WuKong: Automatically Detecting and Localizing Bugs that Manifest at Large System Scales

Bowen Zhou, Jonathan Too, Milind Kulkarni, Saurabh Bagchi
Purdue University
West Lafayette, IN
{bzhou, jtoo, milind, sbagchi}@purdue.edu

ABSTRACT

A key challenge in developing large scale applications is finding bugs that are latent at the small scales of testing, but manifest themselves when the application is deployed at a large scale. Here, we ascribe a dual meaning to "large scale"—it could mean a large number of executing processes or applications ingesting large amounts of input data (or both). Traditional statistical techniques fail to detect or diagnose such kinds of bugs because no error-free run is available at the large deployment scales for training purposes. Prior work used *scaling models* to *detect* anomalous behavior at large scales without training on correct behavior at that scale. However, that work *cannot localize* bugs automatically, *i.e.,* cannot pinpoint the region of code responsible for the error. In this paper, we resolve that shortcoming by making the following three contributions: (i) we develop an automatic diagnosis technique, based on *feature reconstruction*; (ii) we design a heuristic to effectively prune the large feature space; and (iii) we demonstrate that our system scales well, in terms of both accuracy and overhead. We validate our design through a large-scale fault-injection study and two case-studies of real-world bugs, finding that our system can effectively localize bugs in 92.5% of the cases, dramatically easing the challenge of finding bugs in large-scale programs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Statistical Methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics*

General Terms

Reliability

Keywords

Scale-dependent Bug, Program Behavior Prediction, Feature Reconstruction

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HPDC'13, June 17–21, 2013, New York, NY, USA.
Copyright 2013 ACM 978-1-4503-1910-2/13/06 ...$15.00.

1 Introduction

One of the key challenges in developing large-scale software, i.e., software intended to run on many processors or with very large data sets, is detecting and diagnosing *scale-dependent* bugs. Most bugs manifest at both small and large scales, and as a result, can be identified and caught during the development process, when programmers are typically working with both small-scale systems and small-scale inputs. However, a particularly insidious class of bugs are those that predominantly arise at deployment scales. These bugs appear far less frequently, if at all, at small scales, and hence are often not caught during development, but only when a program is released into the wild and is deployed at large scales. As one example of this class of bugs, there is a performance bug in one version of the popular parallel programming library, MPICH2, that arises when the total amount of data being exchanged between the processes of the parallel application is large [1]. The root cause of the bug is overflow of a 32-bit integer variable.

Interestingly, even if a programmer has access to large-scale inputs and systems, detection and diagnosing scale-dependent bugs can be intractable. As system size and complexity increase, correctly attributing anomalous behavior to a part of a program can be quite challenging. If a bug causes a program to crash at large scale runs, it is obvious that the error lies with the program, and hence debugging efforts can focus on locating (and then fixing) the bug in the program. For bugs that do not have a clear symptom, it may not even be apparent that there *is* a bug in the program. For example, if a bug does not manifest at small scales but arises deterministically at large scales, there is no example of correct behavior at large scales to determine that the observed behavior is anomalous.

Once a bug is detected, the next challenge is to localize[1] the bug. Programs meant to scale up often have large code bases, so simply identifying that there is a bug is not sufficient to fix the bug. Instead, the developer would like to know *where* the bug arose: which module, function, or even line number. Unfortunately, performing this localization manually for large-scale systems is overwhelming. Even if the point where the bug *manifests* can be identified (*e.g.*, by examining a stack dump after a program crashes), this location may be far removed from the source of the bug. Even worse is when the bug does not cause the program to crash, but instead results in incorrect results; in the absence of in-

[1] We use the terms "diagnosis" and "localization" interchangeably; both are defined as the action of pinpointing the root cause of the detected error to a region of code.

formation associated with a crash, identifying the source of the error requires examining every possible program location that may be buggy. For example, a hang bug in a popular P2P file-sharing program called Transmission manifests only when the number of peers overflows a fixed-size buffer. However, traditional debugging techniques (*e.g.,*OPROFILE) can at best indicate that a particular method is running for a long time, not that this is a bug. Identifying that the long running time constitutes a bug requires understanding the behavior of the method and *localizing* the bug (even if the method is known to be buggy) requires investigating 252 lines of code. In contrast, a more accurate tool (such as the system we develop) can correctly localize the bug to a specific, 27-line loop.

1.1 Statistical bug diagnosis

A popular approach to detecting and diagnosing bugs is *statistical* debugging. At a high level, a statistical model is built using a set of training runs, which are expected to be *not* affected by the bug. This model captures the expected behavior of a program by modeling the values of selected program *features* (*e.g.*, the number of times a particular branch is taken, or the number of times a particular calling context appears). Then, at deployment time, values of these same features are collected at different points in the execution; if the values of these features fall outside the model parameters, the deployed run represents program behavior different from the training runs and a bug is detected. By examining which program features are deviant, the bug can be localized (to different levels of accuracy; the deviant program feature may just be the one that is most affected by the bug and not be related to the root cause). This approach has been taken by numerous prior bug-detection tools [9, 12, 21].

However, the traditional statistical debugging approach is insufficient to deal with scale-dependent bugs. 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 and are triggered in *every* execution at large scales. Thus, it would be impossible to get *any* sample of bug-free behavior at large scales for training purposes.

A further 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 Detecting scale-dependent bugs

Prior work by us [25] has attempted to address the drawbacks of existing statistical debugging techniques. VRISHA is a tool that exploits *scale-determined* features to detect bugs in large-scale program runs even if the statistical model was only trained on small-scale behavior. At a high level, the intuition behind VRISHA's operation is as follows. Several training runs are collected at different small scales. A statistical model (based on Kernel Canonical Correlation Analysis, or KCCA) is then built from these training runs to infer the relationship between scale and program behavior. In essence, we build a *scaling model* for the program, which can extrapolate the aggregated behavioral trend as the input or system size scales up. Bugs can then be automatically detected by identifying deviations from the trend. Notably, this detection can occur *even if the program is run at a never-before-seen scale.*

Unfortunately, VRISHA suffers from two key drawbacks. First, while KCCA is useful for being able to capture non-linear relationships between scale and behavior, its accuracy decreases as the scales seen in deployment runs become significantly larger than the scales used in training. Hence, VRISHA loses effectiveness when attempting to detect bugs in large-scale systems, primarily reflected in an increase in its false alarms (we show this empirically in Section 5.1).

Second, and more importantly, while the detection of bugs is automatic, VRISHA is only able to identify that the scaling trend has been violated; it cannot determine *which* program feature violated the trend, nor *where* in the program the bug manifested. *Hence, diagnosis in* VRISHA *is a manual process.* The behavior of the program at the various small scales of the training set are inspected to predict expected values for each individual program feature 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 ineffective under many real-world scenarios. First, if different aspects of program behavior are related to scale through different functions (linear, quadratic, etc.), then each feature must be considered separately when attempting to infer a scaling trend. This inference can be intractable when the feature set is large, as it is for reasonable-sized programs and in situations where many of its features need to be considered to achieve reasonable detection coverage. 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 impractical. Finally, some features may not be well-correlated with scale at all, such as the delay in network communication, which will depend on factors external to the particular application being debugged. Without a means to identifying such problematic features, VRISHA will falsely flag them as being erroneous, leading developers on a wild-goose chase.

In follow-on work, ABHRANTA, we tweaked VRISHA's program model to automatically diagnose bugs [24]. However, because ABHRANTA is based on the same KCCA technique as VRISHA, and does not attempt to deal with problematic features, its accuracy precipitously declines as the scale of production runs increases, leading to increasingly inaccurate diagnoses. Consequently, for large-scale runs, it is still necessary to resort to manual inspection to diagnose bugs.

Hence, it is fair to say that neither VRISHA *nor* ABHRANTA *support automated diagnosis of bugs in large-scale systems.*

1.3 Our approach: WuKong

This paper presents WuKong[2], an *automatic, scalable approach to detecting and diagnosing bugs that manifest at large system scales.* WuKong is designed to model program behavior as a group of *features*, each representing the behavior of a particular part of the program in an execution.

In a typical usage scenario, WuKong is deployed in production runs, after a program has been thoroughly tested at small scales, to detect and diagnose bugs. WuKong works towards diagnosing a bug in three steps. First, during development a series of small-scale bug-free runs are analyzed to derive per-feature behavioral models for the program. Second, during production runs, the behavioral models predict what the correct value for each feature at production scale would be if the large-scale run were bug-free. Finally, the actual value and the predicted value of each feature are compared; if any features behave differently from the prediction, a bug is detected, and the most deviant features are identified as the probable root causes for the bug. Since every feature can be attributed to a particular part of the program, it is straightforward to map suspicious features to locations in the source code of the program.

WuKong is based on the same high level concepts as Vrisha and Abhranta, but 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 3), 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.

[2]WuKong, the Monkey King, is the main character in the epic Chinese novel *Journey to the West*, and possesses the ability to recognize evil in any form.

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.

1.4 Outline

Section 2 presents the data collection of WuKong. Section 3 describes WuKong's new approach to modeling the scaling behavior of programs, including our feature pruning strategy. Section 4 discusses how WuKong uses these models to detect and localize bugs. Section 5 demonstrates, through several case studies, the utility of WuKong for automatically diagnosing bugs, and shows, through a fault-injection study, that WuKong scales effectively to 1024 processes. Section 6 discusses related work, and Section 7 concludes.

2 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.

2.1 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 configu-

ration 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 instruction, 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* $\rightarrow$ *not taken* and *not taken* $\rightarrow$ *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.

2.2 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 3.4 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.

3 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.

3.1 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 3.2 and 3.3 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 3.4 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 elaborates further.

3.2 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:

$$Y = \beta_0 + \sum_{i=1}^{N} \beta_i \cdot X_i \quad (1)$$

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:

$$\log(Y) = \beta_0 + \sum_{i=1}^{N} \beta_i \log(X_i) \quad (2)$$

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 $\beta_0 ... \beta_N$:

$$\underset{\beta_0,\ldots,\beta_N}{\arg\min} \,||\log(Y) - \sum_{i=1}^{N} \beta_i \log(X_i) - \beta_0||^2 \quad (3)$$

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 [25], 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.

3.3 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) [4], a measure of relative goodness of fit in a statistical model given by:

$$AIC = -2\ln(L) + 2k \quad (4)$$

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 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 [13]. 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.

3.4 Feature Selection and Pruning

As described in Section 2, 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 3.2 allows us to readily compute the relative prediction error for a given feature, which we require to identify faulty features (see Section 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.* [6]:

$$E_i = |e^{\log(Y_i') - \log(Y_i)} - 1| \quad (5)$$

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 WuKong's regression models, 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 [13]. 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 [15]:

- 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 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[3]. In other words, WuKong uses its models to predict the *expected* behavior of the program at large scale. These predictions are then used to detect and diagnose bugs, as described below.

[3]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.

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 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

$$E_i > \eta M_i \qquad (6)$$

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 5.1 shows how changing η affects false positive and negative rates.

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 5'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.

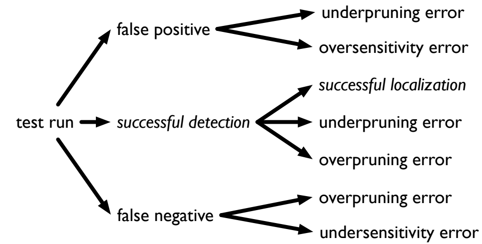

Figure 1: Possible outcomes and errors when using WuKong to detect and diagnose bugs.

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 5 explore these effects empirically. Figure 1 shows the possible outcomes and error types that can occur 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.

5 Evaluation

This section describes our evaluation of WuKong. We implemented WuKong using PIN [20] to perform dynamic binary instrumentation. To collect the features as described in Section 2, 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 [2]. 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 [25]. 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.

5.1 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 3.4, 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 [25], 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 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 scales.* Hence, WuKong's regression models are effective at predicting the large scale

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%

Table 1: Scalability of WuKong for AMG2006 on test runs with 256, 512 and 1024 nodes.

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 [22]).

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 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.

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 4.2 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 3 shows how many

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 2: The accuracy of detection at various levels of detection threshold with a 90% pruning threshold.

features were filtered during pruning, the false positive rate of detection, the false negative rate 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 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 4.3 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.

5.2 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 [25], 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 all-

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%

Table 3: The accuracy and precision of detection and localization at various levels of feature pruning with detection threshold parameter $\eta = 1.15$.

```
if ((recvcount*comm_size*type_size < MPIR_ALLGATHER_LONG_MSG)
     && (comm_size_is_pof2 == 1)) {
/*** BUG IN ABOVE CONDITION CHECK DUE TO OVERFLOW ***/
/* ALGORITHM 1 */
...
} else if (...) {
/* ALGORITHM 2 */
...
} else {
/* ALGORITHM 3 */
...
}
```

Figure 2: MPICH2 bug that manifests at large scale as performance degradation.

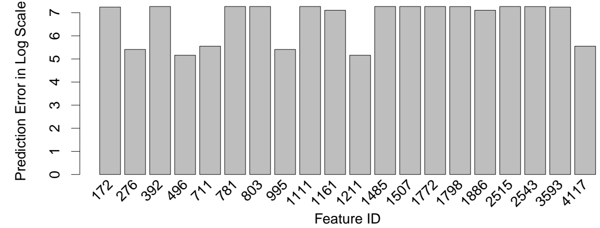

Figure 3: The top suspicious features for the buggy run of MPICH2 ALLGATHER given by WuKong.

```
while (1) {
  l = strtol((char*)buf + i, &q, 10);
  if(q && *q == ':' && l > 0) {
    if(j + l > MAX_VALUES_SIZE)
      continue;
    /*** BUG: i INCREMENT IS SKIPPED ***/
    i = q + 1 + l - (char*)buf;
    ...
  } else {
    break;
  }
}
```

Figure 4: The deadlock bug appears in Transmission, and manifests when a large number of peers are contained in a single DHT message.

GATHER in MPICH2 (before v1.2) contains an integer overflow bug [1], 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 3.4. 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 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 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 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!

5.3 Case Study 2: Deadlock in Transmission

Transmission is a popular P2P file sharing application on Linux platforms. As illustrated in Figure 4, the bug [3] 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 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.

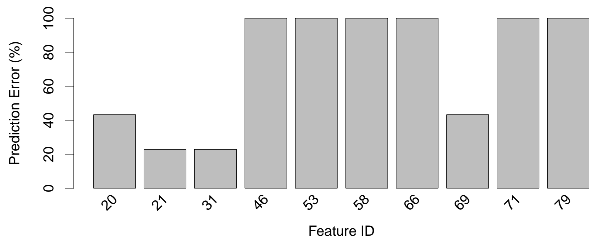

Figure 5: The top suspicious features for the buggy run of Transmission given by WuKong.

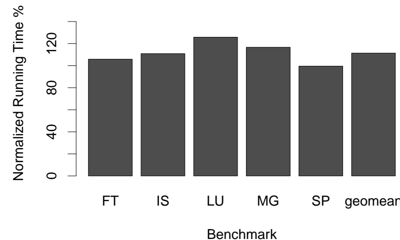

Figure 6: Runtime overhead of WuKong on NPB benchmarks.

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 5. The rank ordering of deviant features highlights Features 53 and 66, which correspond to the line `if (q && *q == ':' && l > 0)` at the beginning of Figure 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.

5.4 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[4]. All benchmarks are compiled in a 64-process configuration and each is repeated 10 times to get an average running time. Figure 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 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 [7] and its predecessor called Probabilistic Calling Context (PCC) [8], both of which allow for efficient recording of dynamic calling contexts. Breadcrumbs builds 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) callsites. We defer to future work the implementation of these techniques in WuKong to capture calling contexts at even lower overheads.

[4]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.

6 Related work

There is a substantial amount of work concerning statistical debugging [5, 9–11, 14, 16–19, 21, 23, 25]. Some of these approaches focus primarily on detection, with diagnosis as a secondary, often *ad hoc* capability [9, 16, 21, 25], while others focus primarily on assisting bug diagnosis [5, 10, 11, 14, 17–19, 23].

The typical approach taken for detection by statistical approaches [9, 16, 21, 25] is to characterize a program's behavior as an aggregate of a set of features. A model is built based on the 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, resulting in false positives. Some approaches mitigate this by also detecting bugs in parallel executions if some processes behave differently from others [21]; this approach does not suffice for bugs which arise equally in all processes (such as our MPICH2 case study).

Other statistical techniques eschew detection, in favor of attempting to debug programs that are known to have faults [5, 10, 11, 14, 17–19, 23]. 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 (a) they 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 WuKong, 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.

The closest prior work is our own—ABHRANTA [24], which shares part of the goal of this paper, namely, to localize bugs that appear at large scales. However, it uses the same modeling strategy as VRISHA, Kernel Canonical Correlation Analysis (KCCA), simplifying some portions of the model to ease the task of localization. To be specific, it uses only linear functions for the canonical correlation analysis for the mapping of the observational features. Since it uses KCCA (or CCA in parts), it suffers from the same issue—that its model becomes increasingly inaccurate as the difference increases between the scales of the training runs and those of the production runs. Further, since WUKONG uses the simpler regression model, it runs less risk of overfitting and is conceptually simpler to deal with. Finally, ABHRANTA did not consider the fact that some features are not dependent on scale or otherwise unable to be modeled and should therefore be pruned prior to the analysis. Thus, it is fair to say that ABHRANTA realized only part of the goal of the current paper and can perform reasonably well only in relatively small scale systems without overly-complex behaviors.

7 Conclusions

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 approach 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.

Acknowledgments

We would like to acknowledge the Extreme Science and Engineering Discovery Environment (XSEDE), supported by National Science Foundation grant number OCI-1053575, for use of the clusters for experiments. We would like to acknowledge GrammaTech for providing us an academic license for CodeSurfer, a static analysis tool for C/C++.

References

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

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

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

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

[5] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In *ECML '07*.

[6] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *ICS '08*.

[7] M. Bond, G. Baker, and S. Guyer. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *PLDI '10*.

[8] M. Bond and K. McKinley. Probabilistic calling context. In *OOPSLA '07*.

[9] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, , and M. Schulz. AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks. In *DSN '10*.

[10] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE '09*.

[11] L. Dietz, V. Dallmeier, A. Zeller, and T. Scheffer. Localizing bugs in program executions with graphical models. In *NIPS '09*.

[12] Q. Gao, F. Qin, and D. Panda. DMTracker: Finding Bugs in Large-Scale Parallel Programs by Detecting Anomaly in Data Movements. In *SC '07*.

[13] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning (2nd edition)*. Springer-Verlag, 2008.

[14] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05*.

[15] M. Kuhn and K. Johnson. An Introduction to Multivariate Modeling Techniques. `http://zoo.cs.yale.edu/classes/cs445/slides/Pfizer_Yale_Version.ppt`.

[16] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons Learned at 208K: Towards Debugging Millions of Cores. In *SC '08*.

[17] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03*.

[18] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05*.

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

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*.

[21] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem diagnosis in large-scale computing environments. In *SC '06*.

[22] K. Tseng, D. Chen, Z. Kalbarczyk, and R. Iyer. Characterization of the error resiliency of power grid substation devices. In *DSN '12*.

[23] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML '06*.

[24] B. Zhou, M. Kulkarni, and S. Bagchi. Abhranta: Locating bugs that manifest at large system scales. In *HotDep '12*.

[25] B. Zhou, M. Kulkarni, and S. Bagchi. Vrisha: using scaling properties of parallel programs for bug detection and localization. In *HPDC '11*.