*, ECML '07, pages 6–17, Berlin, Heidelberg, 2007. Springer-Verlag.

[61] James A. Jones and Mary Jean 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.

[62] Rick Hower. Web site test tools and site management tools: Load and performance test tools. `http://www.softwareqatest.com/qatweb1.html#LOAD`.

[63] Ajay M. Joshi, Lieven Eeckhout, Lizy Kurian John, and Ciji Isen. Automated microprocessor stressmark generation. In *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture*, HPCA-14, pages 229–239, Feb 2008.

[64] Youngtaek Kim, Lizy Kurian John, Sanjay Pant, Srilatha Manne, Michael Schulte, W. Lloyd Bircher, and Madhu S. Sibi Govindan. AUDIT: Stress testing the automatic way. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 212–223, Washington, DC, USA, 2012. IEEE Computer Society.

[65] Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 241–252, New York, NY, USA, 2010. ACM.

[66] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society.

[67] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 156–166, Piscataway, NJ, USA, 2012. IEEE Press.

[68] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 161–169, Washington, DC, USA, 2009. IEEE Computer Society.

[69] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. Bolt: On-demand infinite loop escape in unmodified binaries. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 431–450, New York, NY, USA, 2012. ACM.

VITA

VITA

Bowen Zhou started as a Ph.D student in the Computer Science Department of Purdue University in 2008, doing research on automated software debugging under the guidance of Professor Milind Kulkarni and Professor Saurabh Bagchi. He obtained two Computer Science degrees before coming to Purdue, including a B.E. from the University of Science and Technology of China in 2004 and an M.E. from the Chinese Academy of Sciences in 2007.