Dealing with the Unknown: Resilience to Prediction Errors

Subrata Mitra[†], Greg Bronevetsky[‡], Suhas Javagal[†], Saurabh Bagchi[†]
[†] Purdue University, [‡] Google

***Abstract*—Accurate prediction of applications' performance and functional behavior is a critical component for a wide range of tools, including anomaly detection, task scheduling and approximate computing. Statistical modeling is a very powerful approach for making such predictions and it uses observations of application behavior on a small number of training cases to predict how the application will behave in practice. However, the fact that applications' behavior often depends closely on their configuration parameters and properties of their inputs means that any suite of application training runs will cover only a small fraction of its overall behavior space. Since a model's accuracy often degrades as application configuration and inputs deviate further from its training set, this makes it difficult to act based on the model's predictions.**

This paper presents a systematic approach to quantify the prediction errors of the statistical models of the application behavior, focusing on extrapolation, where the application configuration and input parameters differ significantly from the model's training set. Given any statistical model of application behavior and a data set of training application runs from which this model is built, our technique predicts the accuracy of the model for predicting application behavior on a new run on hitherto unseen inputs. We validate the utility of this method by evaluating it on the use case of anomaly detection for seven mainstream applications and benchmarks. The evaluation demonstrates that our technique can reduce false alarms while providing high detection accuracy compared to a statistical, input-unaware modeling technique.

***Keywords*-Prediction error characterization, extrapolation error, performance estimation, anomaly detection, statistical modeling, software performance bugs, input-aware technique**

I. Introduction

Techniques to predict the dynamic properties of application executions are a critical part of various tools. For example, schedulers need to predict an application's execution time and resource use (*e.g.*, network bandwidth requirements), while anomaly detection tools need to differentiate an application's normal behavior from anomalous behavior that indicate software or hardware problems. Similarly, performance profiling tools need to describe how various parts of an application utilize system resources to enable developers to focus their code optimization efforts. The key capability required by these tools is to predict, before an application executes, various metrics of its execution (*e.g.*, total execution time, energy use, cache miss counts and code execution paths), both at the granularity of the whole application as well as for individual code segments (*e.g.*, function calls and loops).

[†]{mitra4, sjavagal, sbagchi}@purdue.edu
[‡]bronevet@google.com. Part of this work was done while at Lawrence Livermore National Lab.

Prior work on statistical techniques to make such predictions has demonstrated their utility in the design of real tools [1], [2]. These techniques observe multiple runs of the target application, or of individual code regions (a given code region may be executed multiple times in a single application run) to build a statistical model of the various metrics that are observed during these runs. These models are then used to either predict the values of these metrics for the future application runs or to determine whether a given metric value is consistent with a normal application execution or is somehow anomalous. *A key observation of prior research is that to predict these metrics accurately, modeling techniques must take into account properties of application input, configuration and state* (denoted collectively here as "parameters"). For example, to predict the execution time of a shortest-paths graph algorithm it is necessary to know properties of the graph, such as its diameter.

However, the key challenge that limits the use of these techniques in practice is that the number of application parameters that control application behavior is large and their values may span a large range. Hence, for a reasonable-sized training set for the statistical techniques, it can only cover a very small fraction of the overall parameter space. This means that most predictions that will be made in reality will be on application parameters that are outside the predictor's training set. Expectedly, the accuracy with which a model predicts the behavior of a given application decreases as the run grows more different from the runs the model was trained on. For example, when an application runs in production environment, it may use a larger scale and a larger and different characteristic dataset. The application behavior then shows large deviations from that predicted by the model. Hence, we have to either ignore the predictions of the model altogether or run the risk of high false alarms or poor resource utilization, depending on the use case for the prediction. As a simplified example of a real bug, consider the situation in Figure 1. Here, an unseen value of a command-line parameter (q) causes a performance anomaly. During software testing, the values of q used along with calculated values of r from input-data leads to moderate values of `iterCount` which control how many times a compute intensive routine `doMoreCalculations` will be invoked.

What we would rather like to have is a way to characterize the accuracy of the prediction as the parameters deviate from those seen during the training runs. Armed with such a characterization, we can do the equivalent of "uncertainty quantification" for our prediction. We could, for example,

Code for a hypothetical program:

```
1.  void main (argc, argv){

2.     int p = readInput()
3.     int q = argv[1];
4.     int r = calculationsOnInput(p);
5.     float s = otherCalculations()
6.     float someVal = (r % q) + 0.001;
7.     int iterCount = s / someVal;
8.     for (int i=0; i < iterCount; i++){
9.           doMoreCalculations();
       }
     }
```

Testing values:

q	Calculated r	Calculated s	SomeVal	iterCount
2	21	10	1.001	9
3	14	20	2.001	9
5	28	30	3.001	9
40	85	40	5.001	7

Production values:

>$ ./prog –arg1=20 #performance anomaly

q	Calculated r	Calculated s	someVal	iterCount
20	40	10	0.001	**10000**

Fig. 1: A program takes a command-line parameter (stored in q) and reads an input data file. For certain combination of q and input data, the calculated iteration bound of a loop containing a heavy calculation becomes very large, leading to a performance anomaly.

put statistically rigorous error bounds on the prediction. This translates to usable characterization of uncertainty in the various use cases. For the anomaly detection use case, this could allow the user to set application-specific bounds on the rates of false alerts and missed alerts. For performance analysis tools, this could let the user set bounds on the resource utilization of a region of the code.

In this paper, we provide a method to precisely address this requirement, *i.e.*, quantify the accuracy of the prediction. It describes a technique to use a training set of limited number of application runs that cover a small fraction of a target application's parameter space to train an arbitrary regression model that predicts the application's behavior across this entire input parameter space. Further, it quantifies the errors of these predictions as a function of how "different" they are from the training runs. The input to our technique is a set of observations of the parameters and the behavioral metrics of the executions of the entire applications or individual code regions[1]. Examples for behavioral metrics are execution time, number of instructions/floating point instructions, percentage of conditional branches taken, etc. Our technique then trains regression models using this input dataset. Such models are used to predict the behavioral metrics of code regions given certain combination of input parameters. Our first contribution is a technique to create a distribution of the model's prediction error for parameter values *within* the region of the parameter space that the model was trained on. Our second contribution is a method to extrapolate the prediction error outside this region. This is accomplished by considering one parameter at a time and creating the error characterization for that parameter. We then combine the error characterizations when all the parameter values deviate from what had been seen during the training runs.

We demonstrate the utility of our approach by creating a concrete performance anomaly detection tool, called GUARDIAN. We evaluate the use case of anomaly detection using seven popular applications, drawn from a diversity of domains — scientific simulations, matrix-vector algebra, graph search, and financial options trading.

[1]Whether we will use a single model for the entire application or have one for arbitrarily-defined code regions is a choice that depends on how homogeneous the behavior of the application is, across its different code regions.

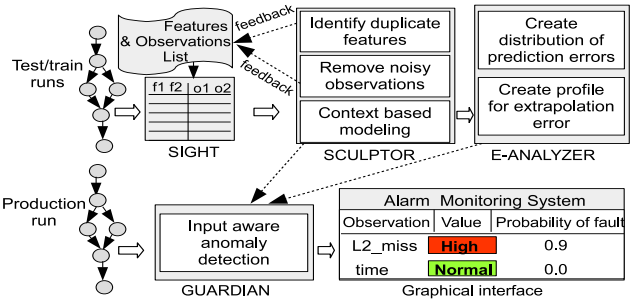


Fig. 2: Workflow of anomaly detection system

Figure 2 shows the complete workflow of the anomaly detection system composed of data-collection framework (SIGHT), SCULPTOR (modeling tool), E-ANALYZER (error analysis tool) and GUARDIAN (anomaly detection engine). The toolchain is composed of

- SIGHT, which tracks application code regions, their parameters and behavioral metrics,
- SCULPTOR, which builds a statistical model for each code region that predicts the region's behavioral metrics from its parameters,
- E-ANALYZER, which quantifies the prediction error of SCULPTOR, and
- GUARDIAN, which detects anomalies in application behavior by comparing actual observations of code region behavioral metrics to their values predicted by SCULPTOR, accounting for the prediction error computed by E-ANALYZER.

Experimental evaluations demonstrate that by using error predictions computed by E-ANALYZER, GUARDIAN is able to achieve false positive rates below 2% which is a 94% improvement over over anomaly detection techniques that ignore model prediction error. Further, our studies that inject synthetic faults into application execution show that this improvement is achieved without reducing the detection accuracy relative to these alternative techniques.

The rest of the paper is organized as follows: Section II briefly talks about our data-collection framework. Section III presents the modeling technique. In Section IV we introduce error analysis techniques. Section V discusses about anomaly detection as an use-case. Section VI presents results of our evaluation. Section VII studies related works. Finally, Section VIII concludes.

II. DATA COLLECTION FRAMEWORK

Example of SIGHT APIs:

```
float func (float x, float y) {
   SightModule funcMod("func", inputs("x",x,"y",y));  // creating a module for the scope of this function
   DoCalculations();                // some computation done by the function
   r = calculateResidual ();   // function computes a residual value to verify convergence
   funcMod.outputs("r", r);    // SIGHT tracks the residual as an output observation
}
```

Fig. 3: SIGHT instrumentation, context aware modules.

We collect parameters and behavioral metrics at the granularity of arbitrary application code regions (denoted "modules"), which may include code blocks, functions or the entire

application. This data is collected using a data collection tool we developed called SIGHT, which provides simple APIs for developers to annotate the modules to be modeled and specify the information to be collected for each one. Section III-A details the parameters and behavior metrics we collect as part of our analysis. Figure 3 shows an example of SIGHT annotation APIs that user may use to mark a module (the duration of function `func`'s module is the lifetime of variable `funcMod`), report its parameters (`x`, `y`) and behavior metrics (output `r` and performance metrics). Modules are tracked in a context-aware manner, collecting observations separately for a single module depending on the calling context it is executed in. Currently module annotation and specification of module inputs and outputs is manual, while call context identification and collection of performance metrics is automatic. Moreover, SCULPTOR provides feedback to the developer (detailed in Section III-B2) if it determines that the data provided is not sufficiently detailed to create an accurate model. Automatic identification of the proper granularity for the modules and the choice of module parameters can be done through instrumentation using LLVM (Clang) [3], [4] or ROSE [5] compiler infrastructure where major code blocks (e.g. function bodies and loops) can be instrumented for monitoring and local variables (e.g. function arguments, loop iterators or variables that are *def-use* [6] related to them) can be tracked. SIGHT's current use model enable developers to explicitly expose the information to be tracked and therefore provides more flexibility, specially for the cases where domain expertise might be required to identify important variables. However automating data-collection and instrumentation is orthogonal to our work which focuses on prediction errors of the models built using the collected data.

III. MODELING APPLICATION BEHAVIOR

In this section we describe our model building tool, SCULPTOR and how it statistically models application behavior.

A. Application properties

Accurate modeling and prediction of application behavior requires knowledge of the parameters that control it and metrics that quantify it.We execute a limited number of application training runs (e.g. during regression testing) and use our data collection framework (discussed in Section II) to collect the parameters and behavior metrics of each application module. We introduce two terms that will hold the center stage through the rest of the paper — **input features** and **observation features**. *Input features* are comprised of all collected parameters — the configuration properties, the command-line arguments, the input properties of a data structure in the application, such as matrix sparsity, graph diameter, or the loop iteration index. *Observation features* are comprised of all those features that quantify the behavior of the application. Application behavior is quantified in terms of various performance metrics (*e.g.*, execution time or hardware performance counter values), application-level quality metrics (*e.g.*, video frame rate, residual values in a linear algebra solver) and output values from a module (*e.g.*, number of search results).

Table I lists the applications we studied and a few of the input and observation features that SIGHT collects for each of them. CoMD (Classical Molecular Dynamics) [7] and LULESH(Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [8] are scientific simulations. FFmpeg [9] is widely used video processing software. SpMV [10] is a benchmark for sparse matrix vector multiplication which appears in scientific applications. LINPACK [11] solves a dense system of linear equations while PageRank [12] is graph search algorithm and used to compute a ranking of all the vertices. Black-Scholes [13] benchmark is a popular stock option pricing benchmark. Note that since our goal is statistical predictability rather than logical correctness, the features we selected are aggregate summaries of application state, whose values characterize application behavior. We do *not* try to model or predict the values of individual scalar values in application state, *e.g.*, values of individual elements of the output matrix in case of matrix multiplication).

B. Feature selection

The accuracy of a model depends on the granularity with which an application's behavior is modeled and the utility of the input features for predicting the observation features. At the time of instrumentation, it is not always clear to developers what granularity and choice of features is most useful. This makes it necessary to pre-process this information to make it maximally useful for subsequent analysis and to identify ways in which the feature set may be improved. Concretely, our pre-processing procedure (i) identifies the input features that are unlikely to be useful for predicting *any* observation feature so that they can be ignored by subsequent analysis and (ii) identifies the observation features that cannot be predicted so that the developer can be prompted to provide additional information as new input features. SCULPTOR accomplishes these by using a theoretical technique *Maximal Information Coefficient* (MIC). We first give a brief background on MIC and then explain how it is used by SCULPTOR.

1) Background: Maximal information coefficient analysis: The MIC [14] algorithm attempts to determine the strength of the relationship between a given pair of input and observation features, regardless of the actual shape of this relationship and works with both functional and non-functional relationships. MIC is based on the key intuition that if a relationship exists between two variables, then a grid can be drawn on the *scatter plot* of the two variables that partitions the data to encapsulate that relationship. Thus, to calculate the MIC of a set of two-variable data, the algorithm explores all grids up to a maximal grid resolution, dependent on the sample size, computing for every pair of integers (x, y), the largest possible mutual information achievable by any $x \times y$ grid applied to the data. Then it normalizes these mutual information values to ensure a fair comparison between grids of different dimensions and to obtain modified values between 0 and 1. A characteristic matrix is defined as $M = (m_{x,y})$, where $m_{x,y}$ is the highest normalized mutual information achieved by any $x \times y$ grid, and the statistic MIC to be the maximum value in M. MIC has been shown to find relationships that cannot be identified

Applications	Input features				Observation features
	Configuration parameters	Properties of input data	Runtime parameters	Internal variables	
LULESH	num of cycles to run, length of cube mesh, number of distinct regions		num processes	timestep loop iteration number	absolute diff, relative diff, perf measure
SpMV	memory-limit, time-limit	size, sparsity	block-size		mflop rates, perf measure
FFmpeg	crf,target resolution	size, bitrate,resolution			PSNR, output bitrate, perf measure
Black-Scholes	num runs	num options	num threads		avg error in predicted price, perf measure
LINPACK	dimension		num processes	num operations	perf measure
CoMD	num of unit cells, lattice parameter, time step		num processes	timestep loop iteration number	result quality, perf measure
PageRank	number of vertex	graph density			convergence diff, perf measure

TABLE I: Examples of input and observation features. Performance measures (perf measure) include, time, total number of instructions executed (TOT_INS), number of floating point instructions(FP_INS), number of load instructions (LD_INS), number of L2 data cache miss (L2_DCM), cache miss-rate (L2_DC_MR), MFLOPS etc. PSNR is peak signal-to-noise ratio.

by other methods such as Spearman's correlation coefficient or linear regression.

2) Using MIC as a filter: A key goal of our data collection mechanism is to minimize the developer's instrumentation effort. A key aspect of this is that developers are free to provide as many input features as they like, without worrying about whether they are actually correlated with each other or with the observation features. However, this means a filtering step is needed before the input features are considered in our model.

a) Identify duplicated features:: One issue that may occur is that multiple features actually represent a single piece of information. For example, in `LINPACK`, problem size `N` and internal variable `ops` (number of floating point calculations), capture the same information and are related by the equation: $ops = (2/3)N^3 + 2N^2$. SCULPTOR performs a MIC analysis between the input features to identify such duplicates (MIC-score (≥ 0.95)) and removes them.

b) Remove noise:: If an input feature is not related to *any* observation feature, including it as an input to a statistical model can add noise and reduce the model's accuracy. These are detected by checking if there exist any observation features for which their MIC is larger than a threshold and are removed if there are none. Conversely, there may be observation features that are not related to *any* input features. These are detected as above and are then communicated to the developer to encourage them to collect more expressive input features that might be useful for predicting these observation features. It might happen, even after few iterations of input feature refinement that some observation features are still not related to any input features. For example in FFmpeg, given an input bitrate, resolution, and a quality metric *Constant Rate Factor (CRF)* [15], we could not properly predict bitrate of the output video. This is because the output bitrate also depends on the actual content of input video (such as motion) and while maintaining same quality (CRF), FFmpeg applies more compression to a high motion video than to a slow video, leading to low output bitrate for the former. Since here we are not considering the actual content of the video, we can not properly predict output bitrate. This kind of observations features with low prediction accuracy are recorded and discarded from further analysis.

C. Choosing the best regression function

While MIC technique can identify useful input features, it cannot provide a model that predicts the observation features from these input features. SCULPTOR creates a predictive

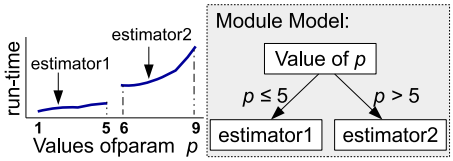


Fig. 4: Handling discontinuous behavior with incremental model update. Here run-time shows discontinuous behavior for values of input parameter p larger than 5. We add a new estimator to the model to predict that behavior.

model for each observation feature by applying a range of linear regression methods. In the experiments reported in Section VI we used Least Absolute Shrinkage and Selection Operator (`LASSO`) [16] with a polynomial fit (upto degree 3), which work well (in terms of accuracy and running time) for the applications we studied. Thus, a regression model is built for each output feature separately in terms of potentially all the input features. LASSO has the property that it tries to reduce the sum of coefficients of the terms, ultimately leading to a smaller number of terms. Our model, being of order three, has terms of the type $x_1, x_1^2, x_1^3, x_1x_2, x_1^2x_2, x_1x_2^2$ and being a linear regression model, it is a linear combination of such terms. [2]

SCULPTOR also verifies the quality of fit and avoids over-fitting using `k-fold` cross-validation (for our experiments we empirically chose $k = 3$ as we did not see any improvements in the accuracy of the model for $k > 3$). We record the observation features for which the prediction model could not achieve a sufficiently high R^2 fit score and do not use those as *detectors* for anomaly detection use-case because, such observation features might raise false alarms.

D. Incremental model update

In some scenarios, the application's performance behavior might be discontinuous. For example, there might be a big jump in run-time when the size of working set no-longer fits in L1-cache. Such discontinuous behaviors typically happen when the application hits some resource bottleneck, say, available cache, memory, network, I/O or memory bandwidth. Another cause of discontinuity might appear in the presence of input features that act as control variables for the program flow, leading to multiple behaviors of a module. When SCULPTOR

[2]It is to be noted, our overall technique is independent of the chosen regression model and a detailed analysis of the impact of the choice of regression model on the quality of anomaly detection would be an interesting future work.

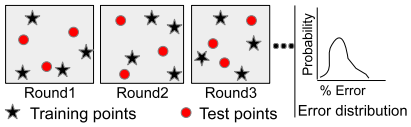


Fig. 5: Resampling train and test data points from training data space. Multiple such rounds are performed to obtain an error distribution for each predictor.

finds a discontinuous behavior in the observations, it adds a new estimator to model the next continuous section after the discontinuity. Thus, for each continuous segment of an observation behavior, with few discontinuities, SCULPTOR maintains separate estimators in a *decision-tree*-like data structure, as illustrated in Figure 4. Since we handle such discontinuities by incrementally updating our model to capture the behavior, users do not have to retrain the entire model.

IV. QUANTIFICATION OF PREDICTION ERRORS

Our modeling is based on a set of measurements of a set of application runs, denoted as *"available data"*, that cover some representative portion of the input feature space of its modules. When the application is executed in production, the input feature of its modules are denoted as *"production points"*. In an ideal scenario, available data could be gathered by sweeping through all possible points in the input feature space and collecting the corresponding observation features. While this would be ideal since every production point's observation features would be known, it is not practical since the input feature space is far too large for most real-world applications.

If the model created based on available data is not perfect (which is the case for most real applications), this situation leads to predictors that behave reasonably well when the production point is close to available data, but its prediction error grows as the production point moves further away from it. We call this the *"extrapolation error"*. Moreover, since prediction models generally cannot capture all the details of a dataset, there is some prediction error even for the production points within the available data set. We call this the *"interpolation error"*.

Quantification of such interpolation and extrapolation error is necessary for many use cases. For example, a job scheduler needs to consider the uncertainty in a job's predicted execution time when deciding whether to schedule it in a given time slice, since the job will be aborted if it runs over its allocation. In this paper, we focus on a use case of anomaly detection, where interpolation and extrapolation errors are used to calibrate the thresholds that determine whether a value of the given observation feature is sufficiently different from its predicted value to signal an alarm. Sections IV-A and IV-B present the error estimation techniques for interpolation and extrapolation error, respectively.

A. Distribution of prediction errors within the available data: Interpolation error

We characterize the interpolation error of our model by systematically evaluating the distribution of its prediction error within the available data set. Our error analysis engine, called

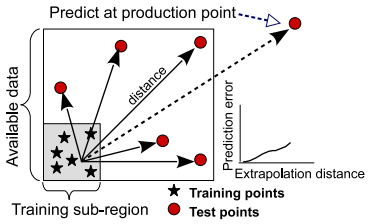


Fig. 6: Quantifying prediction error with extrapolation distance.

E-ANALYZER, uses *Bootstrapping* [17] to create multiple small training and test subsets of the available data set. Similar to Bootstrapping, members of each subset are sampled at random (with replacements), while ensuring each training set is disjoint from the corresponding test set. Then SCULPTOR trains a model on the training set and uses it to predict the observation features of the points in the test set, and records the distribution of these errors. Finally, the points are placed back into the set of available data and more samples are taken until the distribution of prediction error reaches statistical significance. The result is a histogram of model prediction errors. Since the raw counts of prediction errors are noisy, E-ANALYZER uses "Kernel Density Estimation" (KDE) to smooth out irrelevant variation. Finally, these smoothed histograms are normalized to add up to 1, so that they can be interpreted as the probability distributions of error magnitudes (note that individual values may be larger than 1). These distributions, provide an insight into the model's accuracy in the case where test data resembles training data. For example, a highly accurate model will have a narrow distribution around 0, while an inaccurate one will have high probability mass away from 0. Further, these distributions make it possible to calculate `Prob(observedValue | predictedValue)`, the probability that a particular observation feature value `observedValue` will be seen in a normal execution, given that the model predicted a value `predictedValue` for that observation feature at that production point. In Section V, we show how we use this information to achieve an anomaly detection scheme with low false positives. In Fig. 9, we show the empirically measured distribution of interpolation errors for a few interesting observation features from the applications that we studied.

B. Error as a function of distance: Extrapolation error

Models generated by SCULPTOR can predict the expected performance characteristics of the application with high accuracy, when production data points are close to the available data used for modeling. For production points which are far from the available data, the error in prediction typically increases. We want to quantify such extrapolation error as a function of the *distance* between the input feature points in the available data and those encountered during production runs, as illustrated by the graph in Figure 6 for a 2-dimensional input-space.

1) Distance measurement: Our approach for quantifying the dependence of model's error in predicting the output features of a given production point leverages the points in our available data set. First we look at the space of input

feature values of the points in the available data set and train the model on just the points in a corner of this space. We then observe how prediction errors grow as this model attempts to predict remaining points within the available data set as their distance to the model's training points grows.

This approach faces two major challenges. First, we must compute the distance from one input feature vector to another. The Euclidean or Mahalanobis distance formulas cannot be used here because the different dimensions of these vectors may span vastly different ranges of values. Further, for some input features, such as the *size* feature in PageRank [12] or the matrix size in LINPACK [11] there is no apriori bound on this range and therefore we can not normalize to make all features comparable in terms of numeric values.

The second issue is that we must find the region of the input feature space on which to train our model. The ideal location would be a corner in the space, as shown in Figure 6 for the example of a 2D space, so that we can select test points that move away from the training sub-region in *equal steps* along *all dimensions*. This approach does not work in a high-dimensional space, because the training data is too sparse in practice for there to be a large number of points in any corner.

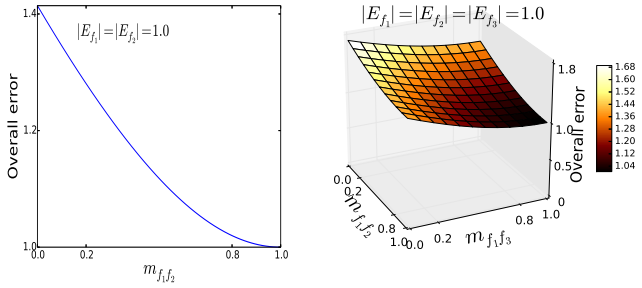
We propose an approach that addresses both of the above issues. The key idea is that although it is difficult to find a sufficient number of points in a corner of a high-dimensional space, this is made much easier if we cover the space by considering one dimension at a time. This way we can train a model on points in the input feature space that have small or large values in a given dimension and consider how prediction errors increase as the point we predict moves away along the same dimension. We can then combine these per-dimension error measurements to create the cumulative error for distance along all dimensions and compute prediction error at a particular production point (detailed in Section IV-C).

Algorithm-1 details our approach. For each input feature dimension, E-ANALYZER first *sorts* the data-points with respect to this feature. It then creates set S to contain the data-points that have similar values for the remaining features and selects the k points from S with the smallest values for this feature to create set S_{train}. It then trains a regression model on S_{train} and predicts the value of each observation feature for the remaining points in S. Finally, it records how the error in predicting each observation feature for each point p relates to its distance from S_{train} (errors for each combination of observation and input features are tracked separately). The distance is computed as $|\frac{f_p - \mu_{S_{train}}}{\sigma_{S_{train}}}|$, where f_p is the value of the current input feature dimension at point p and $\mu_{S_{train}}$ and $\sigma_{S_{train}}$ are the mean and standard deviation, respectively, of the value in this dimension of all points in S_{train}. The choice of k trades off between providing enough points to train an accurate model and leaving enough points to capture the scaling of extrapolation error. We found that k=50% worked well in our test cases.

After the pairs ⟨distance, prediction error⟩ are collected for each input feature and observation feature dimension, E-ANALYZER builds a simple polynomial regression estimator on this data. This makes it possible to predict how prediction error scales as distances grow beyond our set of available data.

Algorithm 1 Creating extrapolation error profile estimators

```
1:  F ← Set of features
2:  O ← Set of Observation features
3:  D ← All available data-points and corresponding observations
4:  procedure CREATEERRORPROFILE
5:      for all f in F do
6:          Sort D w.r.t f
7:          S ← Get data-points from D with similar values for {F − f}
8:          S_train ← First k data-points in S
9:          model ← CreateModel(S_train)
10:         for all pt in {S − S_train} do
11:             for all o in O do
12:                 d ← Distance(pt,S_train)
13:                 e ← PredictionError(pt,o,model)
14:                 errorProfile[o][f].addToList(d,e)
15:             end for
16:         end for
17:     end for
18:     for all f in F do
19:         for all o in O do
20:             errEst ← CreateEstimator(errorProfile[o][f])
21:         end for
22:     end for
23: end procedure
```



(a) Variations in $m_{f_1f_2}$ (b) Variations in $m_{f_1f_2}$ & $m_{f_1f_3}$

Fig. 7: Variation of overall error with variation in MIC between (a) 2 error components, (b) 3 error components

We maintain one such estimator for each combination of input and observation feature dimension.

C. Model error estimation

Given the input feature values for a production application run, we want to estimate the error in the model's prediction of each of its observation features. For a given observation feature, we want to do this by applying the error estimation model of each input feature dimension and combining the errors that all the single-dimension models predict. While it is possible to combine these error estimates via a simple aggregate such as an average or the Root Mean Square (RMS) norm, this estimate of overall error would be biased because some of the input features are correlated with each other. Giving equal weight to each of these correlated features would have the effect of weighting the data source behind this correlation with disproportionate importance. We thus designed a new technique to account for such correlations that gives the same weight to all sources of information that feed into the error estimates along all the individual input features. It works by removing weight from error predictions along individual input feature dimensions in proportion to how correlated they are to other dimensions that have already been accounted for. We start with the the RMS norm of the error predictions along all input features. Then we select one feature as the *base feature* and compute its correlation with all other

features using the *Maximal Information Coefficient* (MIC), which ranges between 0 and 1. Finally, we reduce the weight of the other features in proportion to the correlation with the base feature to account for the correlated contributions. Thus the *overall error* $\Psi(E_{f_1}, E_{f_2}, E_{f_3}, ...) =$

$$\sqrt{[E_{f_1}]^2 + [(1 - m_{f_1 f_2})E_{f_2}]^2 + [(1 - m_{f_1 f_3})E_{f_3}]^2 + ...} \quad (1)$$

Where f_1, f_2, f_3 etc. are the features while E_{f_1}, E_{f_2}, E_{f_3} etc. are the individual error components with respect to those features. We chose E_{f_1} as the *base error metric*. $m_{f_1 f_2}$ is the MIC of E_{f_2} w.r.t E_{f_1}. Similarly, $m_{f_1 f_3}$ is the MIC of E_{f_3} w.r.t E_{f_1} and so on.

Intuitively, in a correlated error scenario, while calculating contribution from E_{f_1} we have already accounted for some of the contributions (captured through MICs: $m_{f_1 f_2}$, $m_{f_1 f_3}$ etc.) from other components such as E_{f_2}, E_{f_3} etc. Thus, while calculating the overall error, we reduce the raw contributions of E_{f_2}, E_{f_3} by a factor of $1-m_{f_1 f_2}$ and $1-m_{f_1 f_3}$ respectively, to maintain the balance.

As can be seen, this formula gracefully satisfies two boundary conditions. When all the components are *independent* (i.e. $m_{f_1 f_k} = 0$), it reduces to the RMS norm. Further, when all the error components are *fully correlated* (i.e. $m_{f_1 f_k} = 1$ and E_{f_1} = E_{f_2} = E_{f_3}), we only consider error coming from the base feature *i.e.* E_{f_1}.

To provide a stronger intuition for how Equation 1 works, in Figure 7, we simulate how the overall error would behave for two and three error components, as we vary the degree of correlations between them. Figure 7 shows the overall error it computes across input features f_1, f_2 and f_3 (where we use 1.0 as the normalized prediction error along each component *i.e.* feature dimension), as the correlations (MIC) among f_1 and f_2, and f_2 and f_3 are varied between 0 and 1. The Figure shows that overall error computed by Equation 1 lies within the range $[1, \sqrt{2}]$ when considering just two features and within the range $[1, \sqrt{3}]$ when considering all three. In general, with normalized error along each dimension as 1.0, the predicted error rises to $\sqrt{k}$ for k features, when no features are correlated (identical to RMS norm). As the correlations between error components increase from 0 to 1, the total error approaches to 1.0.

It might be interesting to understand why overall error predicted by Equation 1 would grow with number of features contributing to the prediction error. The reason is, as we extrapolate along more input feature dimensions, we move further away (along all the dimensions) from the training region of the prediction model. Hence, it is only natural that our prediction error due to extrapolation would grow.

V. Anomaly detection as a use case

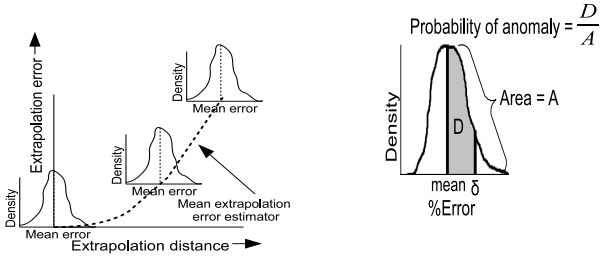
This section evaluates the utility of our error prediction technique by using it as a part of a probabilistic anomaly detector engine, GUARDIAN. By anomaly we mean, deviations of values seen for the observation features of an application module (whole application or a code region) relative to values that would be expected given the module's input features. Currently anomaly detection (*e.g.*, [18], [1], [19]) is done by creating a statistical model, predicting the behavior at the production point using the statistical model, and then using a predefined thresholds (*e.g.*, 50% deviation) for the difference between the predicted and the observed value. The current approach does not take into account the distance of input features from the training space. In contrast to this, GUARDIAN uses interpolation and extrapolation error analysis performed by E-ANALYZER (as discussed in Section IV) to calibrate the threshold based on the *distance* of production point from the training space. GUARDIAN only uses the input and observation features selected by SCULPTOR during modeling as discussed in Section III-B2.

Algorithm 2 Anomaly detection during production runs

```
P ← Production data-point (input features in production run)
Oset ← Set of selected Observation features
predictorObjects ← Dictionary of predictor objects for each observation
procedure ANOMALYDETECTIONPERMODULE
    for all obs in Oset do
        DOANOMALYDETECTION(obs)
    end for
end procedure
procedure DOANOMALYDETECTION(obs)
    actualVal = getObservedValue(obs)
    predVal = predictorObjects[obs].pred(P) //Calculate predicted value at P
    E_overall = calculateOverallExtrapolationError(P) //Expected error
    ObservedError = |actualVal − predVal| / predVal
    if ObservedError ≤ E_overall then
        δ = 0
    else
        δ = ObservedError − E_overall //Deviation from expected error
    end if
    prob = calculateProbability(δ)
    if prob > threshold then
        Flag anomaly error for obs
    end if
end procedure
```

The key idea of our approach, detailed in Algorithm 2 is as follows. During a production application run, our SIGHT tool observes the input and observation features of multiple modules. We use the models generated by SCULPTOR to predict for each observation feature the value $predVal$ that is expected to be observed given the module's input features, and compare it to the value $actualVal$ that is observed during the production run. If the percentage difference of $actualVal$ from $predVal$ is within the range of prediction error due to extrapolation ($E_{overall}$), we consider that value as *normal* because we know that this difference is within our predictor's error bounds. If $actualVal$ is outside this range of mean extrapolation error, we use the probability distribution of interpolation error to calculate the probability that $actualVal$ deviates from the $predVal$ because of the prediction error or because this production run was anomalous.

As before, let $f_1, f_2, ...$ be the set of pre-selected input features of a given module. Let $P(f_1 = x_1, f_2 = x_2, ...)$ denote the input features observed for a single execution of a module during a production run, where features hold values x_1, x_2, etc. respectively. Let o be an observation feature we wish to validate for anomalies. First, using our predictor models, GUARDIAN calculates the predicted value of o as: $predVal = pred(P)$, as discussed in Section III-C. Then we calculate the *distance* of P from the training sub-regions along the feature dimensions as discussed in Section IV-B1. Let d_{f_k} be the extrapolation distance of P along feature f_k. We



(a) Prediction error estimation (b) probability of anomaly

Fig. 8: **a)** Estimation of prediction error: Interpolation error distribution is put on top of estimated mean extrapolation error. **b)** Calculating *probability of anomaly* from distribution of errors, for a deviation of δ from expected extrapolation error.

calculate the error components E_{f_k} at P using extrapolation error estimators of each feature: $E_{f_k} = ErrEst_{f_k}(d_{f_k})$.

These components are then combined to calculate the overall extrapolation error $E_{overall}$ according to Equation 1. Let $actualVal$ be the value observed for observation o during the production run. To compute whether $actualVal$ is anomalous we first compute δ, the *deviation* of prediction error from the expected extrapolation error. If, $actualVal$ is within the range $predVal \pm (predVal \times E_{overall})$, then we consider $\delta = 0$. Otherwise, we calculate the deviation from expected extrapolation error as:
$\delta = |\frac{actualVal - predVal}{predVal}| - E_{overall}$ (Lines 10-17 in Algorithm 2)
Now, it is to be noted that, $E_{overall}$ is essentially the *mean* of extrapolation errors at that particular data point P. As discussed in Section IV-A, prediction errors due to interpolation error generally form a distribution around the mean value, as illustrated in Figure 8a. We assume that properties of this distribution remain the same irrespective of the distance of the production points from the training set, while extrapolation error represents how the mean value of the distribution changes with distance from the training set. Therefore, intuitively, we check whether it is unlikely that during normal execution we would see a larger deviation (of the observed error from the expected extrapolation error) than δ, and if so, signal an anomaly. Concretely, we calculate the *probability of an anomaly* given a δ deviation of observed error from its predicted error by calculating what is the probability that deviation of error under normal execution will be lower. If this calculated probability is lower than a threshold, then we consider the execution as normal, otherwise it is flagged as anomalous.

As illustrated in Figure 8b, GUARDIAN calculates the probability that we can experience a *higher* deviation(δ) from the expected extrapolation error under normal circumstances. If the area under the distribution curve, between the mean (*i.e.*, $E_{overall}$) and ∞, is A and the area under the curve between the mean and δ is D, then we calculate the probability of anomaly as: D/A.When this probability is high, we flag that observation as anomalous. Algorithm 2 summarizes these main steps which are followed during production run, for each code module in the application that has a prediction model.

VI. EVALUATION

A. Error analysis case studies

In this section we experimentally evaluate how effective our error prediction technique is at improving the effectiveness of the GUARDIAN tool for detecting anomalies in application behavior. Our evaluation focuses on the seven representative applications and benchmarks listed in Section III, selected from a wide variety of domains: CoMD, LULESH (scientific simulations), FFmpeg (video processing), SpMV, LINPACK (numerical processing), Black-Scholes (financial modeling) and PageRank (graph processing).

At first, we ran these applications under normal circumstances (*i.e.* without any injected errors) and measured the actual errors in prediction. Figure 9 focuses on the interpolation errors (predictions for points within the bounds of the training set), as described in Section IV-A. It includes two plots for each application each showing the probability distribution of interpolation errors for two representative combinations of module (code region that is measured) and observation feature that is predicted. For example, the top-left plot presents error in predicting the total number of instructions executed within calls to the `LagrangeElements` function in LULESH. The Y-axis of each plot is the density, as obtained from the Kernel Density Estimation[3], and the X-axis is the relative error in prediction. Most of the observations have a narrow distribution around the mean of zero error, which demonstrates that the modeling accuracy of SCULPTOR is high when data points fall within the bounds of training set. This accuracy is primarily controlled by the granularity at which application is tracked and the information content of the input features. For example, consider the CoMD plots in Figire 9, The left plot represents prediction error distribution for the entire `simulation`, which consists of setup-phase, timestep simulation loop and some epilogue. In contrast, the right plot represent the prediction errors for `AdvPos` function which is inside the timestep loop. The prediction error is notably lower for the latter, more focused, module because (i) its behavior is more homogeneous and (ii) additional parameters that control its behavior, such as the *loop iteration index*, are available when it executes, which enables more accurate prediction. The same pattern repeats for LULESH, PageRank and LINPACK, where the left plot corresponds to a larger application region than the right (in SpMV and FFmpeg we only instrumented the core function). Further, FFmpeg shows a related phenomenon where the bitrate of the output video has higher error than other predictions (left plot). This is because this bitrate depends on the actual content of the video, which we do not track as an input feature (recall our discussion in Section III-B2.

Figure 10 focuses on extrapolation error and shows representative examples E-ANALYZER's predictions of how the errors in predictions of various individual output features vary with increasing distance from the model's training set along various individual input features. Each plot focuses on a specific application, code region (module), the input feature, and the observation feature. We show two examples

[3]Y-axis is *not* the probability; area under the curve is the probability.

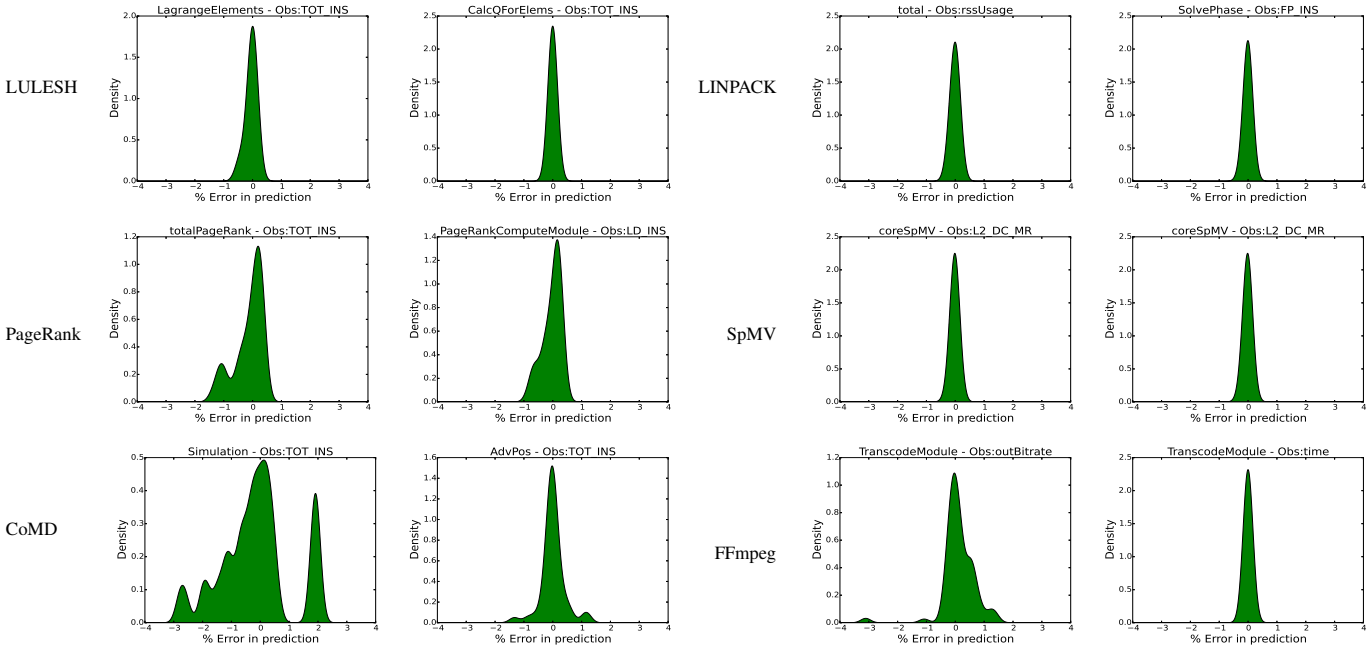


Fig. 9: Distribution of interpolation errors for applications

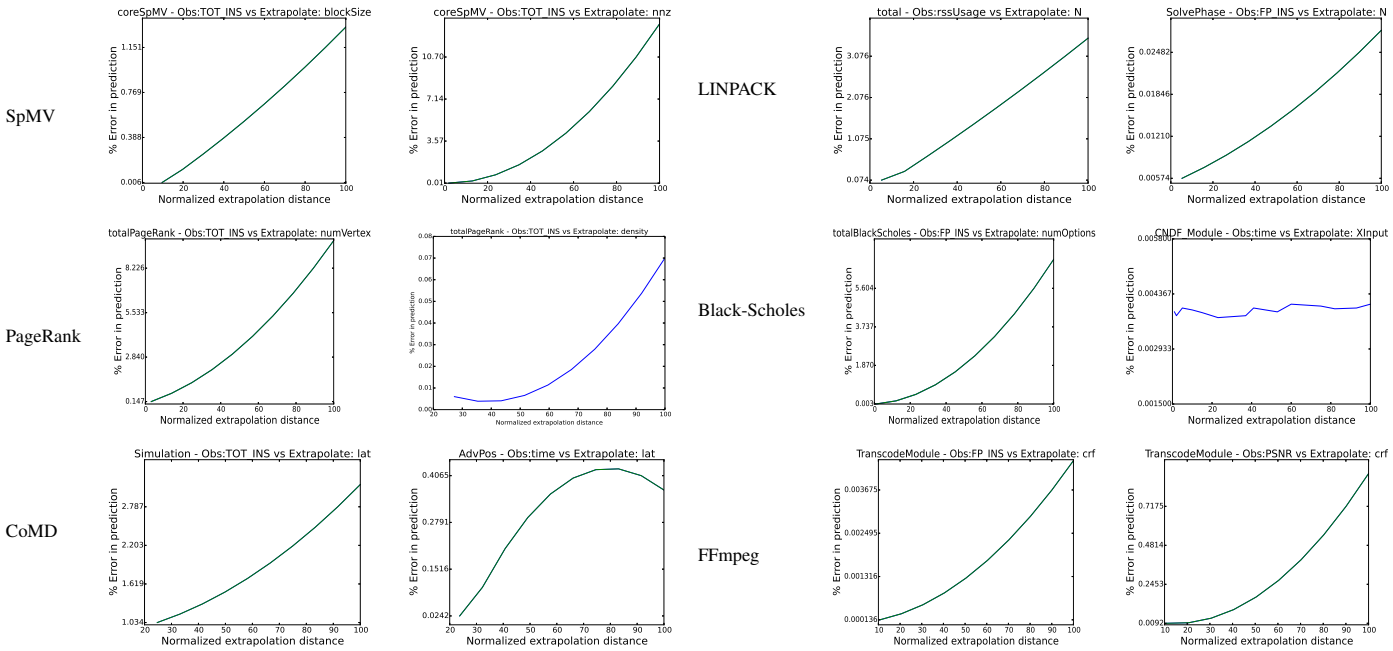


Fig. 10: Extrapolation errors for applications. "Extrapolate: X" in the titles mean, extrapolation is along input feature dimension *X*.
SpMV: *nnz* is the number of non-zero rows. LINPACK: *N* is matrix size. PageRank: *numVertex* is the number of vertex in the input graph, *density* is the density of the graph. Black-Scholes: *numOptions* is the number of stock options in input data, *XInput* is the input to CNDF function. CoMD: *lat* is lattice parameter. *nx* is the number of unit cells in x. FFmpeg: *crf* is constant rate factor, *inputSize* is size of the input video.

for each application. In each plot, the X-axis represents a normalized distance from the training region according to the measure we introduced in Section IV-B1, while the Y-axis is the prediction error. As expected, extrapolation error increases with distance from training set. Interestingly, it grows with different patterns and slopes in different cases. For example, for SpMV, the error in total instruction count relative to the block size is predicted to grow linearly, while the same metric grows quadratically relative to nnz (number of non-zeroes in the matrix). Further, even when patterns are linear, a variety of slopes are observed, for example, in the LINPACK, slopes of total floating point instructions and rssUsage (resident set size)

relative to the size of input matrix (N) are different. Further, in cases where we instrument code at a very fine granularity, such as a few lines of code that perform a very deterministic action, extrapolation error is independent of distance. This is the case for the *CNDF* module in Black-Scholes, which simply calculates a formula value for the cumulative normal distribution function. Another interesting example is the right-most plot of FFmpeg where E-ANALYZER achieves reasonable accuracy in predicting the peak signal-to-noise ratio, a video quality metric, based on the input feature CRF which controls the perceptible image quality, even though they are not directly related by any equations. Finally, an interesting phenomenon is observed in the right-most plot of CoMD, where the prediction error for execution time relative lat, decreases with distance for large distances. This is caused by the fact that a 2^{nd} degree polynomial was used to model this data, as discussed in Sec. IV-B, and a concave shape is an artifact of using this function. In the future work, we will explore the use of a wider range of function to model extrapolation errors, focusing on *isotonic* regression estimators.

It is to be noted that, the interpolation and extrapolation errors observed in Figures 9 and 10 can not be *fully* avoided even with a sophisticated estimator (such as ours) as these kinds of errors are inherent to any kind of modeling process.

B. *Evaluation of* GUARDIAN

This section presents a detailed experimental evaluation of GUARDIAN, when integrated with an input-aware modeling scheme and using threshold calibration based on our model error prediction technique. Our evaluation compares this variant of GUARDIAN to traditional options that use fixed thresholds to decide whether to signal an alarm. The fixed threshold based scheme goes through same modeling steps as done by SCULPTOR but for anomaly detection uses a predefined threshold T, where deviations of the actual value of an observation from the predicted one that are larger than T (actually $T\%$ of the predicted value) are flagged as anomalous. We vary values of T as 10%, 50%, 100% to control the detector's sensitivity level. Further, we evaluate two variants of GUARDIAN. In variant A only the mean of the error is extrapolated and any deviations larger than this mean are flagged as anomalous. Variant B includes the full functionality of GUARDIAN, illustrated in Figure 8a. Deviations of actual errors are compared to the extrapolated mean errors and for those that are larger, the algorithm signals an anomaly if the probability of observing such a large deviation is $< 10\%$.

1) False alarms: In this experiment we evaluate the false positive rate (*FP*) of the different tool variants presented above. Specifically, we evaluate how FP of GUARDIAN varies as the *distance* between available training data and the production point increases. We ran each application 15 times, with production points having feature values at *low distance* (distances between 5-10 from the training space), and *high distance* (distances between 30-35 from the training space). These distances are measured using the formula defined in Section IV-B1 and are multiples of the standard deviation of points within the training set, along each input feature. Since the applications were run under normal conditions, albeit with different parameter values, and the applications had no bugs, there should not be any false alarm.

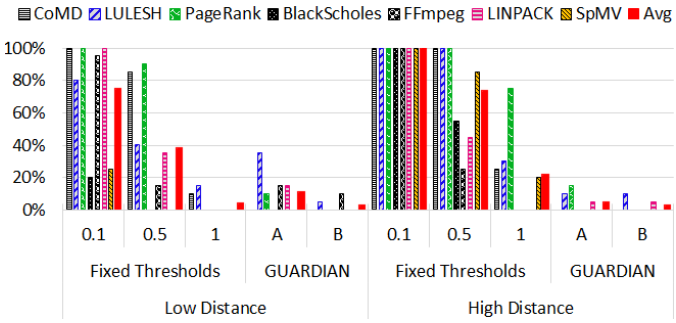


Fig. 11: False positive rates: GUARDIAN (GD) vs. Fixed threshold based detection. Variant A uses only the extrapolated mean prediction error while variant B is its full functionality. (Lower is better)

Figure 11, summarizes the results. On average (the right-most solid red bar), GUARDIAN variant A has 11% FP for production points at low distance and 4% FP for production points at high distance. GUARDIAN variant B, has only 2% FP which is significantly better than 74% and 38% average FP rates achieved by fixed threshold based detector with thresholds set at 10% and 50% of the predicted value, respectively. Threshold-based scheme with T=100% achieves comparable FP rates at low distance, but suffers from poor detection accuracy, as will be seen from the following experiment (Section VI-B2). For the production points at high distance, GUARDIAN shows a larger improvement relative to the threshold-based variants because it can widen the acceptable range of values with the help of extrapolation error prediction. Another important observation is that at higher distances, variants A and B perform almost equally well. The reason is, since in most of the cases, interpolation based distribution is very narrow, it becomes insignificant at higher distances, when the mean extrapolation error is large.

2) Accuracy: This experiment evaluates the detection accuracy of GUARDIAN in the presence of synthetically-injected performance faults. We implemented a fault injector using binary instrumentation tool Pin [20]. For each application, we inject 20 *bugs* of two categories:

- Comp: extra computing loops, and
- Mem: allocates and reads randomly and multiple times from a dummy memory array, which also creates cache contention.

We also vary the intensity of such bugs between *low* and *high*. We record how many times these injected bugs were flagged as anomaly and present the results for Comp in Figure 12 and for Mem in Figure 13. Binary instrumentation through Pin has significant monitoring overhead. To make sure, that our training runs and bug-injected runs experience similar overheads, even for training runs we injected dummy *no ops* using Pin. Since we saw in our earlier experiment that at high distances between the production and the training runs, the fixed threshold scheme is overwhelmed with false alerts, our accuracy experiments focus on low distances where the fixed

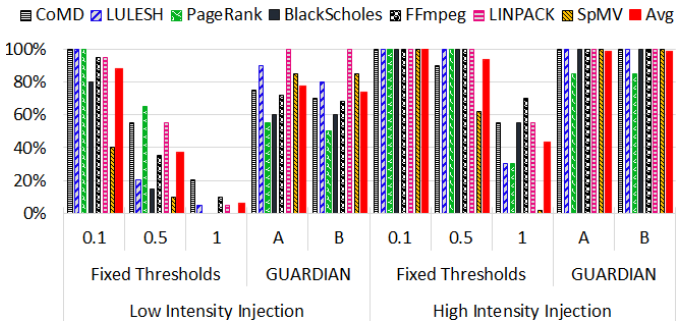


Fig. 12: Detection accuracy for extra computation bug (Comp): GUARDIAN vs. Fixed threshold-based detection. Variant A uses only the extrapolated mean prediction error while variant B is its full functionality. (Higher is better)

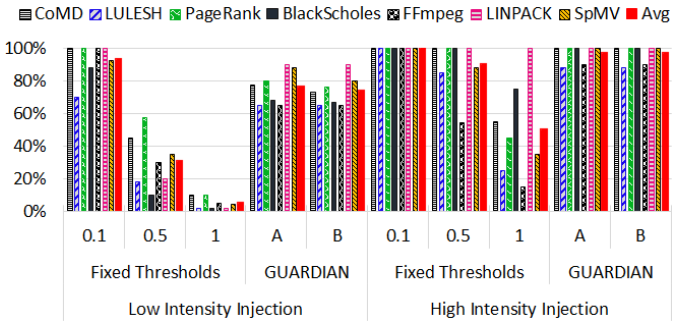


Fig. 13: Detection accuracy for extra memory read bug (Mem): GUARDIAN vs. Fixed threshold-based detection. Variant A uses only the extrapolated mean prediction error while variant B is its full functionality. (Higher is better)

threshold scheme may still be competitive with GUARDIAN. For GUARDIAN, we again use variants A and B, as discussed above and for fixed threshold-based technique, we vary the detection threshold between 10%, 50% and 100%.

We summarize the results of Comp injected faults in Figure 12. For the fixed threshold scheme, a threshold of T=10% has best detection accuracy. However, it is not useful in practice due to its high FP rates. On the other hand, T=100% is too insensitive and missed most of the low intensity faults. T=50% provides the best balance between the FP and detection rates and is thus a reasonable choice for comparison with GUARDIAN. Both the variants of GUARDIAN perform almost equally well and most of the time perform better than the fixed threshold scheme with T=50%, with one exception, PageRank. Our hypothesis is that in this case the intensity of the injected bug was too low compared to the normal variation of PageRank's behavior, which resulted in low accuracy for GUARDIAN. At the same time accuracy of the fixed threshold detector looks good due to its inherent high FP rate (Figure 11). Another observation is that variant A of GUARDIAN is slightly better than variant B because, it considers mean extrapolation error as a hard threshold and therefore is more sensitive to anomalies. For the same reason, this tends to have higher FP rates.

The accuracy results for Mem injected faults is shown in Figure 13 and follows the same trend. GUARDIAN variants A and B perform significantly better than fixed threshold based detection with T=50%. We noticed that even though for Mem faults we injected repeated memory read operations, the majority of the time it was detected either as increased number in total instruction count or as increased L2 cache miss or through increased number of load instructions, but not by increased resident set size.

VII. RELATED WORK

Extrapolation: In the machine learning community, it is well-known that quality of modeling is only as good as the training data. An interesting work [21] related to k-fold cross validation classifier provides a theoretical foundation of estimation errors but they do not cover extrapolation. In the context of pattern discovery, a recent work by Wilson *et al.* [22] used expressive closed-form kernel-based approach for Gaussian process extrapolation but did not handle multi-dimensional inputs. An improved approach for kernel learning for multi-dimensional pattern extrapolation was presented in [23].

Performance prediction: Another body of work focuses on performance forecasting [24], [25] which addresses the issue of predicting the performance of the application as a whole. They do not try to model the application at the granularity of code regions. More recent advanced variations of these techniques [26], [27] also consider the size of application inputs and try to predict how the resource usage would scale for bigger input sizes. These works do not consider how prediction error would grow in the presence of extrapolation. Moreover, for our modeling, we use not only the size of inputs but also more interesting features such as matrix sparsity and graph density. An interesting work by Jiang *et al.* [19] shows that some variables are an early indicator of how other variables will vary later on in the program and can therefore be used for early prediction of later program behavior. Specifically, in the context of dynamic program optimization, they argue, correlation between trip counts of loops as one such key indicator. Some related works focused on predicting performance under interference in multi-tenant environments [28], [29], [30], [31]. Even though these papers use variety of prediction techniques involving system metrics, they do not consider inherent prediction errors.

Anomaly detection and diagnosis: Performance problems have always been a source of frustration for system administrators and software engineers. There are many papers devoted to techniques for diagnosing such problems [32], [33], [34], [18], [35], some of which specifically looks at correlation between performance metrics. These tools do not track program properties such as values of input and internal variables and cannot operate in a production environment far away from the training environment.

Probably two works which are closest two our approach are Daikon [36] and DIDUCE [37]. Both focus on automatic invariant detection. DIDUCE is more advanced as it can also have an anomaly detection engine based on those inferred invariants. Both Daikon and DIDUCE involve some training runs of the correct application to identify invariants, an approach similar to ours. Since they do not consider input-aware model creation, strict invariants identified by these tools can

raise many false alarms in production when they encounter completely new set of values. We consider inaccuracy in predictor models and use input data to quantify such errors and dynamically calibrate our thresholds for detection. Thus GUARDIAN is more resilient to prediction errors with unseen datasets.

VIII. CONCLUSION

Techniques to predict the performance and functional behavior of applications are important in the design of many tools. When these tools take into account application input and configuration parameters they are able to achieve high levels of predictive accuracy but face the challenge that they can only be trained on a small fraction of the overall space of inputs and configurations. Since this means that in most production runs the models will need to make predictions about configurations that are well outside the space on which they were trained, it is necessary for such models to quantify their prediction errors across the entirety of this space. In this paper, we present a systematic approach to quantify such prediction errors and demonstrate the utility of our approach via a practical use case of an anomaly detector. This detector calibrates its alarm thresholds based on the error estimates provided by our technique. Our experimental evaluations confirm that this tool achieves a low false positive rate while maintaining a high detection accuracy compared to a fixed threshold based anomaly detector that do not use our prediction error characterization based technique.

IX. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their invaluable feedback. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1337158 and CNS-1405906 and Department of Energy/Lawrence Livermore Contract No. B610962. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the project sponsors.

REFERENCES

[1] G. Bronevetsky, I. Laguna, B. R. de Supinski, and S. Bagchi, "Automatic fault characterization via abnormality-enhanced classification," in *DSN*, 2012.
[2] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ACM SIGPLAN Notices*, vol. 41, no. 11. ACM, 2006, pp. 185–194.
[3] "Clang fronend for LLVM http://clang.llvm.org/."
[4] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.
[5] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, pp. 215–226, 2000.
[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pp. 451–490, 1991.
[7] "CoMD http://www.exmatex.org/comd.html."
[8] "LULESH https://codesign.llnl.gov/lulesh.php."
[9] "FFmpeg https://www.ffmpeg.org/."
[10] "SpMV Benchmark http://bebop.cs.berkeley.edu/spmvbench/."
[11] "LINPACK Benchmark http://www.netlib.org/benchmark/hpl/."
[12] A. N. Langville and C. D. Meyer, "Deeper inside pagerank," *Internet Mathematics*, 2004.
[13] "PARSEC: Black-Scholes http://parsec.cs.princeton.edu/."
[14] D. N. Reshef, Y. A. Reshef, H. K. Finucane, S. R. Grossman, G. McVean, P. J. Turnbaugh, E. S. Lander, M. Mitzenmacher, and P. C. Sabeti, "Detecting novel associations in large data sets," *Science*, pp. 1518–1524, 2011.
[15] "FFmpeg and H.264 Encoding Guide," https://trac.ffmpeg.org/wiki/Encode/H.264.
[16] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.
[17] C.-F. J. Wu, "Jackknife, bootstrap and other resampling methods in regression analysis," *Annals of Statistics*, pp. 1261–1295, 1986.
[18] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *ICDCS*, 2009.
[19] Y. Jiang, E. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao, "Exploiting statistical correlations for proactive prediction of program behaviors," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2010.
[20] C.-K. Luk, R. Cohn, and et al., "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[21] J. D. Rodriguez, A. Perez, and J. A. Lozano, "Sensitivity analysis of k-fold cross validation in prediction error estimation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, pp. 569–575, 2010.
[22] A. G. Wilson and R. P. Adams, "Gaussian process kernels for pattern discovery and extrapolation," in *ICML*, 2013.
[23] A. Wilson, E. Gilboa, J. P. Cunningham, and A. Nehorai, "Fast kernel learning for multidimensional pattern extrapolation," in *NIPS*, 2014.
[24] C. Stewart, T. Kelly, and A. Zhang, "Exploiting nonstationarity for performance prediction," in *ACM SIGOPS Operating Systems Review*, 2007.
[25] W. Pfeiffer and N. J. Wright, "Modeling and predicting application performance on parallel computers using hpc challenge benchmarks," in *IPDPS*, 2008.
[26] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," in *PLDI*, 2012.
[27] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," in *PLDI*, 2012.
[28] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *MICRO*, 2011.
[29] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ASPLOS*, 2013.
[30] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma, "Mitigating interference in cloud services by middleware reconfiguration," in *Middleware*, 2014.
[31] A. K. Maji, S. Mitra, and S. Bagchi, "Ice: An integrated configuration engine for interference mitigation in cloud services," in *ICAC*, 2015.
[32] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software." in *OSDI*, 2012.
[33] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: Automated classification of performance crises," in *EuroSys*, 2010.
[34] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin, "Accurate application progress analysis for large-scale parallel debugging," in *PLDI*, 2014.
[35] I. Laguna, S. Mitra, F. A. Arshad, N. Theera-Ampornpunt, Z. Zhu, S. Bagchi, S. P. Midkiff, M. Kistler, and A. Gheith, "Automatic problem localization via multi-dimensional metric profiling," in *SRDS*, 2013.
[36] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.
[37] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *ICSE*, 2002.