Adaptive System Anomaly Prediction for Large-Scale Hosting Infrastructures

Yongmin Tan, Xiaohui Gu
North Carolina State University
Raleigh, NC 27695
ytan2@ncsu.edu,gu@csc.ncsu.edu

Haixun Wang
Microsoft Research Asia
Beijing, China 100190
haixunw@microsoft.com

ABSTRACT

Large-scale hosting infrastructures require automatic system anomaly management to achieve continuous system operation. In this paper, we present a novel adaptive runtime anomaly prediction system, called ALERT, to achieve robust hosting infrastructures. In contrast to traditional anomaly detection schemes, ALERT aims at raising *advance* anomaly alerts to achieve just-in-time anomaly prevention. We propose a novel context-aware anomaly prediction scheme to improve prediction accuracy in dynamic hosting infrastructures. We have implemented the ALERT system and deployed it on several production hosting infrastructures such as IBM System S stream processing cluster and PlanetLab. Our experiments show that ALERT can achieve high prediction accuracy for a range of system anomalies and impose low overhead to the hosting infrastructure.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Reliability, availability, and serviceability

General Terms

Reliability, Management, Experimentation

Keywords

Anomaly Prediction, Context-aware Prediction Model

1. INTRODUCTION

Large-scale hosting infrastructures have become important platforms for many real-world systems such as cloud computing [1], massive data analytics [6, 7], and enterprise data centers. Many real-world applications such as stream processing require 24x7 continuous system operation. Unfortunately, today's large-scale hosting infrastructures are still vulnerable to various system anomalies such as performance bottlenecks, resource hotspots, and service level objective (SLO) violations. System administrators are often overwhelmed by the tasks of correcting anomaly problems under time pressure. Thus, it is imperative to achieve automatic system anomaly management to provide robust large-scale hosting infrastructures.

Previous distributed system anomaly management work (e.g., [12, 16, 28, 39]) can be broadly classified into two categories: (1) *reactive* approaches that take corrective actions after an anomaly happens, and (2) *proactive* approaches that take preventive actions (e.g., backup) on *all* system components beforehand. The reactive approach can have prolonged service downtime, which is often unacceptable by continuously running applications such as stream processing. In contrast, the proactive approach offers better system reliability but can incur prohibitive overhead. This motivates us to explore a new *predictive* anomaly management approach [23, 24] that can raise *advance* anomaly alerts to trigger proper anomaly correction in a just-in-time fashion.

To achieve efficient predictive anomaly management, one big challenge is to provide high quality online anomaly prediction. Although previous work (e.g., [11, 15, 20]) has addressed the anomaly detection problem, anomaly prediction needs to capture pre-anomaly symptoms to raise *advance* anomaly alert before the anomaly happens. Second, applications running in the hosting infrastructure are often opaque to the infrastructure provider, which demand black-box approaches to anomaly prediction. Third, a large-scale hosting infrastructure often consists of thousands of hosts and many more software components. To make runtime anomaly prediction practical for large-scale hosting infrastructures, we must employ light-weight learning methods.

More importantly, many real world applications (e.g., data stream processing, Google MapReduce [18]) running in hosting infrastructures are long lived and operate under changing execution contexts such as time-varying input workload and fluctuating resource availability. Applications may exhibit context-dependent behavior. For example, given an input workload of 100 tuples per second and 40% CPU, a normal stream operator can achieve a throughput of 50 tuples per second while a faulty stream operator with a memory leak bug under the same input workload and CPU allocation can only achieve a throughput of 5 tuples per second. However, if the CPU allocation is reduced to 5%, the normal stream operator can only achieve a throughput of 5 tuples per second. Thus, without considering the execution context, we cannot distinguish a faulty component with memory leak from a normal component under low CPU allocation. Thus, the runtime anomaly prediction system must be adaptive in order to achieve high quality prediction for dynamic hosting infrastructures.

In this paper, we present the design and evaluation of a novel adaptive runtime anomaly prediction system called *ALERT*. We focus on anomaly prediction that can raise advance alert before an anomaly happens. ALERT depends on an anomaly detection system (e.g., [11, 17]) to provide normal and anomaly state labels for different measurement samples. However, to achieve prediction, ALERT employs triple-state multi-variant stream classifica-

PODC'10, July 25–28, 2010, Zurich, Switzerland.

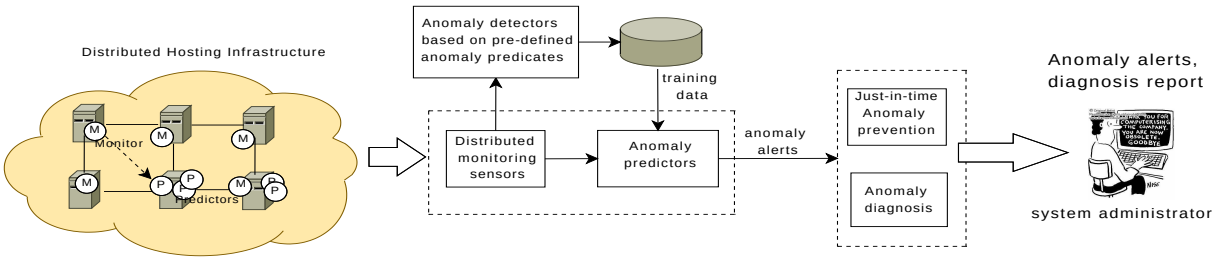


Figure 1: Predictive anomaly management for distributed hosting infrastructures.

tion scheme [23] to capture a special alert state in addition to the normal and anomaly state. The alert state corresponds to a set of measurements *preceding* the anomaly state, which allows the prediction model to capture pre-anomaly symptoms. Thus, the prediction model can raise advance alert when the monitored component enters the alert state rather than wait until the component is already in the anomaly state. We can adjust the scope of the alert state to tune the sensitivity of the prediction model (i.e., tradeoff between the accurate predictions and false alarms).

To adapt to dynamic execution environments, one simple approach is to continuously update the prediction model with new training data. However, this simple incremental approach has two fundamental problems. First, the anomaly prediction system may incur large overhead to the monitored infrastructure due to frequent model re-training. Second, the accuracy of such prediction model may be low when the execution context fluctuates a lot (e.g., alternating between high and low input workloads or high and low resource availability). Third, the execution contexts are unknown a priori and exhibit evolving behavior. To address the challenge, ALERT employs self-evolving, context-aware prediction models. Under a specific execution context, the prediction model gives consistent state labels for the same measurement. Our scheme first employs a clustering scheme to automatically discover different execution contexts. We then train a set of prediction models, each of which captures anomaly behavior under a specific execution context. During runtime, ALERT dynamically switches between different prediction models based on context evolving patterns to achieve high quality anomaly prediction for dynamic systems. Our approach differs from previous model ensemble approaches (e.g., [45]) in several aspects. First, unlike previous ensemble approach, wherein each classifier is learned from a fixed window of data that might span multiple execution contexts, in our approach, we cluster data that belongs to one context and learn a model from the conflict-free data. Second, we establish explicit mapping from prediction models to different execution contexts, which can improve prediction accuracy as well as avoid repetitive learning.

We have implemented a prototype of the ALERT system. To make the ALERT system practical for large-scale hosting infrastructures, we employ fully decentralized monitoring, learning, and prediction architectures, which is illustrated by Figure 1. We have tested the ALERT system on IBM System S stream processing cluster [21,22] and PlanetLab wide-area network system testbed [5]. Our experimental results using real system performance anomalies and host failures show that: 1) a range of system anomalies indeed exhibit predictability; 2) ALERT achieves much higher prediction accuracy than exiting alternative algorithms (e.g. 50% higher true positive rate and 80% lower false alarm rate); and 3) ALERT imposes low overhead for large-scale hosting infrastructures with real-time prediction performance (e.g., a few milliseconds training time and a few microseconds prediction time).

The rest of the paper is organized as follows. Section 2 presents the design details of our approach. Section 3 presents the prototype implementation and experimental results. Section 4 compares our work with related work. Finally, the paper concludes in Section 5.

2. SYSTEM DESIGN

In this section, we present the design details of the ALERT system. We first present the basic anomaly prediction model. Then, we describe the context discovery scheme. Third, we describe our adaptive anomaly prediction algorithm.

2.1 Baseline Anomaly Prediction Model

To perform runtime system anomaly prediction, we deploy monitoring sensors on all hosts in the hosting infrastructure, which continuously monitor a set of metrics $\{x_1, ..., x_k\}$, such as CPU consumption, memory usage, input/output data rate, buffer queue length, for each running host and application component. For example, we collect about 20 metrics on each host in IBM System S [23, 24] and about 66 metrics on each host in PlanetLab [4]. The monitoring sensor periodically samples each metric value at a certain rate (e.g., one sample every 10 seconds) to form a *measurement stream*. To achieve online anomaly prediction, we employ a triple-state stream classifier that can continuously classify each measurement sample into normal, alert, or anomaly state. The prediction model will raise alert when the component enters the alert state that precedes the anomaly state.

To train the prediction model, we first employ an anomaly detection module to label all measurements with either normal or anomaly states. A simple anomaly detection module can use anomaly predicates [20] based on the user's service level objective (SLO) requirements. For example, we can use an anomaly predicate "*processing time* $>$ *50ms*" to check whether a system is in an anomaly state in terms of performance. Previous work also provided more advanced anomaly detection schemes that can accurately distinguish anomaly from application change [15] and infer anomaly labels using similarity clustering [17]. Note that the focus of our work is on anomaly prediction rather than anomaly detection. For prediction, we introduce a special *alert* state to capture pre-anomaly symptoms. In the feature space, the alert state corresponds to a region "preceding" the anomaly points, illustrated by Figure 2.

Different from the normal and abnormal measurements that are labeled by the anomaly detector, whether a set of measurements are labeled as alert is controlled by an *alert interval* (I), which denotes a time interval (e.g., 30 seconds) before the anomaly incident. Suppose the anomaly incident happens at time t, all the measurement points sampled between $t - I$ and t will be labeled as alert. For example, we use a larger alert interval ($I = 3$) in Figure 2(a) than that ($I = 1$) in Figure 2(b). Thus, the alert regions in Figure 2(a) include more measurement points than those in Figure 2(b). As a result, the prediction models in Figure 2(a) will classify more measurement samples as alert state than those

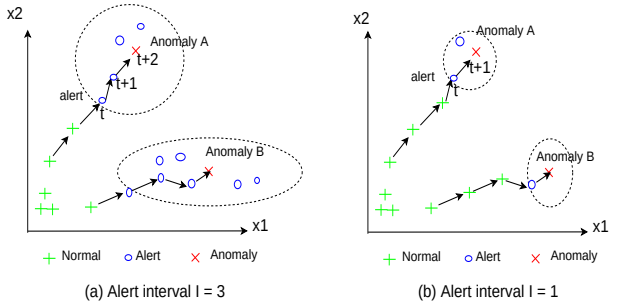


Figure 2: Tunable anomaly predictor with different alert intervals.

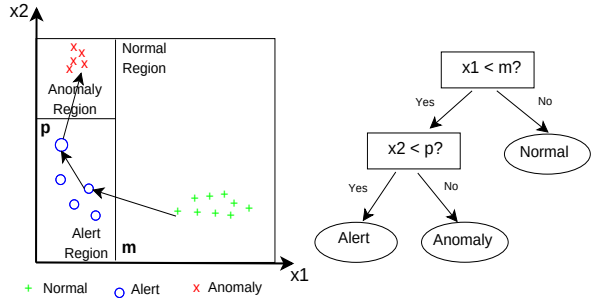


Figure 3: Anomaly prediction using triple-state decision tree classifier.

in Figure 2(b). Intuitively, the larger the alert interval, the more measurement points will be classified as alert, the more likely the predictor is to raise anomaly alerts. Thus, we can use the alert interval I as a tuning knob to control the predictor's true positive rate A_T and false-alarm rate A_F. The anomaly predictor is said to raise a correct alert if the predicted anomaly indeed happens shortly after the anomaly alert is issued. Let N_{tp}, N_{fn}, N_{fp}, and N_{tn} denote the true positive number, false negative number, false positive number, and true negative number, respectively. The true positive rate A_T and false alarm rate A_F of the anomaly predictor are defined in a standard way as

$$A_T = \frac{N_{\text{tp}}}{N_{\text{tp}} + N_{\text{fn}}}, A_F = \frac{N_{\text{fp}}}{N_{\text{fp}} + N_{\text{tn}}} \quad (1)$$

At one extreme, if we set $I = 0$, ALERT becomes conventional reactive approach where the alert state is always empty and no alarm will be generated before anomaly happens (i.e., $A_T = 0$, $A_F = 0$). At the other extreme, if we set $I = \infty$, ALERT becomes traditional proactive approach that performs preventive actions on all components unconditionally (i.e., $A_T = 1$, $A_F = 1$). The optimal solution often lies in-between the two extremes in practice, which motivates us to develop tunable prediction models. As we will show later in our experimental results, the alert interval can indeed be used to tune the tradeoff between true positive rate and false alarm rate.

We chose decision trees [27] to classify component states in this paper since the decision tree classifier can produce rules with direct, intuitive interpretation by non-experts. Thus, the predictor can not only raise anomaly alert but also provide cues for possible anomaly causes[1]. Figure 3 illustrates a simple case of classification using two metrics. For state classification, the decision tree essentially applies a sequence of threshold tests on the metrics. The predicate that corresponds to the alert region is "$x_1 < m$ and $x_2 < p$", and can be determined by following the path in the tree which leads to the leaf labeled *alert*. The decision tree classifier is trained using labeled measurement data from all three states. In order to effectively and automatically discover the appropriate features from the monitored metrics for prediction, the classifier has to incorporate multiple features in the training phase. An additional benefit of decision trees is that they can inherently select those metrics appropriate for state classification by seeking the shortest possible tree that explains the data. In our case, the feature selection occurs whenever we induce a new decision tree classifier under a specific execution context. We train every new classifier by incorporating all monitoring metrics, and rely on the decision tree classifier training algorithm to perform feature selection.

[1]However, our adaptive anomaly prediction scheme is not restricted to the decision tree classification method, which can be applied to other classification methods as well.

2.2 Context-Aware Anomaly Prediction Model

We now present the context-aware anomaly prediction model training algorithm, illustrated by Figure 4. We first employ a clustering algorithm to discover different execution contexts in dynamic systems. We then train a set of prediction models described in Section 2.1, each of which is responsible for predicting anomalies under a specific context. In contrast to common online learning algorithm that frequently updates models with new training data, our scheme induces a set of prediction models over a long period of training data to avoid unnecessary repetitive learning. More importantly, we induce models from conflict-free data, which produces high quality prediction models.

Our approach is based on two observations. First, once a system component enters a certain execution context, it will stay in the context for a period of time, until a certain event occurs which leads the system into another context. Second, the system may operate under similar execution contexts repetitively over a long period of time. For example, a web server often receives higher workload in the morning and lower workload in the evening and such execution context switching repeats. In Figure 4, we show a stream of measurement samples (divided into stream data blocks). Each stream data block d_i contains a small fixed length of measurement samples. Data blocks with the same color belong to the same execution context. Here, the system component operates under three different execution contexts. Context C_1, for instance, has three (non-contiguous) occurrences at time 0, 8, and 18. The system component switches from Context C_1 to C_2 at time $t = 6$.

To achieve context-aware anomaly prediction, one big challenge is that system contexts are hard to identify and characterize as they evolve over time. Our goal is to group measurement samples that correspond to the same context, and then learn models or classifiers from the grouped measurements. Grouping is essential because each individual instance or occurrence of the context often contains too little information to fully characterize the context. A classifier trained from insufficient data will have large overfitting error and will generalize badly for future testing data. On the other hand, by grouping together measurement samples belonging to the same context, we will be able to learn high quality prediction models as we minimize overfitting error. We employ a clustering algorithm to discover different contexts. The goal of the algorithm is to partition a measurement stream into a set of stream segments, , denoted by D_i. Each stream segment contains several continuous data blocks:

$$P = \{D_1, D_2, \cdots, D_k\} \quad (2)$$

that minimizes the global error:

$$E(P) = \frac{1}{|D|} \sum_{D_i \in P} \frac{1}{|D_i|} Err(D_i) \quad (3)$$

where $Err(D_i)$ denotes the error of the predictor learned from

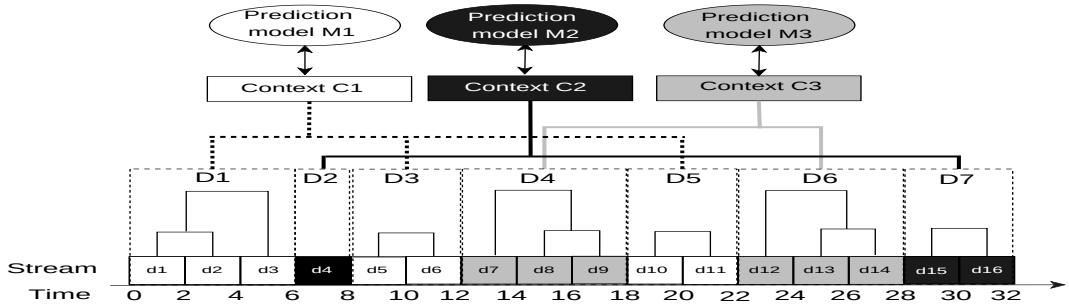


Figure 4: Context-aware anomaly prediction.

stream segment D_i. Before we discuss how to compute $Err(D_i)$, let us first understand the rationale behind the optimization problem. It is easy to see that when $E(P)$ is minimized, the quality of the contexts, or the accuracy of the predictor learned from the contexts, is maximized. The reason is the following. If two segments belong to the same context, then merging them will reduce the overfitting error of the predictor learned from the merged segments. On the other hand, if two segments belong to different contexts, then merging them will create a training data set with conflicting contexts, which reduces the accuracy of the predictor trained from such data. Thus, minimizing $E(P)$ maximizes the quality of the discovered contexts.

To compute $Err(D_i)$, the error of the predictor trained from segment D_i, we perform cross validation. For instance, two-fold cross validation randomly splits data in a segment into two sets. We use data in the first set to train a predictor, and use the other set to compute $Err(D_i)$, which is the percentage of wrong predictions made by the trained predictor. It is easy to see that in order for cross-validation to work, segments must satisfy the constraint $|D_i| > 1$ for all i to avoid the trivial partition wherein each D_i contains only one data sample.

The core task is then to minimize $E(P)$ in Equation 3, which gives us the optimal partition of the stream data and hence the contexts we want to discover. $E(P)$ can be minimized by dynamic programming since the clustering problem we defined above has optimal sub-structures. Let $D_{1,n}$ denotes a stream of length n, and $D_{i,j}$ denote a substream from data block i to j. Let $P_{i,j}$ denote the optimal partition of $D_{i,j}$, that is, predictors trained from segments defined by $P_{i,j}$ have the minimum validation error $E(P_{i,j})$. Our task is to find $P_{1,n}$, the optimal partition for the entire stream.

Assume we know $P_{1,k}$ and $P_{k+1,n}$, the optimal partitions for substream $D_{1,k}$ and $D_{k+1,n}$, $\forall k, 1 \leq k < n$. Consider $P_{1,n}$, there are two cases: it either contains no sub-partition, i.e., the entire stream $D_{1,n}$ forms a single segment, or it is a union of two partitions $P_{1,k}$ and $P_{k+1,n}$, for a certain k. In the first case, we learn a predictor from $D_{1,n}$, and estimate the validation error of the predictor. In the second case, the validation error can be derived by:

$$E(P_{1,n}) = \min_k \{\frac{k}{n} E(P_{1,k}) + \frac{n-k}{n} E(P_{k+1,n})\} \qquad (4)$$

Clearly, finding the best partition for $D_{1,k}$ or $D_{k+1,n}$ is the sub-problem with the same structure. We solve these sub-problems recursively until we reach the simple case $i = j$ when we partition $D_{i,j}$, $1 \leq i \leq j \leq n$. Note that we only need to solve each sub-problem once, as we memorize and reuse solutions to problems that we have already solved. The intermediate results $E(P_{i,j})$ are stored and reused by succeeding computation. By storing back-tracking pointers, the corresponding optimal partition $P_{1,n}$ can be recovered easily.

Assume that we have decided the optimal partition with N stream segments, i.e. $P_{1,n} = \{D_1, D_2, \cdots, D_N\}$. Each segment in $P_{1,n}$ represents a context occurrence. The boundary between two segments means a context change. One context usually has multiple occurrences throughout the entire measurement stream. Therefore, we need to perform a clustering algorithm again to group the segments which maybe non-contiguous into a number of unique contexts. For example, in Figure 4, stream segments D_1, D_3 and D_5 are merged as a specific context C_1. Contexts C_2 and C_3 are formed in a similar way. We still minimize the objective function in Equation 3 to decide the optimal partition. The only difference is that now any two stream segments (i.e., contiguous or non-continuous) are eligible to be merged.

In terms of complexity, when we decide the optimal partition for continuous data blocks, each sub-problem $E(P_{i,j})$ requires us to train and test a classifier to get the prediction error value with no sub-partition. Note that we use decision tree classifier and adopt C4.5 implementation [2]. Assume there are m measurement samples in $D_{i,j}$ and k features for each sample. Previous work has proven that the total cost of building a decision tree classifier is $O(km \log m) + O(m(\log m)^2)$ [43]. There are totally $\Theta(n^2)$ sub-problems. After that, when we decide the optimal partition for N stream segments, a naive solution requires us to examine $\Theta(N^2)$ possible segment pairs (D_i,D_j). Each examination incurs cost of building a decision tree classifier to get the prediction error for the temporarily combined segment $D_i \cup D_j$. Clearly, the overall complexity is high. However, heuristic optimization techniques such as greedy top-down or bottom-up techniques can be developed to find good but suboptimal solutions. In our implementation, we use a two-stage hierarchical clustering algorithm [14] which belongs to the greedy bottom-up technique. We will show later that this lightweight heuristic algorithm has fast training time and still achieves good performance.

After extracting different execution contexts, we train a set of prediction models $\mathcal{M}_1, ..., \mathcal{M}_q$, where $\mathcal{M}_k, 1 \leq k \leq q$ is a unique model for context C_k. In contrast to a monolithic prediction model that is trained using all measurement samples, our adaptive anomaly prediction approach employs an ensemble of prediction models. Different from previous model ensemble approach that trains models from different windows of consecutive training data, our approach induces prediction models from context-based grouped measurements. For example, for the context C_1, we train a prediction model $\mathcal{M}_1$ using training data $\{D_1, D_3, D_5\}$ in Figure 4. In contrast to previous model ensemble approach where different models are induced from windows of consecutive training data, our model is induced from intelligently grouped data that are not necessarily consecutive.

2.3 Adaptive Runtime Anomaly Prediction

During runtime, to decide if an alert should be raised based on system measurements at time t, we must first find out the context

C_k at time t, so that we can apply $\mathcal{M}_k$, the model corresponding to context C_k, on the system measurements to make the decision. However, finding the current context is a nontrivial problem, since instead of exhibiting simple patterns such as periodicity, context changes may occur at any time. For example, in Figure 4, the system component operates under changing execution contexts $C_1 \rightarrow C_2 \rightarrow C_1 \rightarrow C_3 \rightarrow C_1 \rightarrow C_3 \rightarrow C_2$. Our approach of finding the current context is based on Bayesian analysis. We collect the statistics of context changes in the measurement stream, and analyze how different contexts interact with each other. The result is a context switching model, which enables us to switch to the most-likely context giving cues from an online monitoring stream.

Let $P_t(C_k)$ denote the probability that C_k is the context at time t. Let Y_t denote the new measurement that arrives at time t. Let $p(Y_t|C_k)$ denote the probability that Y_t is generated under context C_k. Let $P_t^-(C_k)$ denote the *prior* probability of C_k being the context at time t, that is, the probability that C_k is the context at time t before we see Y_t. According to the Bayesian rule, we have[2]:

$$P_t(C_k) \propto P_t^-(C_k) \cdot p(Y_t|C_k) \quad (5)$$

Our goal is to find the context C_k that has the largest $P_t(C_k)$. To do this, it suffices to compute $p(Y_t|C_k)$ and $P_t^-(C_k)$. We approximate $p(Y_t|C_k)$ using the prediction error of $\mathcal{M}_k$. More specifically, let Err_k be the cross validation error of $\mathcal{M}_k$. Thus, if $\mathcal{M}_k$ predicts Y_t correctly, we have $p(Y_t|C_k) \propto 1 - Err_k$, otherwise, we have $p(Y_t|C_k) \propto Err_k$.

Probability $P_t^-(C_k)$ is computed recursively:

$$P_t^-(C_k) = \sum_i P_{t-1}(C_i)\chi(C_i, C_k) \quad (6)$$

where $\chi(C_i, C_k)$ is the probability of changing to context C_k when context C_i is previously active. Furthermore, we assume all contexts have the same probability to be the context at time $t = 1$. Finally, we need to find $\chi(C_i, C_j)$, which is derived from historical data:

$$\chi(C_i, C_j) = \frac{\text{number of transitions from context } C_i \text{ to } C_j}{\text{total number of context transitions}} \quad (7)$$

For example, in Figure 4, there are altogether six context transitions, with two occurrences of context C_3 following context C_1 (block d_6 to d_7 and block d_{11} to d_{12}). Thus, $\chi(C_1, C_3)$, the probability that context C_1 is followed by context C_3 is $2/6$.

In a dynamic hosting infrastructure, new context will emerge during runtime. In our system, we actively log data for which our prediction is unsatisfactory (e.g., data for which alert is raised but never lead to anomaly even without intervention, and also data for which alert is not raised but leads to anomalies). Those data may indicate the emergence of a new execution context that cannot be captured by our current prediction models. We can thus induce new contexts and models from those data. We also update the context switching model to incorporate these new models into the dynamic context switching pattern. At the same time, we may also remove inactive contexts (i.e., with zero switch probability) to keep the set of active contexts small. If the current prediction model consistently provides unsatisfactory prediction results, we will trigger the context-aware prediction model training algorithm described in Section 2.2 to induce a new set of prediction models based on new training data.

3. IMPLEMENTATION & EVALUATION

In this section, we evaluate our ALERT system. We first describe the system implementation. Then, we show how we collect several trace sets that contain anomaly data. Third, we present the experimental results to validate the prediction accuracy of our approach.

[2] $P_t(C_k)$ is a shorthand for $p(C_k|Y_t, \cdots, Y_1)$, and $P_t^-(C_k)$ a shorthand for $p(C_k|Y_{t-1}, \cdots, Y_1)$.

System S Metrics	Description
AVAILCPU	percentage of free CPU cycles
FREEMEM	available memory
PAGEIN/OUT	virtual page in/out rate
MYFREEDISK	free disk space
RXSDOS	num. of received data objects
TXSDOS	num. of transmitted data objects
DPSDOS	num. of dropped data objects
RXBYTES	num. of received bytes
TXBYTES	num. of transmitted bytes
QUEUELEN	input queue length
UTIME	process time spent in user mode
STIME	process time spent in kernel mode
ROUTING	system data handling time
VMSIZE	address space used by a component
VMLCK	VM locked by the component
VMRSS	VM resident set size
VMDATA	VM usage of the heap
VMSTK	VM usage of the stack
VMEXE	VM executable
VMLIB	VM libraries
Planetlab Metrics	**Description**
LOAD1	load in last 1 minute
LOAD5	load in last 5 minutes
AVAILCPU	percentage of free CPU cycles
MYFREEDISK	free disk space of my slice
DISKUSAGE	percentage of utilized disk space
FREEDISK	free disk space
FREEMEM	available memory
NUMSLICE	num. of registered slices on this host

Table 1: Subset of monitoring metrics.

3.1 System Implementation

We have implemented a prototype of the ALERT system and deployed it on two production hosting infrastructures: 1) IBM System S stream processing cluster [21, 22] that consists of about 250 IBM blade servers, each of which has dual Intel Xeon 3.2GHZ CPUs and 2 to 4 GB memory; and 2) PlanetLab wide-area network system testbed [33]. To achieve generality, the ALERT system is implemented based on standard Linux APIs, which allows us to port the ALERT system to different hosting infrastructure easily. We collect about 20 metrics on each host in IBM System S [23, 24] and about 66 metrics on each host in PlanetLab [4]. Table 1 lists a subset of key metrics collected by ALERT on System S and PlanetLab. All collected metrics are used to train the decision tree classifier.

To make ALERT practical for large-scale hosting infrastructures, we employ a fully decentralized monitoring and learning architecture, illustrated by Figure 1. We first deploy distributed monitoring sensors to collect various monitoring metrics. We then deploy a set of predictors that are associated with different system hosts/components. To avoid affecting normal application workloads, we strive to use *idle* resources in the hosting infrastructure to perform prediction and model training. We also install a set of anomaly detectors to continuously provide normal/anomaly class labels for new measurement samples that will be later used by the predictors as training data. Note that a set of measurement samples preceding the anomaly samples will be later labeled as *alert* by the predictor based on the alert interval. We will describe

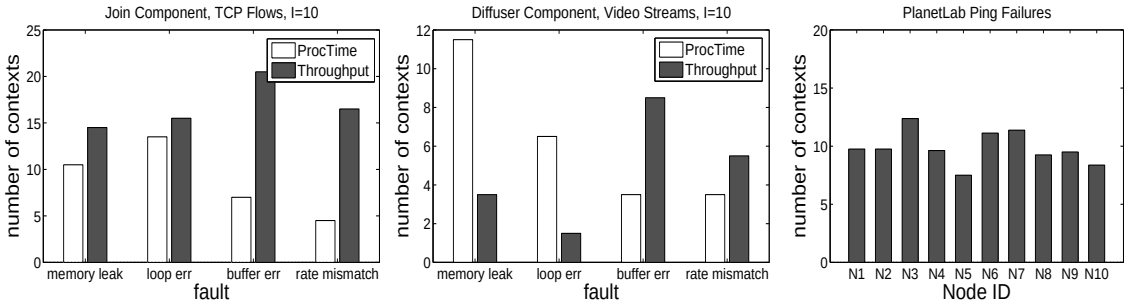


Figure 5: Number of execution contexts in dynamic systems.

the implementation details of the anomaly detectors for different applications in Section 3.2.

3.2 Experiment Setup

We first collected measurement traces on the IBM System S cluster. We deploy monitoring sensors on all hosts and continuously collect various metrics with a sampling rate of one sample every two seconds. We install a set of predicate-based anomaly detectors in the system to catch anomaly incidents: 1) *ProcTime anomaly* when the average per-tuple processing time of the component exceeds a certain threshold (e.g., 4 seconds for a join component and 1200 milliseconds for a diffuser component) ; and 2) *Throughput anomaly* when the output rate of the component is lower than a threshold (e.g., 30 tuples/second), or the ratio between the output rate and the input rate is lower than a threshold (e.g., 0.4).

In our experiments, we run the System S reference application [21] that consists of about 50 distributed stream processing components performing complicated multi-modal stream analysis. The reference application shares resources with other applications started by different infrastructure users. Each trace includes about 5000 measurement samples. To trigger anomalies, we inject various faults in different components at different time instants to test the adaptability of our anomaly prediction model for time-varying workloads. We employed a set of common faults in stream systems: 1) *memLeak*: the component executes a buggy code segment that keeps forgetting to free memory; 2) *loopErr*: the component spawns a CPU-bound thread that includes an infinite loop error (i.e., iterator update mistake); and 3) *bufferErr*: the component forgets to remove processed data units from its input buffers. Each fault lasts 50 seconds: we activate the fault at time t and remove the fault at time t + 50. We report the results on two commonly used stream processing components [25]: 1) a join component that continuously correlates tuples from different wide-area TCP traffic flows based on a pre-defined join condition, and 2) the diffuser component dispatches data items in news video streams to different hosts for load balancing.

We have started to collect measurement traces on the PlanetLab since July 2009 [4]. In our experiments, we deploy monitoring sensors on about 400 PlanetLab nodes and collect various host-level metrics with a sampling rate of one sample every 10 seconds. We deploy a set of anomaly detectors to catch host ping failures: a monitored host does not respond to five successive ping trials. Each failure detected in this way is recorded with timestamp information. The system also collects about 66 monitoring metrics [3] on the failed hosts to capture pre-failure behavior. Each host ping failure trace consists of 4000 to 6000 samples.

In each trace, ALERT detects multiple execution contexts, which indicates real applications do exhibit evolving behavior. The execution context change is caused by time-varying input data rates and

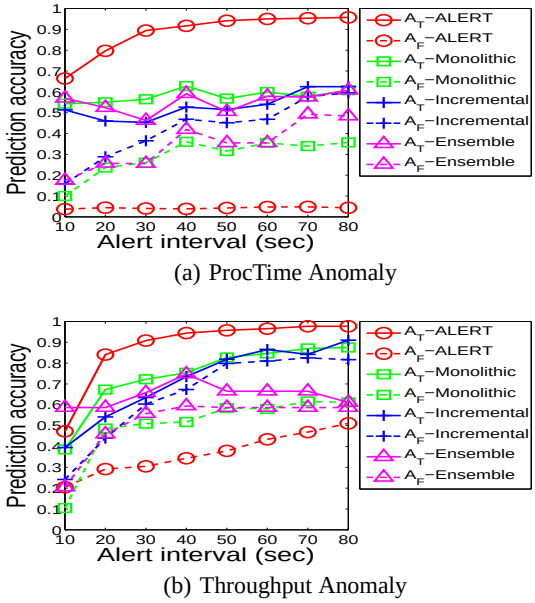
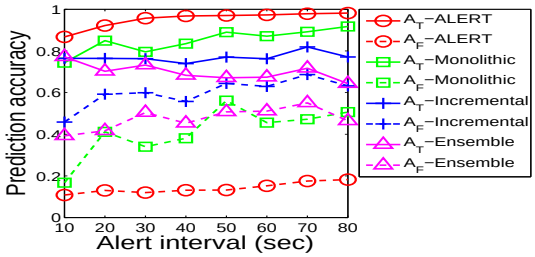


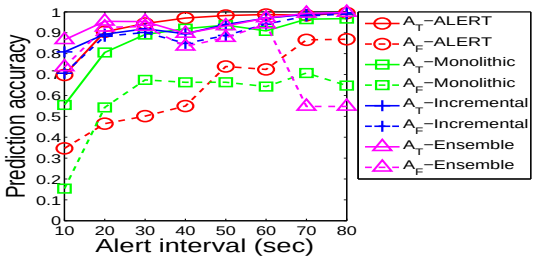
Figure 6: True positive rate (A_T) and false alarm rate (A_F) for the IBM System S Join component with the memory leak fault.

fluctuating resource availability on the shared hosting infrastructure. Figure 5 shows the number of contexts discovered in different IBM System S trace data. We observe that the join component processing TCP traffic streams experiences more dynamic execution context than the diffuser component processing the video streams. As we will show next, the anomaly predictor generally has lower prediction accuracy for more dynamic components. The reason is that with a larger number of different contexts, the predictor is less certain to find the right context in evolving systems, and each context is trained with fewer data. We also observe 8 - 12 execution contexts discovered in different PlanetLab trace data.

For comparison, we implemented three commonly used existing learning algorithms: 1) the *monolithic* scheme that trains a single anomaly prediction model using the entire training data; 2) the *incremental* scheme that incrementally builds and updates the prediction model using a sliding window of recent measurements; and 3) the *ensemble* scheme [45] that maintains an ensemble of prediction models, each of which is trained using different windows of training data. At any moment, the best performed model based on the *balanced accuracy* [45] in the past window of data is used to predict anomaly in the current window of data. Each window includes 500 samples. In all approaches, we use the same decision tree classifier to build the triple-state anomaly prediction model.

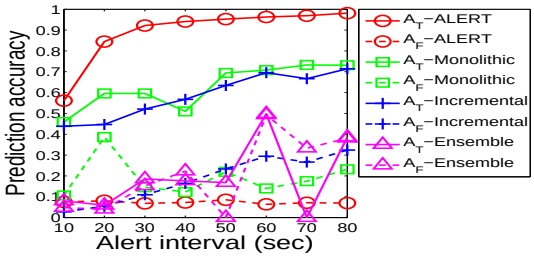


(a) ProcTime Anomaly

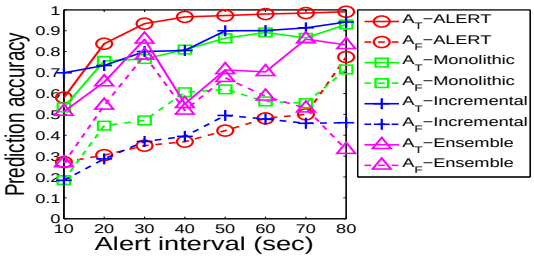


(b) Throughput Anomaly

Figure 7: True positive rate (A_T) and false alarm rate (A_F) for the IBM System S Join component with the loopErr fault.



(a) ProcTime Anomaly



(b) Throughput Anomaly

Figure 8: True positive rate (A_T) and false alarm rate (A_F) for the IBM System S Join component with the bufferErr fault.

We adapt the widely-used decision tree software package C4.5 [2] to make it work in online learning and classification. We use standard evaluation metrics: true positive rate (A_T) and false alarm rate (A_F) denoted by Equation 1 to compare the performance of different learning algorithms. We say the prediction model makes a true positive prediction if it raises an anomaly alert at time t_1 and the anomaly indeed happens at time $t_2, t_1 < t_2 < t_1 + I$. Otherwise, we say the prediction model fails to make a correct prediction. If the predictor raises an alert and the predicted anomaly does not happen after a period of time (e.g., the alert interval), we say that the prediction model raises a false alarm.

3.3 Results and Analysis

In our first set of experiments, we evaluate the prediction accuracy for the performance anomalies of a join component and a diffuser component in the IBM System S. There are six replicated join components running on different hosts in our application. They perform the same operation on similar workload so that they exhibit similar but not identical behavior. In each experiment run, we choose one component as the training set and predict the anomalies of the other five replicated components. Under the same configuration, We swap their roles for six times so that each component is used for training exactly once. All the following experimental results of the join component are aggregated in this way. For the diffuser component, we did not have replica in our application so we use half of the collected data as the training data to predict the anomalies of the other half data.

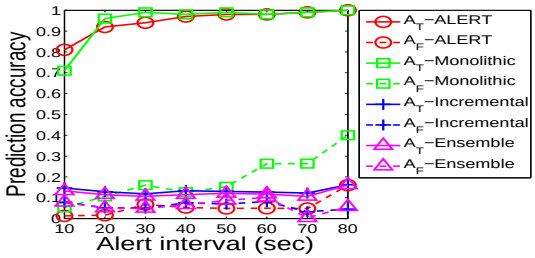
For each algorithm, we show both the true positive rate (A_T) and false alarm rate (A_F) which are calculated for the whole trace data. An ideal predictor should have 100% true positive rate and zero false alarm rate. To show the impact of the alert interval on the sensitivity of the prediction model, we repeat our experiments using a range of alert intervals I = 10, 20, 30, 40, 50, 60, 70, and 80 seconds. Figure 6(a) and Figure 6(b) show the true positive rate A_T and false alarm rate A_F for the faulty join components containing the memory leak fault. The X-axis shows different alert intervals used by the prediction models and the Y-axis shows the true positive rate and false alarm rate achieved by different prediction methods. The results show that ALERT consistently achieves much higher true positive rate and lower false alarm rate than other alternative methods. This confirms that context-aware anomaly predictions are necessary for real application workloads.

We also observe that the alert interval I can indeed be used as a control knob to tune the sensitivity of the prediction model. Generally speaking, as we increase the alert interval, the derived prediction model is more sensitive with both increased true positive rate and false alarm rate. This provides the opportunity for us to achieve tunable anomaly management. Based on the benefit achieved by accurate predictions and cost for handling false alarms, we can configure the prediction model with a proper alert interval to achieve optimal prediction reward. We also observe that different anomaly types exhibit varied predictability. In this case, the ProcTime anomaly is easier to predict than the throughput anomaly that has high false alarm rate. Note that the premise of anomaly prediction is that the anomaly exhibits gradual pre-anomaly symptom. Thus, if the anomaly does not have prominent pre-anomaly symptoms, the predictor will have high errors.

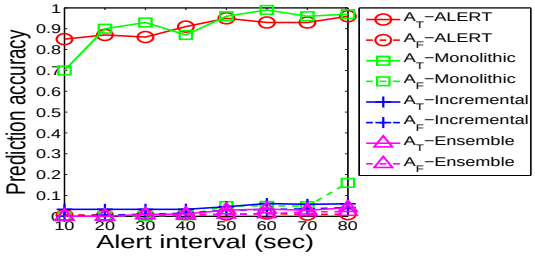
For the ensemble approach, we sometimes observe ups and downs for both true positive rate and false alarm rate as the alert interval I increases. The reason of such fluctuation is that we use the balanced accuracy as the criteria to select the best prediction model from the ensemble, which is defined as $BA = (A_T + 1 - A_F)/2$. Therefore, if we look at the true positive rate individually, it is possible that the true positive rate of a larger I is lower than that of a smaller I. The essence here is to know that the ensemble approach performs worse than ALERT even in terms of the balanced accuracy.

Figure 7(a) and Figure 7(b) show the prediction accuracy results for the join components containing the loopErr fault. Again, we observe that our algorithm can achieve much better performance than all other schemes. However, the anomalies caused by the loopErr fault are more difficult to predict than those caused by the memLeak fault in terms of high false alarm rate. The reason is that the effect of the loopErr fault is more sudden than that of the memLeak fault, which makes it less predictable. Particularly, all algorithms have high false alarm rates for the throughput anomaly, which make it unpredictable.

Figure 8(a) and Figure 8(b) show the prediction results for the join components containing the bufferErr fault. The results again



(a) ProcTime Anomaly



(b) Throughput Anomaly

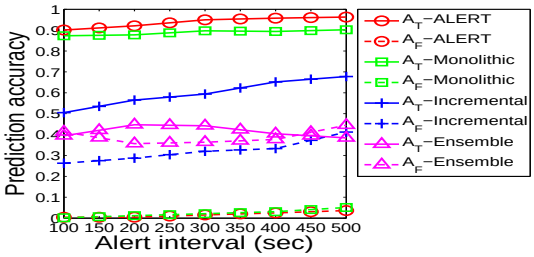
Figure 9: True positive rate (A_T) and false alarm rate (A_F) for the IBM System S Diffuser component with the memory leak fault.

show that our approach consistently achieves higher true positive rate and lower false alarm rate than other learning algorithms. Although different anomaly types exhibit varied predictability, our prediction model can consistently achieve much better prediction accuracy. Similar to the previous cases, the throughput anomaly is more difficult to predict than the ProcTime anomaly.

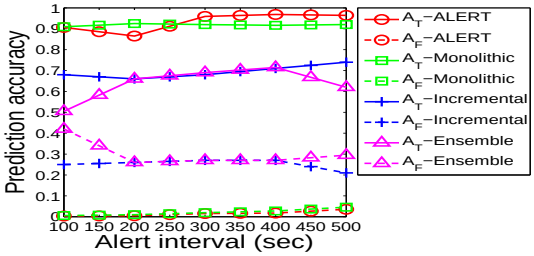
We now evaluate our anomaly prediction approach using a different type of application component called diffuser. Compared to the join component, the diffuser component has simpler application semantics: it dispatches input data to different hosts based on their load conditions. Second, the diffuser processes a different input stream workload: news video streams that exhibit less rate variation than the network traffic streams. Due to the space limitation, we only show a subset of our results. Figure 9(a) and Figure 9(b) show the prediction accuracy results for a faulty diffuser with the memory leak fault. We observe that our scheme still consistently outperforms other schemes. The Monolithic approach achieves similar true positive rate with our scheme but incurs higher false alarm rate. The Incremental and Ensemble approaches have very low true positive rate, which is caused by learning from conflict training data contained in the fixed-length of windows.

We now present the anomaly prediction results for the ping failure on the PlanetLab hosts. For each host, we use half of the trace data as the training data and predict the ping failures in the other half of the trace data. Figure 10(a) shows the average prediction accuracy for all ten failed hosts during our experiments. Figure 10(b) and Figure 10(c) shows the prediction accuracy for two specific failed hosts. The results show that the ping failure on PlanetLab shows good predictability and our approach can achieve high prediction accuracy with over 90% true positive rate and near zero false alarm rate. In contrast, other alternatives achieve much lower true positive rate and higher false alarm rate. After examining the trace data, we found that a set of metrics such as LOAD1 and AVAILCPU exhibit significant difference between normal samples and anomaly ones, and those metrics change gradually.

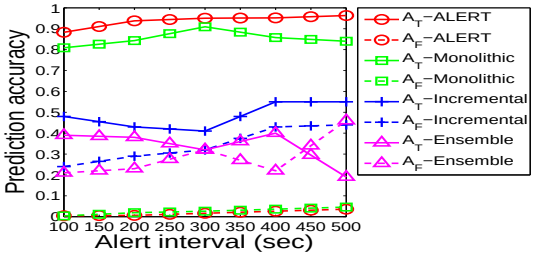
We also measure the prediction lead time (i.e., how early ahead can the prediction model raise an alert) achieved by the ALERT system. Due to the space limitation, we only show a subset of all



(a) Average prediction accuracy



(b) Prediction accuracy on host A



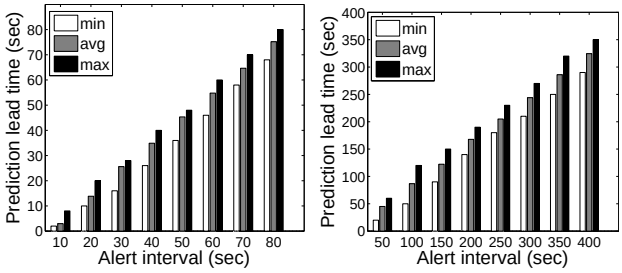
(c) Prediction accuracy on host B

Figure 10: True positive rate (A_T) and false alarm rate (A_F) for Ping failure prediction accuracy on the PlanetLab.

results. Figure 11(a) shows the minimum, average, and maximum prediction lead time achieved by ALERT for the IBM System S anomalies. We observe that the alert interval affects the lead time. As we increase the alert interval, the prediction model tends to raise alerts earlier since more measurement samples will be included in the alert state. The tradeoff here is that larger alert interval may also incur higher false alarm rate, as shown in previous figures. Similarly, Figure 11(b) shows the prediction lead time for the PlanetLab host failure. The results indicate that ALERT can achieve tens of seconds or several minutes lead time to allow just-in-time anomaly diagnosis and correction.

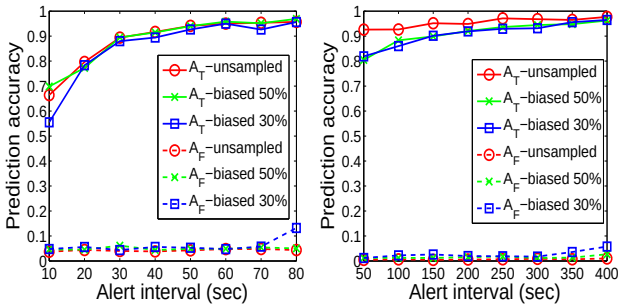
One design objective of the ALERT system is to achieve scalable online anomaly learning and prediction. To achieve the goal, we employ reservoir biased sampling [23] to reduce the measurement sampling overhead. One question is whether the biased sampling can greatly affect the prediction accuracy. We repeat the above experiments using biased sampling that retained 50% and 30% of the total measurements. Due to space limitation, we only show a subset of results shown by Figure 12(a) and Figure 12(b). We observe that biased sampling can maintain similar prediction accuracy while reducing the sampling overhead.

We now evaluate the overhead of the ALERT system. Figure 13(a) shows the cumulative distribution function (CDF) of model training time collected in different experiment runs. The training time includes the time for discovering contexts and building decision tree ensembles corresponding to different contexts. We compare the model training time using full measurement samples with



(a) System S (b) PlanetLab

Figure 11: ALERT prediction lead time.



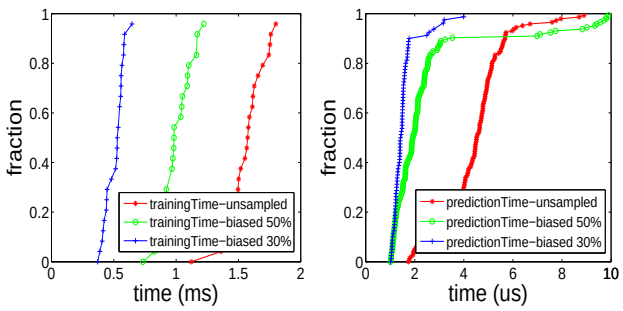
(a) Join component - ProcTime Anomaly (b) PlanetLab - Ping Failure on host A

Figure 12: True positive rate (A_T) and false alarm rate (A_F) under different sub-sampling rates.

the training time using a subset of samples obtained by reservoir sampling. With full measurements, the model training time is in the range of [1,2] milliseconds. By retaining 30% samples, we can reduce the training time to [0.3, 0.6] millisecond. Figure 13(b) shows the CDF of mean prediction time collected in different experiment runs. The results show that our prediction algorithm is fast, which requires less than four microseconds for the model trained using 30% biased samples. The results indicate that ALERT can indeed support online anomaly prediction with realtime evaluation speed. We also measured the cost of online system monitoring, which generally imposes less than 1% load on the monitored host.

4. RELATED WORK

System anomaly detection and debugging have been extensively studied. For example, Noble et al. proposed an anomaly detection algorithm for graph-based data in which system predictability is quantified as graph regularity [31]. Shen et al. proposed a change profile based approach to detect system anomaly symptoms by checking performance deviation between reference and target execution conditions [37]. Wang et al. proposed the PeerPressure to automatically troubleshoot system misconfigurations by checking the status of machines running the same application in the database [41]. Bhatia et al. proposed “sketch" data monitoring structure and correlated anomaly symptoms by predefined rules [10]. Oznat proposed an information-theoretic approach to detect anomalies in the metric behavior for web services [32]. Cherkasova et al. built regression-based transaction models and application performance signatures to detect anomalous application behaviors [15]. Guo et al. explored method of probabilistically correlating monitoring data for failure detection in complex systems [26]. Magpie [9] is a request extraction and workload modeling tool that can record fine-grained system events and correlates these events using an application specific event schema to capture the control



(a) Training time (b) Prediction time

Figure 13: ALERT system computation cost.

flow and resource consumption of each request. Pinpoint [13] also takes a request-oriented approach to tag each call with a request ID and diagnose faults by applying statistical methods to identify components that are highly correlated with failed requests. Aguilera et al. proposed a black box performance debugging technique by analyzing message-level traces of system activities to infer causal paths [8]. Triage [39] leverages lightweight re-execution support to deal with system bugs without requiring an intervention from a programmer. Different from previous work, our research focuses on applying machine learning techniques to achieve advance anomaly prediction instead of post-anomaly detection.

Our work is closely related to the Tiresias system [42] that also addresses the black-box failure prediction problem in distributed systems. Different from the Tiresias system that relies on anomaly detection over individual performance metrics to achieve system state prediction, our work provides predictions using triple-state whole system classification that can easily achieve tunable tradeoff between true positive and false alarm rates. Furthermore, Tiresias does not consider execution context changes in dynamic distributed systems. In [23, 24], we have presented an initial design of our online anomaly prediction system and time-to-anomaly estimation, which, however, do not address dynamic computing context changes that are common in many real-world systems such as stream processing systems.

Recently, machine learning methods have been shown to be promising for autonomic failure management. Much previous work focuses on offline system log analysis (e.g., [29, 30, 35, 36, 40]). Xu et al. developed online console log analysis techniques to detect system problems [44]. In contrast, our research focuses on online characterization of system anomalies using performance and resource metrics. Power et al. investigated the performance prediction power of different statistical learning approaches [34]. Cohen et al. applied Tree-Augmented Bayesian Networks (TAN) to perform performance diagnosis [16], and proposed the signature concept to capture the essential characteristic of a system state [17]. Zhang et al. extended the TAN model and proposed to use ensembles of models for diagnosing performance problems [45]. The Fa system employs various machine learning techniques to achieve automatic failure diagnosis for query processing systems [19]. Shen et al. proposed to construct a whole-system I/O throughput model as the reference of expected performance and used statistical clustering and characterization of performance anomalies to guide debugging [38]. Different from previous work, our research focuses on applying self-evolving learning methods to achieve adaptive runtime anomaly prediction for large-scale hosting infrastructures.

5. CONCLUSION

In this paper, we have presented the ALERT system that provides adaptive runtime anomaly prediction system for large-scale host-

ing infrastructures. ALERT provides a tunable anomaly prediction model and employs self-evolving learning algorithm to adapt to dynamic hosting infrastructures. To the best of our knowledge, our work makes the first attempt to achieve context-aware anomaly prediction for dynamic distributed systems. We have implemented the ALERT system and deployed it on several production hosting infrastructures. We learned the following lessons from our prototype implementation: 1) a range of system anomalies do exhibit predictability; 2) ALERT can achieve much better prediction accuracy than existing learning methods for dynamic systems; and 3) ALERT can provide realtime prediction performance while imposing low overhead to the hosting infrastructure.

6. ACKNOWLEDGMENT

This work was sponsored in part by NSF CNS-09-1-5567, NSF CNS-09-1-5861, U.S. Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI), IBM Exploratory Stream Analytics Award, and IBM Faculty Award. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF, ARO, or U.S. Government.

7. REFERENCES

[1] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[2] C4.5 Release 8. *http://www.rulequest.com/Personal/*.

[3] CoMon. http://comon.cs.princeton.edu/.

[4] InfoScope Distributed Monitoring System. http://dance.csc.ncsu.edu/projects/infoscope/index.html.

[5] PlanetLab. https://www.planet-lab.org/.

[6] The STREAM Group, STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin, 26(1):19-26*, Mar. 2003.

[7] D. J. Abadi and et al. The Design of the Borealis Stream Processing Engine. In *Proc. of CIDR*, 2005.

[8] M. K. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. of ACM SOSP*, 2003.

[9] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. of OSDI*, 2004.

[10] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proc. of OSDI*, pages 103–116, 2008.

[11] J. Breese and R. Blake. Automating computer bottleneck detection with belief nets. In *Proc. of UAI*, pages 36–45, San Francisco, CA, 1995. Morgan Kaufmann.

[12] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot- A Technique for Cheap Recovery. In *Proc. of OSDI*, Dec. 2004.

[13] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *Proc. of NSDI*, 2004.

[14] S. Chen, H. Wang, S. Zhou, and P. S. Yu. Stop Chasing Trends: Discovering High Order Models in Evolving Data. In *Proc. of ICDE*, 2008.

[15] L. Cherkasova, K. M. Ozonat, N. Mi, J. Symons, and E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Proc. of DSN*, pages 452–461, 2008.

[16] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proc. of OSDI*, 2004.

[17] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proc. of SOSP*, 2005.

[18] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of OSDI*, Dec. 2004.

[19] S. Duan, S. Babu, and K. Munagala. Fa: A System for Automating Failure Diagnosis. In *Proc. of ICDE*, 2009.

[20] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Detecting Past and Present Intrusions Through Vulnerability-Specific Predicates. In *Proc. of SOSP*, Oct. 2005.

[21] K.-L. W. et al. Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S. In *Proc. of VLDB*, 2007.

[22] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the system s declarative stream processing engine. In *Proc. of SIGMOD*, 2008.

[23] X. Gu, S. Papadimitriou, P. S. Yu, and S. P. Chang. Toward Predictive Failure Management for Distributed Stream Processing Systems. In *Proc. of ICDCS*, 2008.

[24] X. Gu and H. Wang. Online Anomaly Prediction for Robust Cluster Systems. In *Proc. of IEEE ICDE*, 2009.

[25] X. Gu, P. S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In *Proc. of ICDE*, 2007.

[26] Z. Guo, G. Jiang, H. Chen, and K. Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *Proc. of DSN*, pages 259–268, 2006.

[27] R. Jin and G. Agrawal. Efficient decision tree construction on streaming data. In *Proc. of KDD*, 2003.

[28] E. Kiciman and A. Fox. Detecting Application-Level Failures in Component-based Internet Services. *IEEE Transactions on Neural Networks*, 2005.

[29] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. Failure Prediction in IBM BlueGene/L Event Logs. In *Proc. of ICDM*, 2007.

[30] J. F. Murray, G. F. Hughes, and K. Kreutz-Delgado. Comparison of machine learning methods for predicting failures in hard drives. *Journal of Machine Learning Research*, 2005.

[31] C. C. Noble and D. J. Cook. Graph-based anomaly detection. In *Proc. of KDD*, pages 631–636, Aug. 24–27 2003.

[32] K. M. Ozonat. An information-theoretic approach to detecting performance anomalies and changes for large-scale distributed web services. In *Proc. of DSN*, pages 522–531, 2008.

[33] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proc. of HotNets-I*, Princeton, New Jersey, October 2002.

[34] R. Powers, M. Goldszmidt, and I. Cohen. Short term performance forecasting in enterprise systems. In *Proc. of KDD*, pages 801–807, 2005.

[35] R. K. Sahoo and et al. Critical event prediction for proactive management in large-scale computer clusters. In *Proc. of ACM SIGKDD*, 2003.

[36] B. Schroeder and G. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean too you? In *Proc. of FAST*, 2007.

[37] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *Proc. of SIGMETRICS/Performance*, pages 85–96, 2009.

[38] K. Shen, M. Zhong, and C. Li. I/o system performance debugging using model-driven anomaly characterization. In *Proc. of FAST*, 2005.

[39] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing Production Run Failures at the User's Site. In *Proc. of SOSP*, 2007.

[40] R. Vilalta, C. V. Apte, J. L. Hellerstein, S. Ma, and S. M. Weiss. Predictive algorithms in the management of computer systems. *IBM Systems Journal*, 2002.

[41] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *Proc. of OSDI*, pages 245–258, 2004.

[42] A. W. Williams, S. M. Pertet, and P. Narasimhan. Tiresias: Black-box failure prediction in distributed systems. In *Proc. of IPDPS*, 2007.

[43] I. H. Witten and E. Frank. *Data Mining : Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[44] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Large-scale system problems detection by mining console logs. In *Proc. of SOSP*, 2009.

[45] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensemble of models for automated diagnosis of system performance problems. In *Proc. of DSN*, 2005.